An Optimal *in–Situ* Disk Sorting Algorithm Based on Heapsort

Preliminary Version

Jiří Wiedermann, Jiří Šíma, Roman Neruda *

Department of Theoretical Informatics, Institute of Computer Science

Czech Academy of Sciences

Pod vodárenskou věží 2, 182 07 Prague 8

Czech Republic

e-mail: {wieder|sima|roman}@uivt.cas.cz

April 14, 1997

Abstract

A versatile variant of Heapsort, adapted for external disk sorting, is presented. On a single processor RAM with internal memory capacity $\Theta(m + kbw)$ words, it sorts a file of n records of length w words, stored in m physical blocks, each consisting of b records, on a disk, using $O(m \log m / \log k)$ input/output operations, and $O(n \log n)$ internal key comparisons, for any $k \geq 2$. For internal memory of capacity s words, with $s = \Theta(m) = \Theta(kbw)$, the previous estimate is asymptotically optimal. However, unlike the other known disk sorting algorithms of comparable time efficiency, the new algorithm sorts the file *in–situ* — i.e., without using any other additional external working files. Moreover, a simple general algorithm is proposed that on a multiprocessor machine consisting of p RAMs of the previous type, connected in a line, each of them equipped by its own external disk memory, enables to achieve within a certain range a linear speed–up of any external sorting algorithm of superlinear I/O complexity. Especially, for our algorithm, a p–fold speed–up for $1 \leq p \leq O(\log m / \log k)$ will be achieved.

1 Introduction

It is a well known fact that standard algorithms for internal sorting cannot be used straightforwardly for efficient external sorting, since there is a major difference between the two underlying models of computation.

*This research was partially supported by EC Cooperative Action IC-1000 (Project ALTEC: *Algorithms for Future Technologies)*

While, in the former case, the entire file to be sorted fits into the internal memory, and hence the cost of sorting is completely determined by the cost of all instructions performed within the internal memory during sorting process, in the latter case, the file — being too large to fit into internal memory — resides in the secondary storage, nowadays typically made up of one or more magnetic disks. This brings up the necessity to transfer the data from the file to internal memory, and vice versa, in a series of contiguous so–called *physical blocks*, that in the typical case are on the order of several kilobytes, and hence the file must be processed *per partes*. To read/write a series of physical blocks, it is necessary to move the disk read/write head to the proper place (track) on a disk (to perform so–called *seek*), and to transfer the required amount of data. The time complexity of this operation is dominated by the seek operation that is usually much longer than the transfer time, that in turn is, by several order of magnitudes, longer than any instruction performed in the internal memory.

Hence, the time complexity of any algorithm working with data stored in external files and making frequent accesses to this data — what, as it is known is the case with external sorting — is governed by the cost of these accesses, i.e., by the cost of input/output (I/O) communications.

The previous model of computation with secondary storage, where internal computation time is ignored, has been considered by Aggarwal and Vitter [3], and by Vitter and Shriver [9], who have shown that the worst case, and the average case number of I/Os required for external sorting is

$$\Theta\Big(\frac{m}{p}\frac{\log m}{\log\left(s/(bw)\right)}\Big)$$

where m denotes the number of physical blocks, each of size b records, each consisting of w words, in the file of a total of n records (i.e., $n = mb$), p is the number of internal processors (in case of a parallel p–processor RAM), and s is internal memory size, measured in terms of words, for $1 \le b \le s/w < n$ and $1 \le p \le \lfloor s/w \rfloor$. [1] Further, we assume that the size of a word in internal memory is sufficient to be able to address the entire file — i.e., word size should be of order $\Theta(\log m)$.

In these, and in their subsequent paper [7], the authors have designed sorting algorithms (both randomized and deterministic ones), based on variants of merge and distribution sorting, that achieved optimal I/O bounds within the above model, and some of them even on more generalized memory hierarchies [1, 2]. In the design of these algorithms, their space complexity was not an issue and, unfortunately, as it appears, none of these algorithms is optimal from this particular point of view — i.e., none of them is an in–place sorting algorithm (except internal memory, and for the original input file they make use of other external working files).

Surprisingly, it seems that in the published works not even a sequential, I/O optimal, *in–situ* disk sorting algorithm has been described. There are few notable ex-

[1] Unlike in [3] and [9], where the internal memory capacity is measured in terms of records, we prefer to measure the memory capacity in words, as it is customary in practice.

ceptions from the early eighties in which sequential *in–situ* disk sorting algorithms have been proposed that are I/O optimal for the case of "small" internal memories, i.e., for the case $s = \Theta(bw)$ [8, 10], and there is one recent proposal [4] of an *in–situ* sorting algorithm, based on a bubble sort, with I/O complexity of order $\Theta(m^2)$ (!).

Therefore, it appears to be useful still to pay attention first to sequential, I/O optimal, *in–situ* disk sorting algorithms, and then to try to extend the results obtained to the parallel case while preserving, of course, the minimal external space requirements of underlying algorithms. Namely, in very large scale external sorting applications that we have primarily in mind, the availability of extra space could present the borderline between feasibility and unfeasibility: with the given fixed capacity of external memory devices, we are often in situations when we cannot afford to double space requirements, while reasonable (not asymptotical) prolongation of sorting time could be fully acceptable.

