Alcom-FT Technical Report Series
ALCOMFT-TR-01-171

I/O-Space Trade-Offs

Lars Arge*[1] and Jakob Pagter**[2]

[1] Department of Computer Science, Duke University, Durham, NC 27708, USA
[2] BRICS***, University of Aarhus, DK-8000 Aarhus C, Denmark

Abstract. We define external memory (or I/O) models which capture space complexity and develop a general technique for deriving I/O-space trade-offs in these models from internal memory model time-space trade-offs. Using this technique we show strong I/O-space product lower bounds for Sorting and Element Distinctness. We also develop new space efficient external memory Sorting algorithms.

1 Introduction

In internal memory models the time and space complexity, as well as trade-offs between the two, are well studied for fundamental problems such as Sorting and Element Distinctness. For example, Pagter and Rauhe [20] recently proved an $O(N^2)$ upper bound on the time-space product for Sorting N elements, matching a lower bound of Beame [9]. Their time optimal algorithm can be used to improve space usage of internal memory Sorting by a factor of $\Theta(\log^2 N)$ compared to classical algorithms like MergeSort and HeapSort. Such an improvement would be of considerable practical interest when dealing with massive data sets residing on external storage devices such as disks—it would maybe even make the difference between being able to solve a problem or not. For example, if dealing with 50GB of data even a factor of $\log N$ would amount to a space reduction of a factor of more than 30. Unfortunately, very little is known about space complexity in external memory models where the main complexity measure is the number of I/Os needed to solve a problem. For example, even though several I/O-optimal external Sorting algorithms have been developed, no algorithm using sub-linear disk space—not counting the (read only) space holding the input—is known. One reason for this is that no external memory model capturing sub-linear space complexity has been defined.

In this paper we define external memory models which capture space complexity and use them to study I/O-space trade-offs for fundamental problems such as Sorting and Element Distinctness.

* Supported in part by National Science Foundation ESS grant EIA–9870734, RI grant EIA-997287, and CAREER grant EIA–9984099. E-mail: `large@cs.duke.edu`.

** Partially supported by the IST Programme of the EU under contract number IST-1999-14186 (ALCOM-FT). Parts of this work while the author was visiting Duke University and while visiting University of Toronto. E-mail: `pagter@brics.dk`

*** **B**asic **R**esearch **i**n **C**omputer **S**cience. Centre of the Danish National Research Foundation.

1.1 Related Work

The study of internal memory time-space trade-offs attempts to give formulae that relate time T and space S for a given problem. A typical result is of the form $T \cdot S = O(f(N))$ for some problem and function f. Time-space trade-offs for SORTING are well studied [9, 10, 12, 13, 19, 20], and for time above (roughly) $N \log_2 N$ the exact complexity has been established to be $T \cdot S = \Theta(N^2)$ [9, 20]. This means that if $S = \Omega(N/\log_2 N)$ then we can sort in time $O(N \log_2 N)$, and this is the best possible. Similarly, if $T = O(N^{3/2})$ then $S = \Theta(N^{1/2})$ is possible and required. Time-space trade-offs for ELEMENT DISTINCTNESS has also been studied extensively [3, 11, 15].

The standard model for studying external memory algorithms (or I/O-algorithms) is the I/O-model of Aggarwal and Vitter [2]. In this model the internal memory of size M is divided into $m = M/B$ blocks of size B each. The external memory is also divided into blocks of size B, and initially the N input data elements reside in the first $n = N/B$ blocks of external memory. An I/O is the movement of one block of elements to or from external memory, and the goal is to design algorithms that use as few I/Os as possible under the constraint that computation can only be performed on elements in internal memory.

During the last decade, a large number of I/O-efficient algorithms have been developed in the I/O-model—see e.g. recent surveys [5, 23]. For example, it is well known that (under some restrictions) SORTING requires $\Omega(n \log_m n)$ I/Os and several $O(n \log_m n)$ algorithms using $O(n)$ extra space (disk blocks) have been developed. However, no results are known about I/O-space trade-offs for SORTING. Similarly for ELEMENT DISTINCTNESS, Arge et al. [6] showed that the problem is as hard as SORTING in a comparison based model, and Arge and Miltersen [7] gave an $O(n)$ I/O (and space) randomized algorithm for the problem, but nothing is known with respect to I/O-space trade-offs.

It should be mentioned that one reason no I/O-space trade-offs are known in the I/O-model is that it allows for the input to be overwritten. Thus $\Omega(n)$ space is always available for algorithms in the model and one cannot formally express sub-linear space bounds. External memory models that capture space complexity have been introduced in the area of straight-line computation models (i.e., models where branching or "if-then-else" statements are not allowed). An example is the so-called red-blue pebble games—see e.g. [14, 21]. However, disallowing branching is too severe a restriction for our purposes. No non-straight-line external memory model capturing sub-linear space has been defined.

