Organic Mergesort and Finger Buffer-Tree Sort: Adaptive Sorting Algorithms with External-Memory or Parallel Implementations

Gerth Stølting Brodal
Aarhus Univ.
Aarhus, Denmark
gerth@cs.au.dk

Michael T. Goodrich
Univ. of Calif., Irvine
Irvine, USA
goodrich@uci.edu

Ryuto Kitagawa
Univ. of Calif., Irvine
Irvine, USA
ryutok@uci.edu

Nodari Sitchinava
Univ. of Hawaii at Manoa
Honolulu, USA
nodari@hawaii.edu

Rolf Svenning
Aarhus Univ.
Aarhus, Denmark
rolfsvenning@cs.au.dk

***Abstract*—We describe two adaptive sorting paradigms, *organic mergesort* and *finger buffer-tree sort*, which have parallel or external-memory implementations. Depending on the implementation regime, our mergesort algorithms have optimal time, work, and/or input/output (I/O) operations with respect to the run-based entropy of the input sequence, which matches the performance of optimal sequential internal-memory natural mergesort algorithms, such as Timsort, Peeksort, and Powersort.**

***Index Terms*—adaptive sorting, entropy, parallel algorithms, external memory, cache-aware, cache-oblivious**

I. Introduction

Knuth [26] introduced ***natural mergesort***, where we first find maximal increasing or decreasing runs in an input sequence, X, of N comparable elements and then perform a mergesort procedure starting from these runs.[1] For example, if we let $\rho(X)$ denote the number of such runs, then we can define a complete binary merge tree, T, of depth $O(\log \rho(X))$, where each internal node corresponds to a merge operation and each leaf corresponds to a run.[2] By a simple analysis based on observing that each level of T uses $O(N)$ time, the running time of such a natural mergesort is $O(N(1 + \log \rho(X)))$. For example, if X consists of a constant number of bitonic sequences [28], then the running time of natural mergesort is $O(N)$, whereas the traditional mergesort algorithm would run in $\Theta(N \log N)$ time; see, e.g., Cormen, Leiserson, Rivest, and Stein [17]. Perhaps because of this history, any mergesort algorithm that starts with maximally increasing or decreasing runs is called a "natural mergesort", and there are many such variants; see, e.g., [5], [6], [13], [21], [25], [33], [35], [36].

Let $X = \langle x_1, x_2, \ldots, x_N \rangle$ be an input sequence of distinct elements that come from a total order, and let $\mathcal{R} = \{R_1, R_2, \ldots, R_{\rho(X)}\}$ be a partition of X into a set of maximal increasing or decreasing runs. Let r_i denote the size, $|R_i|$, of the i-th run in X; so $\sum_{i=1}^{\rho(X)} r_i = N$. The ***run-based entropy***, $\mathcal{H}(X)$, for X is a value in the range 0 to $\log N$ defined as follows [5], [6], [13], [21], [25], [33], [35], [36]:

$$\mathcal{H}(X) = \sum_{i=1}^{\rho(X)} \left(\frac{r_i}{N}\right) \log \left(\frac{N}{r_i}\right).$$

Several researchers have noted that any sequential comparison-based sorting algorithm has a lower bound for its running time that is $\Omega(N(1 + \mathcal{H}(X)))$ [6], [36] and there are many sequential natural mergesort algorithms whose running time is $O(N(1 + \mathcal{H}(X)))$, which can be as small as $O(N)$ depending on the input instance. Such algorithms are known as ***adaptive*** or ***instance-optimal*** sorting algorithms. For example, Munro and Wild [33] introduce PeekSort and PowerSort, which have this running time, and Jugé [25] provides adaptive ShiversSort, which also has this running time; see also, e.g., Gelling, Nebel, Smith, and Wild [21]. In addition, Auger, Jugé, Nicaud, and Pivoteau [5] show that the popular Timsort algorithm has this running time as well.[3] Further, other mergesort algorithms that have been shown to run in $O(N(1 + \mathcal{H}(X)))$ time include Buss and Knop's α-MergeSort [13], Takaoka's MinimalSort [36], Schou and Wang's PersiSort [35], and adaptive search-tree sorting by Barbay and Navarro [6]. Although these algorithms achieve running times that are optimal with respect to the run-based entropy of the input sequence, they don't necessarily result in merge trees that have $O(\log \rho(X))$ depth, as in Knuth's natural mergersort [26]. Such a merge tree would be useful, for instance, in parallel and external-memory implementations. However, we are not aware of previous work on parallel sorting algorithms that have been shown to have optimal work with respect to run-based entropy, nor of previous external-memory algorithms that are optimal with respect to I/O run-based entropy.