In the present paper, we shall design an external disk sorting algorithm based on Heapsort, that, making efficient use of internal memory, achieves optimal I/O performance and, at the same time, unlike the previously mentioned algorithms, is an in–situ algorithm that works without using any additional external working files.

The resulting algorithm seems to be the first known external I/O optimal sorting algorithm based on Heapsort: it works over the data structure called *multiple block heap*, that is a generalization of standard heaps used in internal sorting. It further combines design techniques known from merging, heapsorting, treesorting, and file rearrangement with the efficient disk and internal memory management.

Also, due to the idea of selection sort used in its design, the new algorithm possesses some desirable features not shared by other algorithms based on merging or distribution sorting: namely, it starts to produce blocks of elements in sorted order soon after the start of sorting process (precisely speaking, immediately after building the corresponding block heap), and when necessary, the sorting process can be stopped after producing some prescribed number of the first elements in sorted order, without the necessity of sorting the entire file.

Instead of considering a special parallel version of our algorithm, a general construction is proposed that enables to achieve a linear speed–up, proportional to the number of processors, of **any** sorting algorithm, on a realistic model of a parallel multiprocessor, modeled by the RAMs of the above mentioned type, connected in a line, each of them equipped by its own disk memory. The *in–situ* property of underlying sorting algorithm can be preserved. The method used is based on the idea of parallel odd-even external merge. This result seems to be of independent interest.

The structure of the paper is as follows: section 2 contains the definition of a multiple block heap, and section 3 describes a high level (sequential) algorithm for its creation. Its efficient implementation and complexity analysis is presented in section 4 and 5, respectively. Then, in section 6 an external sorting algorithm based on ideas of internal heapsorting is considered and shown not to be efficient enough. Therefore, in section 7 a final efficient sequential algorithm based on tree–sorting is presented and analyzed. Section 8 contains a brief description of a general method by which any sequential sorting algorithm can be sped–up linearly w.r.t. the number of processors of the underlying parallel machine model. Conclusions and open problems are mentioned in section 9.

The full version of the paper will be available as a technical report. Here also the first experimental results concerning the "practical" efficiency of the new algorithm will be reported.

2 Multiple Block Heaps

The design of our sorting algorithm is inspired by the well–known in–situ internal sorting algorithm Heapsort. However, the underlying data structure over which the sorting works is generalized in two ways:

- first, each node of the heap is enlarged to contain roughly as many records as can fit into the internal memory;
- second, the usual binary heap is modified so that each internal node can possess up to k descendants, i.e., a k–ary heap is being used, for the largest k such that $\Theta(k)$ physical blocks will fit into internal memory.

In what follows, some familiarity with the standard heapsorting will therefore be welcomed, although not necessary.

Suppose now that the ordered (multi)set S to be sorted contains n elements. W.l.o.g., in the definition that follows we shall not distinguish between items to be sorted, and their keys, and we shall make them equivalent to elements of S.

A *k–ary block heap* with load factor $c > 0$, representing the ordered multiset S of size $n = qc$, is a complete k–ary tree T of q vertices, of height h such that

- all leaves in T are in depth h, or $h-1$, respectively;
- all leaves of depth h are to the left of all leaves of depth $h-1$;
- to each vertex v from T a set $S_v \subseteq S$ — a so–called *logical block* — of c elements from S is assigned in the way that the following *heap invariant* is satisfied: *if v is a descendant of some f in T, then for any $x \in S_v$, $y \in S_f$, $x \leq y$ holds*;
- elements in every logical block are sorted in nondecreasing order.

A k–ary heap of height h is called *a perfect heap* if it contains exactly k^i vertices at level i, for $i = 0, 1, \ldots, h$, i.e., altogether $q = 1 + k + k^2 + \ldots + k^h = (k^{h+1} - 1)/(k-1)$ vertices.

Note that any multiple block heap can be represented implicitly (i.e., without using any pointers) in the random access file, similarly as binary heaps in an array, in the following way.

Let the set S to be sorted consist of n records, let the records be grouped into m physical blocks consisting of b records, for some $b > 0$, and let r physical blocks create one logical block, for some $r > 0$. Then we clearly have $n = mb = qrb = qc$. The corresponding random access file, representing S, can be declared (in a Pascal—like language) e.g. as

type
  item: **record** *key*:*integer*; *info*:...**end**;
  physical_block: **array** [1..b] **of** *item*;
  logical_block: **array** [1..r] **of** *physical_block*;
var
  F: **file** [1..q] **of** *logical_block*;

Then, if S is represented as a k–ary block heap T with load factor c, as described above, and the nodes of T are numbered from 1 to q in a natural way, level by level, from left to right, in top–down manner, then the root of T is placed at $F[1]$, and the j–th son of a node numbered by i is located at $F[k(i-1)+j+1]$, for $j = 1, 2, \ldots, k$, as long as $k(i-1)+j+1 \leq q$.

In Fig. 1 an example of a ternary heap, with load factor 8, representing an 45 element set S, and its direct access file representation, is depicted.

3 Creating a Multiple Block Heap

First of all, we shall describe in a more detail the basic heap creation algorithm, without paying much attention to efficient use of internal memory. Subsequently, we shall sketch how this algorithm can be implemented *in situ*, using only a reasonable amount of internal memory.