1.2 Our Results

In Section 2 we introduce computational models which allow us to study I/O-space trade-offs. We first introduce an extension of the Aggarwal and Vitter model (or rather an I/O-version of the RAM-model equivalent to their model) in which sub-linear space complexity can be expressed. This is the model we will use when developing space-efficient I/O-algorithms. We next extend the branching program model [10, 12, 21]—the model most commonly used for showing internal memory time-space trade-offs—in order to capture I/O and space complexity simultaneously. We also prove that lower bounds in this model are valid in the extended I/O-model.

In Section 3 we develop a technique for obtaining I/O-space trade-offs for external memory computation from time-space trade-offs for internal memory computation. More precisely, let T and S denote the time and space usage of an algorithm solving a problem P over $U = \{1, \ldots, R\}$. Similarly, let T^{IO} denote the number of I/Os and S^{IO} the space usage of an external memory algorithm solving P. We prove that if $T = \Omega(f(N, S))$, then $T^{IO} = \Omega(\frac{f(N, S^{IO})}{B})$.

Using the general result we prove I/O-space trade-offs for SORTING and ELEMENT DISTINCTNESS. Combining the result with a result of Beame [9], we for example show that for SORTING the I/O-space product is $\Omega(N^2/B) = \Omega(N \cdot n)$. Using the internal memory algorithm of Pagter and Rauhe [20] in external memory shows that this bound is tight among algorithms using more than (roughly) $N \log_2 N$ I/Os. This is an interesting result, as it suggests that when disk space is restricted, traditional internal memory approaches can lead to optimal external memory algorithms. Using internal memory results of Ajtai [3] and Yao [24] we also provide lower bounds for ELEMENT DISTINCTNESS.

Finally in Section 4 we discuss an external memory generalization of the algorithm of Pagter and Rauhe which for certain choices of M and B obtains optimal I/O-space trade-off for SORTING down to the optimal number of I/Os $O(n \log_m n)$. In general however, we can only prove an $O((N^2/(B+m)) \log m)$ upper bound on the I/O-space product. We conjecture that our lower bound for SORTING is tight for all values of M and B, that is, that an algorithm achieving $O(N^2/B)$ exists.

2 Models of Computation

In this section we introduce computational models which allows us to study I/O-space trade-offs. In Section 2.1 we first consider upper bound models and in Section 2.2 we then consider lower bound models. We discuss the relationship between the models in Section 2.3.

One main difference between our models and the standard I/O-model is that we assume input to be read-only. A natural question is whether this is reasonable and we claim that it is. Consider for example the task of SORTING a huge database by a secondary key: In such an example it might be important not to overwrite the original database sorted by primary key. A typical example is the customer database of a bank, which will normally be sorted by account numbers. Occasionally the bank might want a phone book over its customers, requiring the database to be sorted on customer names, but rarely will it be interested in erasing the original database used for all standard business transactions. Other examples occur when the input is stored on a medium which is physically read-only, for example on a CD-ROM.

2.1 Upper Bound Models

In this section we define two external memory extensions of the unit cost RAM model. The unit cost RAM model is a popular internal memory model for showing upper bounds. In this model input consists of N words $x_1, \ldots, x_N$ from some universe $U = \{0,1\}^w$, and it is a normal convention that each word in memory can hold exactly w bits, corresponding to one input element. We use this convention and define a parameter $R = 2^w$ (i.e. each input element can represent one of R values). We will always assume that $R \geq N$.

External RAM Models.

Definition 1 (External RAM) *The external RAM consist of two parts: A memory layout, and an instruction set:*

Memory layout: *Input is located on a read-only input medium consisting of N words of $\log R$ bits, grouped into $n = N/B$ consecutive blocks of B words. Output is to be written to a separate write-only output medium. Furthermore, we have an external memory consisting of blocks of B words of $\log R$ bits each, and an internal memory consisting of M/B blocks of B words of $\log R$ bits each.*

Instructions: *The algorithm can execute the following instructions:*

1. *Read a block from the input into internal memory*
2. *Write a block from internal memory to the output*
3. *Swap a block from internal memory with one from external memory (both may be "empty")*
4. *Perform some unit-cost operation on two words in internal memory, writing the result to one word in internal memory.*

□

The number of I/Os T_R^{IO} performed by an external RAM is the number of times the algorithm executes instructions 1, 2 or 3, and we define space S_R^{IO} to be the number of bits occupied by the external memory. Note that we ignore the $M \log R$ bits available in internal memory since we will always assume that $S_R^{IO} \geq M \log_2 R$. Also note that we define space in terms of bits and not words or blocks. One can of course easily translate our space measure into measures based on words ($S_R^{IO}/\log R$) or blocks ($S_R^{IO}/(B \log R)$). In the introduction the space bounds were expressed in terms of blocks.