Another important measure is the number of ***inversions*** in an input sequence, X, $\mathrm{Inv}(X) = |\{(i, j) \mid 1 \leq i < j \leq$

This material is based upon work performed while attending AlgoPARC Workshop on Parallel Algorithms and Data Structures at the University of Hawaii at Manoa, in part supported by the National Science Foundation under Grant No. 2452276.

[1] For the sake of consistency with the external-memory literature, throughout this paper we use N to denote the number of elements in the input sequence to be sorted.

[2] All logarithms used in this paper are base 2 unless otherwise stated.

[3] Timsort was recently replaced by PowerSort in the CPython reference implementation of Python [21].

$N \wedge x_i > x_j\}|$. Brodal, Fagerberg, and Moruz [11] give cache-oblivious and external-memory adaptive sorting algorithms that are optimal with respect to a number of different measures of near-sortedness, including $\mathrm{Inv}(X)$, but their work does not show that their methods are optimal with respect to run-based entropy.

A. Additional Related Work

There is considerable work on adaptive sorting algorithms; see, e.g., the survey paper by Estivill-Castro and Wood [18], which posed the meta-problems of designing parallel algorithms or external-memory algorithms that are optimal with respect to a reasonable measure of near-sortedness. This is a meta-problem because there are several measures of near-sortedness for a sequence, X, and it is an open problem whether there is a single measure of near-sortedness that subsumes all known measures; see, e.g., [7], [14], [18]. Besides run-based entropy and $\mathrm{Inv}(X)$, other measures include $\mathrm{Rem}(X)$, the minimum number of elements that can be removed to result in a sorted sequence, and $\rho(X)$, the number of maximal increasing or decreasing consecutive subsequences, i.e., runs. Note that $\mathrm{Rem}(X)$ is $\Omega(\rho(X))$, but $\rho(X)$ can be asymptotically smaller than $\mathrm{Rem}(X)$, and it is easy to construct a sequence such that $\mathrm{Inv}(X)$ is $\Theta\big(N^2\big)$ but $\mathcal{H}(X)$ is $O(1)$.[4]

Although we are not aware of any previous work on parallel algorithms that has been shown to have optimal work with respect to the run-based entropy of the input sequence, there is prior work on parallel algorithms for other measures of near-sortedness. Carlsson and Chen [14] give a parallel algorithm with $O(\log N \log \rho(X))$ span and $O(N(1 + \log \rho(X)))$ work in the EREW PRAM model. Their algorithm does not achieve optimal span or work, however, as it requires super-logarithmic span when $\rho(X)$ is not constant and it does not, in general, achieve a work bound of $O(N(1 + \mathcal{H}(X)))$. Levcopoulos and Petersson [29] give a parallel sorting algorithm that runs in $O(\log N)$ time in the EREW PRAM model using $O(N + N \log \mathrm{Rem}(X))$ work. Levcopoulos and Petersson [30] also present a parallel sorting algorithm that runs in $O(\log N)$ time and $O(N(1 + \log(\mathrm{Inv}(X)/N)))$ work in the EREW PRAM model. Chen and Levcopoulos [15] give a parallel algorithm that has $O(\log N)$ span and $O(N(1 + \log(\mathrm{Osc}(X)/N)))$ work in the CRCW PRAM model, where $\mathrm{Osc}(X)$ is a measure of the number of oscillations in X.[5] Their algorithm does not achieve optimal span or work, however, as sublogarithmic time sorting is possible in the CRCW PRAM model, and it is easy to construct examples where their algorithm uses $\Theta(N \log N)$ work but $\mathcal{H}(X)$ is $O(1)$.[6]

Recently, Blelloch and Dobson [7] provide a parallel sorting algorithm in the binary-forking model that has $O\big(\log^3 N\big)$ span and $O(\mathrm{LIB}(X))$ work, where $\mathrm{LIB}(X)$ is an alternative measure of disorder, which they call the "Log-Interleave Bound." Although their algorithm is work-optimal with respect to $\mathrm{LIB}(X)$, it does not achieve a good span and the $\mathrm{LIB}(X)$ measure is itself fairly complex and is not optimal with respect to other measures of sortedness [7].