To simplify the following exposition we shall assume that the heap to be built is a perfect heap.

A k–ary perfect block heap T of height h with the load factor c is being built from the file of (sorted) logical blocks in the following way:

- if $h = 0$ then T consists of a single logical block;
- if $h > 0$, then T is built recursively from k k–ary block heaps $T_1, \ldots, T_k$ of height $h-1$, and from the next vertex v containing logical block S_0 as follows:
  - v is assigned as the root of a new heap T, and roots of $T_1, \ldots, T_k$ are made sons of v. Then it is verified, whether heap invariant is preserved in v. In the positive case, we are done, otherwise we proceed in as follows:

Let $S_1, \ldots, S_k$, respectively, be the sets contained in the logical blocks in the roots $v_1, \ldots, v_k$ of heaps $T_1, \ldots, T_k$, respectively. Denote $s_i = \min\{S_i\}$, for $i = 0, 1, \ldots, k$, and

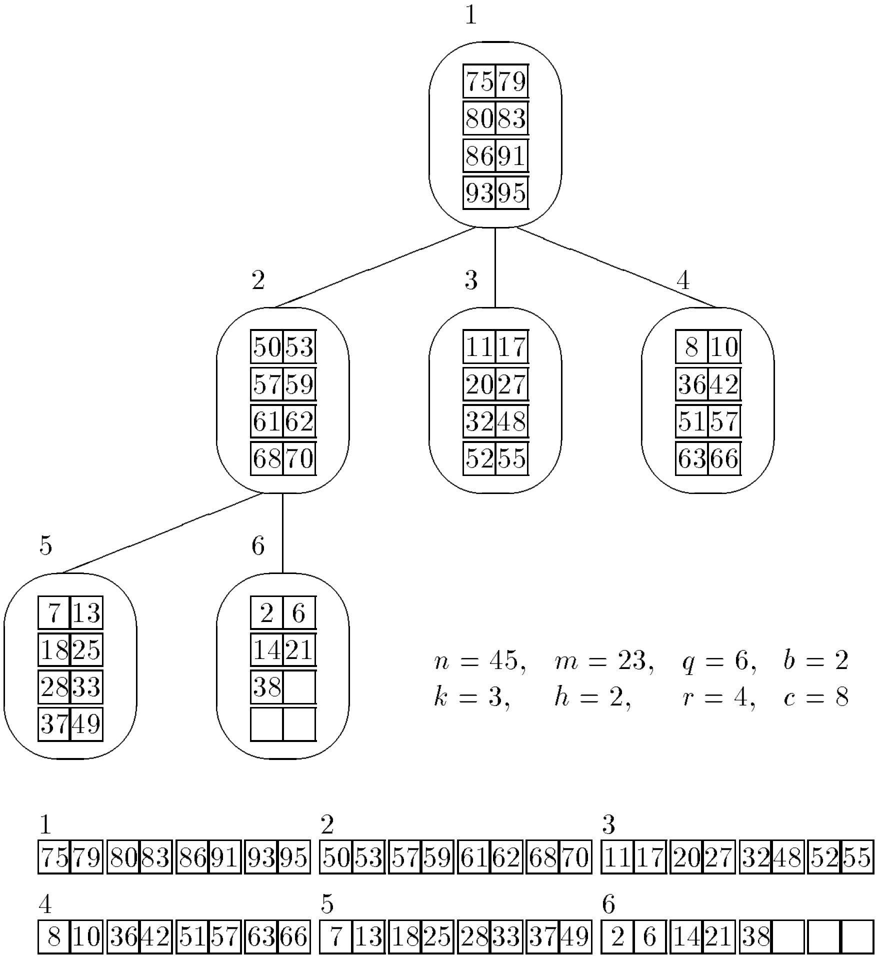

Fig. 1 : A ternary block heap and its representation in a random access file

let $\pi = (\pi(0), \ldots, \pi(k))$ be the permutation of the elements of the set $\{0, 1, \ldots, k\}$ (i.e., $\pi(i) \in \{0, 1, \ldots, k\}$ for $0 \leq i \leq k$, and $\pi(i) \neq \pi(j)$, for $i \neq j$) such that $s_{\pi(0)} \leq \ldots \leq s_{\pi(k)}$.

Create $S_0 \cup S_1 \cup \ldots \cup S_k = U$ of cardinality $(k+1)c$, sort it in nondecreasing order, using a $k+1$–way merge sort (remember that elements within each logical block are sorted), and split the resulting sequence Σ into segments $\sigma_0, \ldots, \sigma_k$ of length c each.

Let $t_i = \min \sigma_i$ for $i = 0, 1, \ldots, k$, let $pos(x, \Sigma) = j$ iff x finds itself at the j–th position in the sequence Σ, for any $x \in U$. Then, clearly, $pos(t_i, \Sigma) = ci + 1$, and $pos(s_{\pi(i)}, \Sigma) \leq ci + 1$, due to the definition of s_i's. It follows that $s_{\pi(i)} \leq t_i$, for $i = 0, 1, \ldots, k$.