It should be clear that the external RAM model is essentially a straightforward extension of the I/O-model of Aggarwal and Vitter—basically the input has just been made read-only.

Definition 2 (Comparison external RAM) *A comparison external RAM is an external RAM, with instruction 4 replaced by*

- *4a Compare input elements, or copies thereof, in two words in internal memory using a binary comparison*
- *4b Perform some unit-cost operation on two words in internal memory which are not occupied by input elements, or copies thereof, writing the result to one word in internal memory*

□

The number of I/Os T_C^{IO} performed by a comparison external RAM is the number of times the algorithm executes instructions 1, 2 or 3, and the space use S_C^{IO} is the number of bits occupied by the external memory.

Note that in a comparison external RAM one can *only* access words containing input elements, or copies thereof, through comparisons. All other words in internal memory can be manipulated freely. We think of each word containing (a copy of) an input element as being marked, and such marked words can only be accessed via comparisons with another marked word. This somewhat strange way of enforcing a comparison model is a result of the fact that in an external memory model it is vital that we are allowed to make and move copies of elements (since block movement is the main complexity measure). Another standard way of enforcing a comparison model is to ensure that elements are indivisible and that new elements cannot be produced, that is, that each word in memory is either empty or contains an input element (the so-called *indivisibility assumption* [2, 7]). However, there is evidence that the indivisibility assumption drastically increases the complexity of certain problems [1, 7], and furthermore, we would like to be able to manipulate objects such as pointers in our algorithms.

Properties of External RAM Models.

It should be clear that the external RAM models can easily simulate algorithms constructed for models where one is allowed to overwrite the input: Simply make a copy of the input in the external memory and run the algorithm on this copy. In particular, the I/O-optimal $O(n \log_m n)$ SORTING algorithms of Aggarwal and Vitter [2] may be implemented on the comparison external RAM. Note also that the external RAM is at least as strong as the comparison external RAM.

2.2 Lower Bound Models

The models we will use for showing lower bounds (on I/O-space products) are extensions of the branching program model. The branching program model is a well established internal memory model for showing lower bounds on time-space complexity, see e.g. [3, 9, 10, 12, 17, 24]. Branching programs come in two main variants: Comparison based branching programs, which were initially studied in [22] according to which they were introduced by Pippenger, and R-way branching programs introduced by Borodin and Cook [10]. Detailed discussions of branching programs can be found in [12, 21].

A *comparison branching program* is a directed acyclic graph (DAG) with one root. Each non-leaf node is labeled $(i : j)$ and has two outgoing arcs labeled $x_i < x_j$ and $x_i \geq x_j$. A computation starts at the root and proceeds in the natural way until a leaf is reached. If one is studying decision problems, each leaf will be labeled 0 or 1 depending on whether the corresponding computation rejects or accepts the input. For functions such as SORTING, each arc may be further labeled with elements from the output domain. The output of the computation is the ordered concatenation of the outputs encountered along the computation path.

Time T_C in the comparison branching program model is defined as the height of the branching program, corresponding to the number of comparisons performed in the worst case, and space S_C is defined as $\log|V|$, where V is the set of vertices of the branching program. This is an adequate space measure since it gives a lower bound on the number of bits required to distinguish between the states of the program. We will discuss this in greater detail when comparing RAM models and branching programs in Section 2.3.

An *R-way branching program* is a DAG with one root, where each non-leaf (branching) node is labeled with an index i ($1 \leq i \leq N$) and has an outgoing arcs for each element of $U = \{0,1\}^w$ (recall that $R = 2^w$). A computation proceeds as before except that in a node labeled i the arc labeled l is followed if $x_i = l$; i.e., a branch is made according to the value of x_i.

As previously, T_R is defined as the height of the DAG, corresponding to the number of times we read one of the elements from the input in the worst case. Space S_R is defined as for comparison branching programs. As we can simulate a comparison using two R-way branches, R-way branching programs are asymptotically stronger than comparison branching programs (for $R = N^{O(1)}$). From a practical point of view, the R-way branching program model is an unrealistically strong model of computation—given enough space to remember the value of all the input elements ($O(R^N)$ nodes or space $O(N \log_2 R)$), one can decide any problem after reading each element *once*, i.e., in linear time. On the other hand, when restricting the space one can prove interesting and sometimes even tight time-space trade-offs in the model. In the following we define external versions of the branching program models.