Aggarwal and Vitter [1] introduce the external-memory model, which has explicit parameters for input size, N, internal memory size, M, and block size, B, and they gave external-memory sorting algorithms that are optimal in the worst case in terms of their I/O bounds. Frigo *et al.* [20] introduce the cache-oblivious model, which does not require such explicit parameters, and they provide optimal cache-oblivious sorting algorithms, including an algorithm known as ***funnelsort***. Brodal *et al.* [8] provide a simplified cache-oblivious sorting algorithm, which they call ***lazy funnelsort***. Also, Brodal *et al.* [9] provide a cache-oblivious priority queue based on funnels [9]. None of these results consider an I/O analog of run-based entropy, however.

B. Our Results

We describe simple adaptive sorting algorithms, ***organic mergesort*** and ***finger buffer-tree sort***, which can be implemented in parallel or in external memory. Depending on the implementation regime, which also includes a cache-oblivious implementation of organic mergesort, our algorithms have optimal time, work, or input/output (I/O) operations with respect to the run-based entropy of the input sequence, which matches the performance of optimal sequential natural mergesort algorithms, such as Timsort, Peeksort, and Powersort. We refer to our mergesort algorithms as "organic" because they are natural mergesort algorithms that also have parallel or external-memory implementations that are optimal with respect to the run-based entropy of the input sequence. Our finger buffer-tree sorting algorithm, on the other hand, involves adding an interesting finger search capability to the buffer tree external-memory data structure of Arge [2], which is a simple tree structure supporting sorting and batched search tree operations and supports a priority queue implementation. The original buffer tree only supported updates in a top-down fashion, whereas the finger buffer tree supports updates that move bottom-up and top-down.

II. Preliminaries

Let $X = \langle x_1, x_2, \ldots, x_N \rangle$ be an input sequence of elements that come from a total order, such that X is stored as an array in shared memory. In this paper, we are interested in comparison-based algorithms; hence, without loss of generality, we assume the elements in X are distinct; otherwise, we can replace each element x_i with the pair, (x_i, i), and perform comparisons lexicographically. Define a ***run*** in X to be a strictly increasing[7] contiguous subsequence of X, i.e., $\langle x_i, x_{i+1}, \ldots, x_j \rangle$ such that $x_k < x_{k+1}$ for $i \leq k < j$. Let

[4] E.g., suppose $X = \langle 1, 3, 5, 7, \ldots, \lceil N/2 \rceil, 2, 4, 6, \ldots, \lfloor N/2 \rfloor \rangle$ for odd N.

[5] For each x_i in X, define $C(x_i) = \{j \mid \min\{x_j, x_{j+1}\} < x_i < \max\{x_j, x_{j+1}\}\}$. Then $\mathrm{Osc}(X) = \sum_{i=1}^{N} |C(x_i)|$.

[6] Again, suppose $X = \langle 1, 3, 5, 7, \ldots, \lceil N/2 \rceil, 2, 4, 6, \ldots, \lfloor N/2 \rfloor \rangle$ for odd N.

[7] We define runs to be strictly increasing in the remainder of this paper, since we can detect all maximal increasing and decreasing contiguous subsequences and reverse the decreasing ones.

$\mathcal{R} = \{R_1, R_2, \ldots, R_{\rho(X)}\}$ be a partition of X into a set of maximal increasing runs. We let r_i denote the size, $|R_i|$, of the i-th run in X. Thus, $\sum_{i=1}^{\rho(X)} r_i = N$.