Assign now the set σ_i to the root of $T_{\pi(i)}$, for $\pi(i) = 1, \ldots, k$, and σ_{i_0} to the root v of T, for i_0 such that $\pi(i_0) = 0$; this will restore the heap invariant in all subheaps $T_1, \ldots, T_k$, but possibly not in v.

Exchanging now the set σ_{i_0} stored in v with the set σ_k stored in $v_{\pi(k)}$ (when

$\pi(k) \neq 0$), will restore the invariant in v (since σ_k, containing the first c maximal elements from U, has been stored in $v_{\pi(k)}$), and, at the same time, will violate this invariant in the root of $T_{\pi(k)}$, unless $\pi(k) = 0$. Here the invariant must be further restored recursively.

The correctness of the previous algorithm should be clear from previous considerations.

A nonrecursive version of the above algorithm is described by the program from Fig. 2, that builds in the file F a perfect k–ary block heap with the root at $F[i]$, providing that in $F[k(i-1)+j+1]$, for $j = 1, \ldots, k$ are the roots of subheaps at next lower level. A complete perfect k–ary block heap, consisting of q logical blocks per r physical blocks

```
procedure restore(i,k);
var
    U: array [1..(k+1)c] of key;
    π: permutation of {0..k};
    S: array [0..k] of logical_block;
    s: array [0..k] of integer;
    j: integer;
begin
L: do k(i − 1) + 1 < q → comment F[i] is not a leaf;
        1. for j := 1 to k do S[j] := read(F[k(i − 1) + j + 1]) od;
        2. S[0] := read(F[i]);
        3. for j := 0 to k do s[j] := min{S[j]} od;
        4. compute_permutation π : s_π(0) ≤ . . . ≤ s_π(k);
        5. for j := 0 to k do U[cj + 1..c(j + 1)] := S[j] od;
        6. merge_sort({U}) comment a (k+1)-way merge sort should be used here;
        7. for j := 0 to k do S[π(j)] := U[cj + 1..c(j + 1)] od;
        8. exchange(S[π(k)], S[0]);
        9. for j := 1 to k do F[k(i − 1) + j + 1] := write(S[j]) od;
        10. F[i] := write(S[0]);
        11. if π(k) = 0 → exit L;
            [] π(k) ≠ 0 → i := k(i − 1) + π(k) + 1;
            fi;
    od
end
```

Fig. 2: Restoring heap invariant at $F[i]$

each, in file F is then built by the program from Fig. 3. Note that, prior to heap building, we have to sort the elements within each logical block using some kind of (preferably

```
procedure buildheap(k,q);
var
    i: integer;
    S: array [1..c] of key;
begin
    for i := 1 to q do
        begin
            S := read(F[i]);
            sort{S};
            F[i] := write(S);
        end;
    i := (q − 1)/k;
    do i ≥ 1 → restore(i, k); i := i − 1 od
end
```

Fig. 3: Building a perfect k–ary block heap

in–situ) internal sort.

Also note that in both programs from Fig. 2 and Fig. 3, neither any optimization of I/O operations, nor the internal memory utilization by the underlying algorithm has been taken into consideration — rather, as mentioned already before, so far we have concentrated to the clear presentation of the main idea of heap building algorithm. An efficient implementation of the previous algorithm will be sketched in the next section.

4 Efficient Implementation of the Heap Creation Algorithm

The algorithm described in Fig. 2 and 3, respectively, is designed at the level of abstraction that is not very realistic from implementation point of view: namely, it is implicitly assumed there that complete logical blocks can be read from the file in one read operation (though one may suppose that in practice the size of logical blocks will be noticeably greater than that of physical ones), and that many logical blocks (exactly $k + 1$) can be held in internal memory at the same time. Therefore it is necessary to "tune" this algorithm in a way that would take into account the real constraints imposed by the machine environment we are dealing with.

This means, especially, that the file must be read in pieces consisting of physical, rather than logical, blocks, and that internal memory space must be used in an economical way, to enable $(k + 1)$–way merge of elements from different logical blocks (we shall later see that, in the given internal memory space, building heaps with k as large as possible is to be preferred, from time efficiency reasons).

These requirements lead to the following changes in the design of heap building algorithm (see program from Fig. 2 and the respective description in Section 3). In describing the following algorithm we assume that the contents of each logical block in F is already sorted.

First, logical blocks are read into the internal memory in pieces, consisting from individual physical blocks, as it is to be expected. For that purpose we define $k+1$ *input buffers*, and one *output buffer*, of size b (records) in the internal memory.

Moreover, two special data structures, are initialized here.

The first one – called *Space_Directory* — is of type **queue** [1..k+1] **of** *addr*, where *addr* : [1..m]; here the information about currently empty physical blocks will be stored.

In the second one — called *Rearrangement_Directory* — the information needed for final rearrangement of physical blocks, in order to restore the heap invariant in the respective logical blocks, is stored. This data structure is of type **array** [1..(k+1)r] **of** *addr*.

Then, to each input buffer the first physical block from the respective logical block is read in. Doing so, the respective physical blocks can be considered as being empty, and therefore their addresses (within F) are all entered into *Space_Directory*. These actions correspond to line 1 and 2 from program in Fig. 2.

Then the minimal elements from each buffer are extracted — this will give us the elements s_i, for $0 \leq i \leq k$, thanks to the fact that elements within each logical block are sorted in nondecreasing order (corresponds to line 3 in the respective program).