External Branching Program Models.

Definition 3 (Comparison external branching program) *A comparison external branching program is a comparison branching program with two types of nodes—comparison nodes and I/O-nodes: An I/O-node replaces any B elements in the internal memory of size M with any B elements from the input. A comparison node can only compare two elements which are both in internal memory.* □

The number of I/Os T_C^{IO} performed by a comparison external branching program is defined as the maximum number of I/O-nodes encountered along any root-leaf path. As previously, space S_C^{IO} is defined as $\log|V|$, where V is the set of vertices of the branching program.

We emphasize the fact that one can read *any* B elements when making a block transfer (I/O) from input to internal memory. This seems like a strong and unrealisticly powerful operation, but not only can we prove interesting—in some cases even tight—lower bounds in this model, we also (as we will see) need this strength in order to simulate comparison external RAM algorithms (Theorem 1). Note also that the term "internal memory" is somewhat misleading, as the M elements are not physically present in any kind of memory. At any point of the computation, the elements "in internal memory" are just the M input elements (out of the N possible elements) the external branching program is allowed to compare. Finally, note that comparison external branching programs *cannot* make copies of elements, since they can only access input elements through comparisons.

Definition 4 (R-way external branching programs) *An R-way external branching program is an R-way branching program with two types of nodes—branching-nodes and I/O-nodes: An I/O-node replaces any B elements in the internal memory of size M with any B elements from the input. A branching-node performs a branch based on the value of one of the M input elements in internal memory.* □

The number of I/Os T_R^{IO} performed by a R-way external memory branching program, as well as the space use S_R^{IO}, is defined as for the comparison external branching program. For both R-way external branching programs and comparison external branching programs we again use the natural assumption that $S^{IO} \geq M \log_2 R$.

We call a (comparison or R-way) external branching program where all nodes have in-degree at most 1 an *external tree*. Arge et al. [6] defined a model similar to a comparison external tree called the *I/O decision-tree*, and Munagala and Ranade [18] defined a model similar to an R-way external tree. In both models one is only allowed to read contiguous input elements. However, unlike external trees, both models contain a mechanism for rearranging (writing) the input elements.

Properties of External Branching Programs.

A key property of (comparison or R-way) external branching programs is that removing the I/O-nodes results in a standard branching program. In this subsection we will discuss two other properties of external branching program.

A branching program is called *leveled* if the nodes of the program can be partitioned into T classes $V_1, V_2, \ldots, V_T$ such that arcs emanating from nodes in V_i go to nodes in V_{i+1}. Pippenger proved that any standard branching programs of height T using space S can be transformed into a leveled branching program with height $T+1$ and using space less than $2S$—see e.g. Borodin et al. [12]. We say that an I/O-node w is the *immediate I/O-successor* of a node v, if w is the first I/O-node encountered on one of the paths from v to a leaf of the branching program. Note that v may have several immediate I/O-successors. We say that an external branching program is *I/O-leveled*, if the I/O-nodes can be partitioned into T^{IO} classes $V_1, V_2, \ldots, V_{T^{IO}}$ such that all I/O-nodes in V_i have their immediate I/O-successors in V_{i+1}. With a simple modification of Pippenger's proof we can show the following.

Lemma 1 *Any (comparison of R-way) external branching program using T^{IO} I/Os and S^{IO} space can be transformed into an I/O-leveled external branching program solving the same problem using $T^{IO}+1$ I/Os and space less than $2S^{IO}$.*

The *immediate I/O-ancestor* of a node v can be defined analogously to immediate I/O-successor. An external branching program is called *I/O–separated* if all internal nodes other than I/O-nodes have only one immediate I/O-ancestor.

Lemma 2 *Any I/O-leveled (comparison or R-way) external branching program using T^{IO} I/Os and S^{IO} space can be transformed into an I/O-separated and I/O-leveled external branching program solving the same problem using T^{IO} I/Os and less than $2S^{IO}$ space.*

Proof: The details of the proof appear in the full version of this paper [8]. $\square$

In the following, we will assume without loss of generality that all external branching programs are I/O-leveled and I/O-separated.

2.3 External RAM's vs. External Branching Programs

In this section we prove two theorems which allow us to prove lower bounds for external RAM's using the external branching program models.