Any natural mergesort algorithm can be associated with a merge tree, T, which has the runs of $\mathcal{R}$ as its leaves so that each internal node, v in T, is associated with a merge operation for v's children. If we define the ***size***, N_v, of a node v in T to be the number of elements associated with descendants of v (including v itself), then we can define the ***weight***, $w(T)$, of T as

$$w(T) = \sum_{v \in T} N_v.$$

For each node, v in T, define the set, S_v, to be the union of all the elements stored in the descendants of v, where we consider a leaf to be a descendant of itself; hence, $N_v = |S_v|$. The goal is to produce a sorted representation of S_r where r is the root of T, with $O(w(T))$ work.

The efficiency of a natural mergesort algorithm is strongly related to the structure of its associated merge tree, T. For example, in a sequential natural mergesort algorithm, we perform the merge for each internal node, v in T, in $O(N_v)$ time. Thus, the running time of a sequential natural mergesort algorithm with merge tree T is $O(w(T))$.

In a sequential natural mergesort algorithm, such as Timsort [5], we don't necessarily need to define its mergesort tree, T, in advance, but for our organic mergesort algorithms, we use T explicitly, in that we divide our parallel mergesort algorithms into two phases. In Phase 1, we are given the input sequence, X, and we construct a set, $\mathcal{R}$, of maximal runs for X and a merge tree, T, for $\mathcal{R}$. In Phase 2, we are given X, $\mathcal{R}$, and T, and we output an ordered copy of S_r, where r is the root of T.

A. Parallel Models

We are interested in parallel sorting algorithms, where a collection of physical or virtual processors work together to sort an input sequence, X, that is initially stored in their shared memory space. In the Parallel RAM (PRAM) model [24], processors compute synchronously. In the Exclusive-read, Exclusive-write (EREW) PRAM model, every read and write must be exclusive, in the Concurrent-read, Exclusive-write (CREW) PRAM model, concurrent reads are allowed but writes must be exclusive, and in the Concurrent-read, Concurrent-write (CRCW) PRAM model, concurrent reads and writes are allowed, with concurrent writes being resolved by some rule; see, e.g., [24].

In such models, the parallel ***time*** (or ***span***) of an algorithm is the minimum number of parallel steps performed and the ***work*** is the total number of primitive operations performed. Intuitively, the span of a parallel algorithm is the time that could be achieved by implementing the algorithm with an unbounded number of processors and the work of a parallel algorithm is the running time that could be achieved by implementing it with a single processor. Naturally, we are interested in parallel algorithms with small span and work.

B. External-Memory Sorting

In the external-memory model [1], we consider memory divided into multiple layers, with two being the most critical for performance purposes. In the faster, smaller layer, which we call "internal memory," we have a random access memory of size M, and in the larger, slower layer, which we call "external memory," we have the initial input of size N as well as auxiliary storage. Because of the delays in moving data between these two layers, memory transfers are done in blocks of size $2 \leq B < M$, referred to as I/Os. For example, scanning through the blocks of the input takes $\Theta(N/B)$ I/Os. In the worst case, sorting requires $\Theta(\mathrm{Sort}(N))$ I/Os, where $\mathrm{Sort}(N) = (N/B)\log_{M/B}(N/B)$ [1]. Moreover, with respect to the inversions measure, $\mathrm{Inv}(X)$, for an input sequence, X, of length N, Brodal *et al.* [11] show that sorting requires $\Theta\Big((N/B)(1+\log_{M/B}(1+\mathrm{Inv}(X)/N))\Big)$ I/Os under a tall cache assumption. As mentioned above, it is often desirable to design external-memory algorithms that do not need to be explicitly defined in terms of the parameters, M and B, and such algorithms are known as ***cache-oblivious*** algorithms; see, e.g., [8]–[11], [20].