Subsequently also the permutation π can be computed and remembered (line 4).

Sorting of U (lines 5 and 6) proceeds now in the following way: the idea is to merge the respective runs, prepared in input buffers, with the help of a $(k+1)$–way merge, using the technique of multiway merging with replacement selection (see the details of this technique e.g. in [6]).

The size of the respective selection tree is $\theta(k)$.

The individual elements from the respective buffers enter this tree at its leaves, and as they proceed towards the root — smaller ones "winning" over larger ones — their places are taken by new incoming elements from the respective input buffers. Once some buffer gets empty, the next physical block from the respective logical block is read in, and the address of this particular physical block is entered into the *Space_Directory*. Once some logical block gets empty (i.e., all its physical blocks have been exhausted in the previous merging process), 'infinitely large' elements — denoted as $+\infty$ — enter into the respective input buffers, until the first of them appears at the root of selection tree; this ends the merging process.

The elements of U in sorted nondecreasing order, as produced by the selection tree, are moved continuously into the output buffer. Once it gets full, its contents is temporarily written into the physical block whose address *addr* is learned (and removed) from *Space_Directory*, and the merging process can continue.

The physical blocks, as produced in sorted order by selection process, are continuously numbered, and the respective address *addr*, where the particular — say i–th — block in the final block ordering, gets stored, is remembered in *Rearrangement_Directory* at position i.

After finishing of merging process the physical blocks, stored temporarily in the blocks allocated during the process of merging by *Space_Directory*, are rearranged into the proper logical blocks by making use of the information stored in *Rearrangement_Directory* (this information enables to learn where the i–th physical block in the final block ordering is to be found, in F), and of the permutation π (this information says, into which logical block the i–th physical block should be stored, in order to recreate the heap invariant). All the above action correspond to line 7, 8, 9, and 10, respectively, in the program from Fig. 2.

Finally, the process proceeds recursively at the next lower level.

The respective program in a more formal notation will be presented in the full version of the paper.

5 Complexity of Multiple Heap Creation

From the description of the heap creation algorithm in section 3, and from program in Fig. 2, it is readily seen that building a k–ary heap of height h, with load factor c requires the construction of, recursively, k similar heaps of height $h-1$, and their subsequent joining by a new logical block of size c, using the procedure *restore*.

Even from the sketched "refined" description of the heap building procedure in section 4, one may infer that the realization of *restore* at one level in the heap requires $O(r(k+1))$ I/Os (more exactly: reading and writing $k+1$ times r physical blocks twice), which for all subsequent iterations at lower levels gives a total of $O(hr(k+1))$ I/Os.

Thus, denoting by $H(k,h)$ the number of I/Os required to build a perfect k–ary block heap of height h, from previous considerations we get the following recurrence inequality:

$$H(k,0) = 0$$
$$H(k,h) \le kH(k,h-1) + O(rh(k+1)), \quad \text{for } h > 0$$

for any $k \ge 2$, with the solution

$$H(k,h) = O(r(k+1)\sum_{i=0}^{h}(h-i)k^i = O\Big(\frac{r(k+1)}{(k-1)^2}[k^{h+1} - k(h+1) + h]\Big)$$

Since our perfect heap has $q = (k^{h+1}-1)/(k-1)$ nodes, the previous estimation transforms to

$$H(k,h) = O\Big(\frac{k+1}{k-1}rq\Big) = O(m)$$

(recall that m denotes the file size in terms of physical blocks — see the relations among r, q, and m at the close of section 2).

This means that a k–ary block heap can be constructed, using linear number of I/Os, that does not depend on the multiplicity of the constructed heap.

The memory requirements follow from the space requirements of data structures held in the internal memory, as described in section 4.

Theorem 1 *On a RAM with internal memory capacity of $O(kbw + kr)$ words, a k–ary block heap with load factor $c = rb$, stored on a disk as external random access file, consisting of m physical blocks of b w-word records each, can be constructed, using $O(m)$ I/Os, for any $k \geq 2$.*

6 Sorting by Heapsort

Having created a multiple block heap, one can sort similarly as in the standard heapsort: the maximal elements from the root are exchanged with elements from the leaf, located at the "right end" of the file; then the file is reheaped in a way similar to procedure *restore* from Fig. 2, etc., until eventually the entire file is sorted.

While this process can certainly be performed *in situ*, requiring internal memory of capacity $O(kbw + kr)$ words, unfortunately, from a time complexity point of view, it appears not to be efficient enough.

Namely, to output a single logical block in the final ordering, $k+1$ logical blocks must be processed, which requires $O(r(k+1))$ I/Os. Reheaping multiplies the above cost by the height of the heap — i.e., roughly by $O(\log_k q)$, and this must be repeated for each logical block — i.e., q times.

In a total, $O(krq \log_k q) = O(km \log m / \log k)$ I/Os are needed, which is by factor of $\Theta(k)$ worse then the lower bound from section 1, for $k = \Theta(s/(bw))$ (i.e., essentially for internal memory completely filled by input buffers, which maximizes the order of merge).

Nevertheless, for $k = 2$, i.e., for very small internal memories, the previous algorithm is optimal. In this case also, the heap creation phase can be simplified substantially.