Theorem 1 *For any comparison external RAM algorithm solving a problem P in T_C^{IO} I/Os and space $S_C^{IO} \geq M \log_2 R$ there exists a comparison external branching program solving P in T_C^{IO} I/Os and space at most $2S_C^{IO}$.*

Proof: The basic idea of the proof is to construct a branching program with a node for each state of the RAM algorithm and with arcs reflecting how the computation proceeds. However, there seems to be one major problem with this idea, namely that while the comparison external RAM can make copies of input elements and move them around in memory, a comparison external branching program can only compare input elements. However, as we will show below, this problem can be overcome using the comparison external branching programs (powerful) ability to move *any* B input elements into internal memory instead of just contiguous ones. We also use the fact that comparison external RAM's can only access (copies of) input elements using comparisons.

As the comparison external RAM algorithm uses S_C^{IO} bits in external memory and have $M \log_2 R$ bits in internal memory, is has at most $2^{S_C^{IO}+M\log_2 R} \leq 2^{2S_C^{IO}}$ distinct states. We split these states into three types based on the operation performed in a given state:

1. I/Os (instructions 1, 2, and 3).
2. Comparisons of (copies of) input elements (instruction 4a).
3. Other operations (on non-input elements) (instruction 4b).

After performing an operation in a given state the comparison external RAM algorithm proceed to a new state: In a type 1 state the algorithm performs an I/O and proceeds to *one* unique state. In a type 3 state the algorithm performs some operation and proceeds to *one* unique state. Note that a sequence of type 3 states are "straight-line" in the sense that the computation performed in the sequence only depends on the first state in the sequence. In a type 2 state the algorithm proceeds to *one out of two* unique states, depending on the outcome of the comparison.

We construct a comparison external branching program from the comparison external RAM as follows: For each state in the comparison external RAM we construct a node in the comparison external branching program, and we connect these nodes according to how computation proceeds from state to state. A comparison external branching program cannot contain nodes corresponding to states of type 3, but as these perform straight-line computation we can remove all such nodes/states by coalescing a node/state (or a sequence of them) into their first successor of type 1 or 2. Nodes corresponding to type 2 states can be left unmodified, but nodes corresponding to type 1 states need to be modified since I/O-nodes in a comparison external branching program always read B input elements from input to internal memory, while much more complicated I/Os can be performed in type 1 states. Nodes corresponding to type 1 states which only *read from the input* we just leave unchanged. Nodes corresponding to type 1 states which *write to output* are removed and the arc between its predecessor and successor is labeled with the appropriate output information. Nodes corresponding to type 1 states which *swap a block* in internal memory with a block from external memory are a little more complicated since we can only read input elements in comparison external branching programs. Consider the block of B elements swapped into internal memory in such a node/state. Some of these elements are copies of the original input elements and some contain other information. The latter can be disregarded since the same information is represented in the state itself (recall that a state represent the content of the entire memory). We replace the node with a node which read the relevant less than B elements from the input. In general these input elements will not constitute a block of the input but we utilize that external branching programs can read any B input elements in one swap operation.

It should be clear that the comparison external branching program constructed in this way solves the problem P in T_C^{IO} I/Os. The branching program uses less than $\log_2 2^{2S_C^{IO}} = 2S_C^{IO}$ space. □

We can prove a similar theorem for external RAM algorithms and R-way external branching programs. We omit the proof as it is very similar to the proof of Theorem 1.

Theorem 2 *For any external RAM algorithm solving a problem P in T_R^{IO} I/Os and space $S_R^{IO} \geq M \log_2 R$ there exists a R-way external branching program solving P in T_R^{IO} I/Os and space at most $2S_R^{IO}$.*

3 Lower Bounds

In this section we first present a general method for obtaining I/O-space trade-offs in external branching program models from time-space trade-offs in normal branching program models. Using this method we then obtain trade-offs for SORTING and ELEMENT DISTINCTNESS.

3.1 General Lower Bound Method

In [6] Arge et al. described a general technique for transforming an internal memory decision tree lower bound into an I/O-decision tree I/O lower bound (see also [7, 18]). The main idea in their technique is a method for transforming an I/O-decision tree algorithm into an internal decision tree algorithm, such that the number of comparisons performed by the internal algorithm is bounded by a function of the number of I/Os performed by the external algorithm. In this section we apply the same idea to branching program models. We first consider the comparison model.

Theorem 3 *Suppose that for any comparison branching program solving a problem P in time T_C and space S_C we have $T_C \geq f(N, S_C)$. Then for any comparison external branching program solving P*

$$T_C^{IO} = \Omega\left(\frac{f(N, S_C^{IO})}{B \log_2 M}\right).$$

Proof: As mentioned, the proof is based on the ideas used by Arge et al. [6]; we will describe a general method for transforming a comparison external branching program solving P using T_C^{IO} I/Os and S_C^{IO} space into a comparison branching program solving P. The comparison branching program lower bound then results in a comparison external branching program lower bound.