Incidentally, this simplified version of the present algorithm presents the first external sorting algorithm, based on Heapsort, that appeared in 1983 [10] as a technical report written by the first author. Its successful implementation on PDP computers was reported later in [11], and became a subject of a patent. Therefore both reports have never been openly published.

7 Sorting the Block Heap by Tree–Selection

The idea of a sorting algorithm that works over multiple block heaps, that would asymptotically save disk accesses over the previous one, is based on the technique of tree–

selection sorting.

We shall explain the basic idea first for the "almost *in–situ*" case — viz., for the case when there is an *output "write–only" file* at our disposal, where the sorted elements from the original file should appear.

In this case, we first remove the elements from the root and place them to output file. Then, among the elements in roots of k subheaps at the next lower level a tournament is organized — i.e., using a k–way merge (performed with the help of a selection tree, located in the internal memory, similarly as in the heap creation algorithm from section 4) the elements from these roots are promoted to their father (root of the original heap). When the root is full, its elements are output to output file.

When some subroot becomes empty during this process, a similar process to fill this particular root should be started at the next lower level. Nevertheless, prior to starting such a process the internal memory must be made clear from elements that have already been input into internal memory, but so far have not been output out of it.

To clear the memory, we collect all elements residing there, and internally sort them in nondecreasing order. Then, we redistribute the segments of length b of the resulting sorted sequence back into input buffers in such a way that no smaller elements than those that have found themselves in the respective buffers prior to the current "clean up" phase, are stored now into buffers. This is done in much the same way as we did in the case of heap creation algorithm from Fig. 2. Doing so the buffer corresponding to the empty subroot will remain empty, since before getting completely empty it must have contained the elements that have been output among the last elements for sure.

Then the contents of non-empty buffers are written back to corresponding subroots. Note that the respective subroots need not be completely filled after this operation, nevertheless heap invariant will be restored here again. The information, where (at what position) in a given logical block (heap node) "empty" physical blocks (that emerge during clean–up phase) start is remembered in a special data structure called *Shadow_Heap*, that is implemented as an **array** [1..q] **of** [1..r]. The respective information is used later on when the algorithm returns to some node that has not yet been completely emptied, for a quick localisation of non–empty physical blocks within the node at hand. Also note that the subroot that became originally empty remains empty also after the clean–up phase, thank to the before mentioned fact that the corresponding (input) buffer remained empty.

Then the internal memory is free and again the "promotion process" can start at the particular empty root at hand: the vacant places there are filled up by merging process (performed in the internal memory) by elements from the sons of the respective root.

In this way sorting proceeds in promoting the elements through vacant places up to the root, where elements are taken off and transferred to output file.

The sorting process ends by exhausting all elements.

When elements are read from and written to logical blocks in chunks consisting of single physical blocks, similarly as described in section 4 in the efficient implementation of heap creation algorithm, then the I/O complexity, and internal memory utilization, of the above mentioned sorting process is quite favorable.

Namely, to promote elements from r physical blocks (that will fill one logical block) $\Theta(r)$ physical blocks from lower level nodes must be read into the internal memory in the merging phase, and about the same number of blocks must be written back in the clean–up phase. This is to be repeated for each level, and for each logical block.

Hence, in a total we get that $O(qr \log_k q) = O(m \log q / \log k)$ I/Os are necessary to realize the algorithm sketched above. Comparing this figure with the optimal estimate from section 1 we see that for $k = \Theta(s/b)$, i.e., for internal memory size $s = \Theta(kb)$ records, the resulting algorithm is I/O optimal, with the hidden constant in Θ–notation reasonably small (between 1 and 2).

Now it remains to show that the previous algorithm can be realized also without the help of write–only output file.

A simple, although somewhat brute–force solution, is the following: we make use of vacant places in logical blocks that remain "free" after the respective physical blocks have been emptied by promoting the corresponding elements to high levels.

To implement this idea we need a mechanism that keeps track of free physical blocks in the heap, and fills them up with blocks being output by the sorting process.

However, since the blocks get free in general in a non–regular manner, while, fortunately, the blocks are being output in a sorted order, using an other mechanism it is necessary to keep track which block have been placed where.

Both mechanisms can be realized with the help of data structures we have already seen in the efficient implementation of heap creation algorithm from section 4 — namely with the help of *Space_Directory*, and *Rearrangement_Directory*.

The drawback of this method is that in order to preserve the "*in–situ* property" of the underlying algorithm both data structures must be kept in the internal memory. While it is enough to have *Space_Directory* of size $\Theta(k)$ words, (since it gets empty after each clean–up phase), the size of *Rearrangement_Directory* will be as large as m words (or, if we have to economize space, $\Theta(m \log m)$ bits), since during the sorting process the information about each physical block (more precisely: its address $addr$) will be entered into this directory. Nevertheless, with the sizes of contemporary internal memories also this requirement can be fulfilled.

The information stored in *Rearrangement_Directory* is then used in the final rearrangement of physical blocks, according to the resulting ordering determined by the sorting process. Such rearrangement is possible to do using linear number of I/Os without any auxiliary storage by following the cycle structure of the underlying permutation (see e.g. [5])

Internal memory requirements of the algorithm at hand are as follows: we need about m words for *Rearrangement_Directory*, about q words for *Shadow_Heap*, $\Theta(k)$ words for *Space_Directory* and other auxiliary variables and data structures, and $(k+1)bw$ words for k input buffers and one output buffer. Morover, to be able to sort each logical block in internal memory (at the beginning of sorting — see program from Fig. 3), c must not exceed the entire internal memory size s (measured in terms of records).

What remains to estimate is the total asymptotical number of comparisons performed during the sorting process. This is relatively easy. First, preparing q initial runs of lengt c in all logical blocks requires $\Theta(qc \log c)) = O(n \log n)$ comparisons. Then, to each I/O operation, dealing with a physical block consisting of b records, in average $\Theta(b \log k)$ internal operations are performed, since k–ary merging contributes to each element output by $\Theta(\log k)$ comparisons. Hence, from the previous estimate for I/O complexity of our algorithm after some algebraic manipulation we determine that $\Theta(n \log n)$ comparisons altogether are performed.

Similarly as before we are not going to give a more formal description of the above tree–sorting process, since the resulting algorithm tends to be quite technical; nevertheless, it is included in the full version of the paper.

The characteristics of the resulting sorting algorithm are summarized in the following theorem:

Theorem 2 *Let a random access file F consist of n records of length w words each, with $w \geq 1$; let F be further partitioned into q logical blocks per c records each (i.e., $n = qc$), let each logical block consist of r physical blocks, let each physical block consist of b records (i.e., $c = rb, n = mb$), for some $1 \leq q \leq m$, $c \geq 1$, $r \geq 1$, and $b \geq 1$. Then F can be sorted* in–situ *on a single processor RAM with internal memory capacity* $\max\{cw, m+q+(k+1)bw\}+O(k)$ *words, using $O(m \log m / \log k)$ input/output operations, and $O(n \log n)$ key comparisons, for any $k \geq 2$.*

By comparing the bound from the previous theorem with lower bound estimation from section 1, we see that when the total internal memory capacity s is large enough to accommodate all the necessary data structures to perform the above sort, and when we select k such that $s = \Theta(kbw)$, and when simultaneously $s = \Theta(m)$ and $s = \Theta(cw)$ (compare these figures with the original block heap design goals, mentioned at the beginning of section 2), then asymptotically optimal sorting w.r.t. to I/O complexity will be achieved.

It is an interesting open question whether there is a variant of our (or any other) algorithm that would sort optimally files consisting of m physical blocks *in–situ*, without having the need to store auxiliary information of size $\Theta(m)$ words in the internal memory.

8 Parallelizing Sorting Algorithms

When it comes to the design of parallel algorithms, the underlying parallel machine model must be specified, since its properties determine the class of algorithms that can be run efficiently on the model at hand. In what follows we shall consider a realistic model of a *"multiRAM"*, consisting of p asynchronously working RAMs (processors) $P_1, P_2, \ldots, P_p$, each of them equipped with its own internal memory and with its own (disk) external memory, and with a processor interconnection topology presented by a simple line (i.e., each processor, except the first, and the last one, can communicate with its immediate predecessor or successor, respectively).

It appears that for such a multiRAM, a general construction, that speeds up the sorting process and works for any sorting algorithm, can be designed.

Let F, as before, be the file to be sorted. Split the file evenly among p "units" of our multiRAM in such a way that the external memory of P_i will hold the i–th segment F_i of F, consisting of $\lceil m/p \rceil$ physical blocks, for $i = 1, 2, \ldots, p$.

Then sort, in parallel, the respective segments of F at each processor, by your favorite sorting algorithm into a nondecreasing sorting order, say.

What remains to be done, is to merge the resulting sorted segments into one sorted file, split evenly among all processors.

This can be achieved by a variant of an *odd–even merge sort* that in a cycle, in parallel, alternately, first for pairs of processors of form (P_i, P_{i+1}), $1 \leq i \leq p-1$, starting with odd numbers, and then for pairs starting with even numbers, repeats altogether $p - 1$ times the following steps:

- P_i (P_{i+1}) reads F_i (F_{i+1}) from the left (right) end, block by block, record by record, and compares the keys of records residing "symmetrically" at the opposite ends of the respective files (i.e., we compare the j–th record key from F_i with the $(\lceil m/p \rceil - j + 1)$–th record key from F_{i+1}), until the first pair (x, y) of keys with $x \geq y$, for some $x \in F_i$, $y \in F_{i+1}$, at known positions, is found;

- at the positions found, split both files into two files called L_i, R_i, and L_{i+1}, R_{i+1}, respectively (see Fig. 4); clearly, $v \leq z$ for all $v \in \{L_i\} \cup \{L_{i+1}\} = L$, and $z \in \{R_i\} \cup \{R_{i+1}\} = R$, and $|L| = |R|$;

- move L_{i+1} to P_i, and R_i to P_{i+1};

- in P_i, perform a two-way merge of L_i with L_{i+1}, temporarily store the outcomming sorted sequence in physical blocks whose contents reside at that very moment in the internal memory, and remember the respective addresses in a *Rearrangement_Directory* (that it used in much the same way as in section 7); in parallel, do the analogous actions in P_{i+1};

- rearrange the physical blocks both in P_i and P_{i+1}, in parallel, according to resulting ordering remembered in the respective *Rearrangement_Directories*;

- iterate.