Consider an I/O-leveled and I/O-separated comparison external branching program, and as previously let the sub-branching program rooted at an I/O-node v consist of all nodes reachable from v without passing through another I/O-node. Recall that as the comparison external branching program is I/O-leveled, each comparison-node is contained in precisely one sub-branching program. Our aim is to replace each sub-branching program with a new sub-branching program that computes the total ordering of all the elements in internal memory. Given this total ordering and the information implicit in the state of the comparison external branching program any question regarding the input answered by the sub-branching program can easily be answered as the sub-branching program only has comparison based access to the input. We first transform the comparison external branching program into another comparison external branching program where we know the total order of the input elements in internal memory in each I/O-node. (Note that there are $M!$ different orderings of the M elements in internal memory). In order to do so we first make $M!$ copies of the original comparison external branching program—in our construction each of these copies will be used to represent a unique total order of the elements in internal memory. Each copy of the comparison external branching program is now transformed into another comparison external branching program I/O-level by I/O-level, top-down (while maintaining the invariant that the internal memory element order is know in I/O-nodes on already processed levels). We transform the sub-branching program rooted in I/O-node v as follows: We replace the sub-branching program with a sub-branching program which first computes the total order of the B input elements loaded into internal memory by v (i.e. it sorts them), and then finds the positions of the B elements among the sorted elements already in internal memory (i.e. it merges the B "new" elements with the $M-B$ "old" elements). As shown in [6], the height of the new sub-branching program can be bounded by $O(B \log_2 B + B \log_2 m) = O(B \log_2 M)$, which in turn means that it contains $O(2^{B \log_2 M})$ nodes. Each leaf l_n of the new sub-branching program corresponds to a total order of the elements in internal memory. Thus it also corresponds to a unique leaf l_o in the original sub-branching program, namely the leaf at which inputs with this particular total ordering would end up. To guarantee that the transformed comparison external branching program solves P, we want to connect l_n to the same I/O-node v_o on the next I/O-level as l_o is connected to. Note however, that several leaves (total orders) in the new sub-branching program could correspond to l_o. In order not to lose information about the order of the elements in internal memory we therefore connect l_n to v_o of the copy of the original program corresponding to the total order of the elements in internal memory. After processing all I/O-nodes on the same level as v, we go on and process the next I/O-level.

It should be clear that after the transformation we have a new comparison external branching program with the same overall structure as the original comparison external branching program, specifically the new branching program has at least as much knowledge about the input as the original branching program; additionally the new branching program knows the total order of the elements present in internal memory. Consequently the new branching program can answer any question that the original branching program answers, hence if the original comparison external branching program solves a problem P then so does the new comparison external branching program.

From the construction it should also be clear that the new comparison external branching program has the same I/O-height T_C^{IO} as the original comparison external branching program. That the space use of the new comparison external branching program is $O(S_C^{IO})$ can be seen as follows: We have $M!$ copies of the original program, each containing no more than $|V| = 2^{S_C^{IO}}$ I/O-nodes. For each I/O-node we have a sub-branching program of size less than $2^{B \log_2 M}$. Thus in total the new comparison external branching program use space $O(\log_2(M! \cdot 2^{S_C^{IO}} \cdot 2^{B \log_2 M}) = O(M \log_2 M + S_C^{IO} + B \log_2 M) = O(M \log_2 R + S_C^{IO}) = O(S_C^{IO})$.

Next we simply remove the I/O-nodes from the transformed comparison external branching program and obtain a (standard) comparison branching program with height $T_C = O(T_C^{IO} \cdot B \log_2 M)$ using space $S_C = O(S_C^{IO})$. The theorem follows since $T_C \geq f(N, S_C)$. □

We can prove a similar theorem R-way external branching programs.

Theorem 4 *Suppose that for any R-way branching program solving a problem P in time T_R and space S_R we have $T_R \geq f(N, S_R)$. Then for any R-way external branching programs solving P*

$$T_R^{IO} = \Omega\left(\frac{f(N, S_R^{IO})}{B}\right).$$

Proof: We use the same proof technique (construction) as in the comparison model (Theorem 3). The $B \log_2 M$ factor in the comparison model construction was a result of this being the height of the sub-branching program used to obtain full information about the elements in internal memory after an I/O.

We will use the power of the R-way model to decrease this height and thus obtain an improved bound. The crux of the new construction is that reading and "remembering" all of the B new elements—which can be done in time B—we know all there is to know.

We omit the details, which appear in the full version of this paper [8]. $\square$

Note the interesting and somewhat counterintuitive fact that by switching to a stronger model (from Theorem 3 to Theorem 4) we obtain a better lower bound.