From the analogy with the odd–even transposition sort it holds that after $p-1$ iterations the file will be sorted.

Obviously, one parallel step of the above algorithm can be performed in linear time w.r.t. the file length processed by each unit, i.e., using $\Theta(m/p)$ I/Os. Therefore the entire distributed file sort can be finished after $\Theta(m)$ I/Os, providing that the I/O

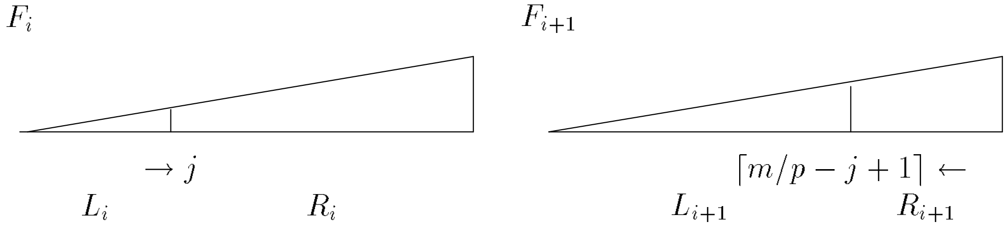

Fig. 4: A scheme of parallel external odd–even merge in P_i and P_{i+1}

operations are synchronized (or that all occur within the same reasonably short time interval), and clearly the underlying algorithm is an *in–situ* algorithm.

Theorem 3 *On a multiRAM, consisting of p processors with internal memory capacity* $\Theta(m/p+bw)$ *words, a parallel merge of p sorted random access files, consisting of* $\lceil m/p \rceil$ *physical blocks of length b w–word records each, can be performed* in–situ *using* $\Theta(m)$ *parallel I/Os.*

Applying the previous construction to our disk sorting algorithm from section 7, it is clear that in this way a roughly p–fold speed–up could be achieved, for $1 \leq p \leq O(\log m/\log k)$ (we cannot go below $O(m)$ boundary, because of the complexity of the final merging process). Clearly, this is about the best we can expect by using p processors.

9 Conclusion

A new asymptotically optimal disk sorting algorithm, based on the ideas of internal heapsorting, has been proposed. Its main advantage over other known disk sorting algorithms of comparable efficiency, lies in the fact that it is an *in–situ* algorithm, i.e., one that sorts an external file without needing any other external auxiliary files.

Further, a fairly general method for parallelization of sequential disk sorting algorithms has been proposed. It is based on the idea of external parallel odd–even merging, and on realistic models of parallel machines, achieves, within a certain reasonable range, a linear speed–up proportional to the number of processors.

By the design of the previous sorting algorithm new avenues for the research of external sorting algorithms that are optimal both from the time, as well as from the space complexity point of view, have been opened. The new algorithm presents but the first approach to the solution of the above problem.

The question whether there exist more practical algorithms of the above kind (with a smaller constant hidden in the "big–O" notation), of both sequential and parallel nature, and whether similar algorithms can be designed also for the more general case of multilevel memory hierarchies, remains open for further research.

References

[1] Aggarwal, A. — Alpern, B. — Chandra, A. — Snir, M.: *A Model for Hierarchical Memory*, in Proceedings of the 19–th Annual ACM Symposium on Theory of Computing, May 1987, pp. 305–314

[2] Alpern, B. — Carter, L. — Feig, E.: *Uniform Memory Hierarchies*, in Proceedings of the 31–st Annual IEEE Symposium on Foundations of Computer Science, October 1990, pp. 600–608

[3] Aggarval, A. — Vitter, J.S.: *The Input/Output Complexity of Sorting and related Problems*, Communications of ACM, Vol. 31, September 1988, pp. 1116–1127

[4] Dufrene, J., W.R. — Lin, F.C.: *An Efficient External Sort Algorithm with no Additional Space*, The Computer Journal, Vol. 35, No. 3, 1992, pp. 308–310

[5] Knuth, D.E.: *The Art of Computer Programming, Vol. 1, Fundamental Algorithms*, Addison–Wesley Reading, Mass., 1968

[6] Knuth, D. E.: *The Art of Computer Programming, Vol. 3, Sorting and Searching*, Addison–Wesley Reading, Mass., 1973

[7] Nodine, M.H. — Vitter, J.S.: *Deterministic Distribution Sort in Shared and Distributed Memory Multiprocessors*, in Proceedings of 5–th Annual ACM Symposium on Parallel Algorithms and Architectures, Velen, Germany, June 1993, pp. 120-129

[8] Six, H.W. — Wegner, L.: *Sorting a Random Access File in–Situ*, The Computer Journal, Vol. 27, No. 3, 1984, pp. 270–275

[9] Vitter, J.S. — Shriver, E.A.M.: *Algorithms for Parallel Two–Level Memories*, Algorithmica, to appear

[10] Wiedermann, J.: *Heapsorting Direct Access Files*, Research Report No. 1, Computing Research Center, Bratislava, 1983 (in Slovak)

[11] Wiedermann, J. — Lamačka, P.: *External Sorting without Working Files*, Technical Report No. 1, Computing research Center, Bratislava, 1984 (in Slovak)