3.2 Applications of the Lower Bound Method

In the R-way model, Beame [9] proved that $T_R \cdot S_R = \Omega(N^2)$ for the problem of SORTING elements in the universe $U = \{1, \ldots, N^2\}$. Using Theorem 4 we thus obtain the following.

Corollary 1 *Any R-way external memory branching program* SORTING *N numbers from a universe $U = \{1, \ldots, N^2\}$ has $T_R^{IO} \cdot S_R^{IO} = \Omega(\frac{N^2}{B})$.*

Ajtai [3] proved that for ELEMENT DISTINCTNESS over $U = \{1, \ldots, N^2\}$ we have that if $T_R = O(N)$ then $S_R = \Omega(N)$. Using Theorem 4 we obtain the following.

Corollary 2 *Any R-way external memory branching program solving the* ELEMENT DISTINCTNESS *problem on N numbers from the universe $U = \{1, \ldots, N^2\}$ with $T_R^{IO} = O(n)$ uses space $S_R^{IO} = \Omega(N)$.*

Corollary 1 implies (among other things) that we must use $\Omega(N/(\log_m n))$ space in order to sort N elements in the optimal $O(n \log_m n)$ I/Os. This should be compared to the $N \log_2 R$ space use of the sorting algorithm of Aggarwal and Vitter [2]. Corollary 2 means that in order to decide ELEMENT DISTINCTNESS in a linear number of I/Os we must use at least linear extra space.

Yao [24] proved that any comparison branching program deciding ELEMENT DISTINCTNESS on N numbers must have $T_C \cdot S_C = \Omega(N^{2-\epsilon(N)})$, where $\epsilon(N) = 5/\sqrt{\log_2 N}$. Using Theorem 3 we thus obtain the following.

Corollary 3 *Any comparison external branching program deciding* ELEMENT DISTINCTNESS *on N numbers has $T_C^{IO} \cdot S_C^{IO} = \Omega(\frac{N^{2-\epsilon(N)}}{B \log_2 M})$.*

4 Upper Bounds

In this section we briefly discuss our new upper bounds on the I/O-space product for SORTING. Details appear in the full version of this paper [8].

Modifying the internal memory SORTING algorithm of Pagter and Rauhe [20] in a straightforward way to make it work in external memory we can obtain the following.

Theorem 5 *There exists positive constants c_1 and c_2 so that there exists a comparison external RAM algorithm* SORTING *N numbers in $c_1 N \log_2 N \leq T_C^{IO} \leq c_2 N^2/(B \log_2 N)$ I/Os and space S_C^{IO} such that $T_C^{IO} \cdot S_C^{IO} = O(\frac{N^2}{B})$.*

It follows from Corollary 1 that Theorem 5 is optimal. Note that this means that traditional internal memory approaches can lead to optimal external memory algorithms when disk space is restricted.

In the more important case where $n \log_m n \lesssim T_C^{IO} \lesssim N \log_2 N$ we can sometimes also obtain optimal bounds. In order to do so we need to make several modifications to the algorithm by Pagter and Rauhe. Their algorithm is based on a complicated binary tree data structure similar to a tournament tree (see e.g [16]). By designing a m-ary version of their data structure we first reduce T_C^{IO} to $O(N^2/(kB) + N \log_2 m \cdot \log_m k)$ using space $O(k \log_2 m)$—i.e. we obtain a data structure where we may vary I/O and space usage by varying the parameter k. To improve this bound we utilize a variant of the buffering technique of Arge [4], as well as the fact that one can sort M elements in internal memory without performing any I/Os. Using these ideas we can further reduce T_C^{IO} to

$$O\left(\frac{\frac{N^2}{k(B+m)} + N \log_m k}{B}\right)$$

increasing the space use with a factor B to $O(Bk \log_2 m)$. By choosing k appropriately we then obtain the following.

Theorem 6 *There exists positive constants c_1 and c_2 so that there exists an comparison external RAM algorithm* SORTING *N numbers in $c_1 n \log_m n \leq T_C^{IO} \leq c_2 N \log_2 N$ I/Os and space S_C^{IO} such that $T_C^{IO} \cdot S_C^{IO} = O(\frac{N^2 \log_2 m}{B+m})$.*

Note that Theorem 6 means that if $M \geq B^2$ (a realistic and standard assumption in the external memory literature) we can sort I/O-optimally (using $O(n \log_m n)$ I/Os) using (sub-optimal) space $O((N \log_2 m)/\log_m n)$. In comparison, the SORTING algorithm of Aggarwal and Vitter [2] uses roughly a factor $\log_2 n$ more space $O(N \log_2 N)$. If also $\log_2 m \leq \log_m n$ we achieve optimal space. We conjecture that our lower bound for SORTING is tight for all values of M and B, that is, that an algorithm achieving a I/O-space product of $O(N^2/B)$ exists.

5 Acknowledgments

We would like to thank an anonymous referee for valuable comments on presentation. The second author would like to thank Faith Fich.

References

1. M. Adler. New coding techniques for improved bandwidth utilization. In *Proc. 37th Annual Symposium on Foundations of Computer Science*, pages 173–182, 1996.
2. A. Aggarwal and J. S. Vitter. The Input/Output Complexity of Sorting and Related Problems. *Communications of the ACM*, 31(9):1116–1127, 1988.
3. M. Ajtai. Determinism versus Non-Determinism for Linear Time RAMs with Memory Restrictions. In *Proc. Thirty-First ACM Symposium on Theory of Computing*, 1999.

4. L. Arge. The Buffer Tree: A New Technique for Optimal I/O-Algorithms. In *Proc. Workshop on Algorithms and Data Structures, LNCS 955*, pages 334–345, 1995. A complete version appears as BRICS technical report RS-96-28, University of Aarhus.
5. L. Arge. *Efficient External-Memory Data Structures and Applications.* PhD thesis, University of Aarhus, 1996.
6. L. Arge, M. Knudsen, and K. Larsen. A General Lower Bound on the I/O-Complexity of Comparison-based Algorithms. In *Proc. of the Workshop on Algorithms and Datastructures, LCNS 709*, pages 83–94, 1993.
7. L. Arge and P. B. Miltersen. On showing lower bounds for external-memory computational geometry problems. In J. Abello and J. S. Vitter, editors, *External Memory Algorithms and Visualization.* AMS Press, 1999.
8. L. Arge and J. Pagter. I/O-Space Trade-Offs. Technical Report ??, BRICS, 2000.
9. P. Beame. A General Sequential Time-Space Tradeoff for Finding Unique Elements. *SIAM Journal on Computing*, 20:270–277, 1991.
10. A. Borodin and S. Cook. A Time-Space Tradeoff for Sorting on a General Sequential Model of Computation. *SIAM Journal on Computing*, 11(2):287–297, 1982.
11. A. Borodin, F. E. Fich, F. Meyer auf der Heide, E. Upfal, and A. Wigderson. A Time-Space Tradeoff for Element Distinctness. *SIAM Journal on Computing*, 16:97–99, 1987.
12. A. Borodin, M. J. Fischer, D. G. Kirkpatrick, N. A. Lynch, and M. Tompa. A Time-Space Tradeoff for Sorting on Non-Oblivious Machines. *Journal of Computer and System Sciences*, 22:351–364, 1981.
13. G. N. Frederickson. Upper Bounds for Time-Space Trade-offs in Sorting and Selection. *Journal of Computer and Systems Sciences*, 34:19–26, 1987.
14. J. W. Hong and H. T. Kung. I/O complexity: The red-blue pebble game. In *Proc. ACM Symp. on Theory of Computation*, pages 326–333, 1981.
15. M. Karchmer. Two Time-Space Tradeoffs for Element Distinctness. *Theoretical Computer Science*, 47:237–246, 1986.
16. D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming.* Addison-Wesley, 2nd edition, 1998.
17. Y. Mansour, N. Nisan, and P. Tiwari. The Computational Complexity of Universal Hashing. *Theoretical Computer Science*, (107):121–133, 1993.
18. K. Munagala and A. Ranade. I/O-complexity of graph algorithm. In *Proc. Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 687–694, 1999.
19. J. I. Munro and M. S. Paterson. Selection and Sorting with Limited Storage. *Theoretical Computer Science*, 12:315–323, 1980.
20. J. Pagter and T. Rauhe. Optimal Time-Space Trade-Offs for Sorting. In *proc. 39th Annual Symposium on Foundations of Computer Science*, pages 264–268, 1998.
21. J. E. Savage. *Models of Computation.* Addison-Wesley, 1998.
22. M. Tompa. Time-Space Tradeoffs for Straight-Line and Branching Programs. Technical Report 122/78, Department of Computer Science, University of Toronto, July 1978. Ph.D. Thesis.
23. J. S. Vitter. External memory algorithms and data structures. In J. Abello and J. S. Vitter, editors, *External Memory Algorithms and Visualization.* AMS Press, 1999.
24. A. C.-C. Yao. Near-optimal Time-Space Tradeoff for Element Distinctness. *SIAM Journal on Computing*, 23:966–975, 1994.