Since we are not aware of previous results for sorting in external-memory with respect to the run-based entropy of the input sequence, let us define the ***run-based I/O entropy*** of an input sequence, X, of size N that can be partitioned in to a set, $(R) = \{R_1, R_2, \ldots, R_{\rho(X)}\}$, of maximal increasing or decreasing runs, with $r_i = |R_i|$, as follows:

$$\mathcal{H}_{\mathrm{I/O}}(X) = \sum_{i=1}^{\rho(X)} \left(\frac{r_i}{N}\right) \log_{M/B}\left(\frac{N}{r_i}\right) = \frac{1}{\log \frac{M}{B}} \cdot \mathcal{H}(X).$$

In this framework, sorting has the following lower bounds:

Theorem 1. *A comparison-based sorting algorithm must perform* $\Omega\big((N/B)(1+\mathcal{H}_{\mathrm{I/O}}(X))\big) - O\Big(\frac{N}{B}\log_{M/B} B\Big)$ *I/Os to sort a sequence, X, of size N. If $M = \Omega\big(B^2\big)$ then the lower bound is* $\Omega\big((N/B)(1+\mathcal{H}_{\mathrm{I/O}}(X))\big)$.

Proof. Let the runs in X have sizes, $r_1, r_2, ..., r_{\rho(X)}$. Takaoka (Lemma 3 [36]) showed that the number of comparisons to sort X is at least $\frac{1}{4}N \cdot \mathcal{H}(X)$. The approach was to give a lower bound of $\left(\frac{N!}{r_1!\cdots r_{\rho(X)}!}\right)\frac{r_1\cdots r_{\rho(X)}}{N(N-1)\cdots(N-\rho(X)+1)}$ for the number of permutations with the same entropy as X and considering the depth of the comparison tree for these permutations as in [31], [34]. Barbay and Navarro showed a similar lower bound for the number of comparisons (Theorem 2 [6]). Using the general technique by Arge *et al.* (Corollary 5 [3]) for converting comparison-based lower bounds into external memory lower bounds yields that the following number of I/Os are necessary to sort X:

$$\begin{aligned}\frac{N(\frac{1}{4}\mathcal{H}(X) - \log B)}{B\log\frac{M-B}{B} + 3B} &\geq \frac{\frac{1}{4}N\cdot\mathcal{H}(X)}{4B\log\frac{M}{B}} - \frac{N\log B}{B\log\frac{M}{B}} \\ &= \frac{1}{16}\frac{N}{B}\mathcal{H}_{\mathrm{I/O}} - \frac{N}{B}\log_{M/B} B\,.\end{aligned}$$

Additionally, N/B I/Os are, of course, necessary to read the input, so this term is also required. Under the tall-cache assumption $\log_{M/B} B \leq 1$ and sorting requires $\Omega\big(\frac{N}{B}\left(1 + \mathcal{H}_{1/O}\right)\big)$ I/Os. □

The above lower bound is similar to those for external-memory multiple selection and duplicate removal [12], [19].

Theorem 2. *Let X be a sequence of N indivisible items that can be partitioned into a set, $\mathcal{R} = \{R_1, R_2, \ldots, R_{\rho(X)}\}$, of maximal increasing or decreasing runs, such that $|R_i| = r_i$. Sorting X requires $\Omega(\min\{A, B\})$ I/Os where,*

$$A = N - \sum_{i=1}^{\rho(X)} \frac{r_i \log r_i}{\log N}$$

$$B = \mathrm{Sort}(N) - \sum_{i=1}^{\rho(X)} \frac{r_i}{B} \log_{M/B} r_i.$$

Proof. The proof is an adaptation of the lower bound of Aggarwal and Vitter for an arbitrary permutation [1]. They proved that an algorithm that performs T I/Os on input of size N can generate at most $(B!)^{N/B}\left(N(1+\log N)\binom{M}{B}\right)^T$ distinct permutations. The number of distinct sequences X that consist of $\rho(X)$ strictly increasing or decreasing contiguous subsequences of sizes $r_1, r_2, \ldots, r_{\rho(X)}$ is $N!/\prod_{i=1}^{\rho(X)} r_i!$. Therefore, the minimum number of I/Os required to generate any one of such sequences in the worst case must be:

$$\begin{aligned} T &\geq \frac{\log(N!) - \log\left(\prod_{i=1}^{\rho(X)} r_i!\right) - \log\left((B!)^{N/B}\right)}{\log\left(N(1+\log N)\binom{M}{B}\right)} \\ &= \Omega\left(\frac{N\log(N/B) - \sum_{i=1}^{\rho(X)} r_i \log r_i}{\log N + B\log\left(\frac{M}{B}\right)}\right), \end{aligned}$$

by applying the inequalities $\binom{n}{k} \leq \left(\frac{n\cdot e}{k}\right)^k$ and $\left(\frac{k}{3}\right)^k \leq k! \leq \left(\frac{k}{2}\right)^k$. There are two cases to consider:

1) If $\log N > B\log\left(\frac{M}{B}\right)$, then we have $B < \log N$, i.e., $\log(N/B) = \Omega(\log N)$. Thus,

$$T = \Omega\left(N - \sum_{i=1}^{\rho(X)} \frac{r_i \log r_i}{\log N}\right).$$

2) If $\log N \leq B\log\left(\frac{M}{B}\right)$, then we have the following:

$$\begin{aligned} T &= \Omega\left(\frac{N\log\frac{N}{B} - \sum_{i=1}^{\rho(X)} r_i \log r_i}{B\log\frac{M}{B}}\right) \\ &= \Omega\left(Sort(N) - \sum_{i=1}^{\rho(X)} \frac{r_i}{B}\log_{M/B} r_i\right). \end{aligned}$$

□

III. Finding Runs

We begin by describing simple algorithms for finding maximal runs in the input sequence, X. Of course, sequentially finding the runs in the input sequence, X, is easy to do in linear time. We simply scan X, noting when consecutive elements are increasing or decreasing, marking where transitions occur. Likewise, this algorithm also can easily be implemented in external-memory or cache-oblivious contexts to use $O(N/B)$ I/Os.

Finding runs in parallel is also relatively simple. For each element, x_i in X, we compare x_i with x_{i+1} and set a less-than bit, $L[i] = 1$ if $x_i < x_{i+1}$ and otherwise we set $L[i] = 0$. Thus, an increasing run in X is a maximal sequence of consecutive 1's in L and a maximal decreasing run in X is a maximal sequence of consecutive 0's in L. We can thus identify the runs in X by two parallel prefix operations, which have $O(\log N)$ span and $O(N)$ work in the CREW PRAM model; see, e.g., [24]. Once we have identified the decreasing runs in X, we can easily reverse them to become increasing runs using an algorithm with span $O(1)$ and work $O(N)$. Thus, let us assume without loss of generality that all runs in X are increasing.

Theorem 3. *Given a sequence, X, of N comparable elements, we can find a set of runs in X in $O(\log N)$ span and $O(N)$ work in the CREW PRAM. This can also be done in external-memory or cache obliviously with $O(N/B)$ I/Os.*

IV. Parallel Organic Mergesort

In this section, we describe our parallel organic mergesort algorithms. Let us assume we have a set, $\mathcal{R} = \{R_1, R_2, \ldots, R_{\rho(X)}\}$, of runs in the input sequence, X, where the runs are listed left-to-right. In the case of external-memory or CREW PRAM implementations, this set is represented explicitly and for the CRCW PRAM implementation, it is represented implicitly, as explained above.

A. Constructing a Low-Weight Merge Tree

We begin by constructing a merge tree with weight $O(N(1+\mathcal{H}(X)))$ in parallel. Our algorithm is a method that mixes two classic algorithms that Mehlhorn identifies as "Method 1" and "Method 2" for building a near-optimal binary search tree [32]. Incidentally, the Method 1 and Method 2 search trees are separately used in the sequential Peeksort and Powersort [33] algorithms, respectively.

Let us describe our parallel method first. We start by grouping the runs in $\mathcal{R} = (R_1, \ldots, R_{\rho(X)})$ into contiguous subsequences of $\Theta(\log \rho(X))$ runs and building a merge tree for each group sequentially using the linear-time algorithm of Mehlhorn. We then assign a virtual processor to each such group and treat its range of runs as a single "super-run," R'_i, for the sake of constructing the merge tree, T, which we build top down. At each level, we maintain the invariant that we have a node, v in T, at that level that is associated with a consecutive subsequence of super-runs, $\mathcal{R}'_v = \langle R'_s, \ldots, R'_t\rangle$, that go from

x_i to x_j, i.e., where $R'_s = \langle x_i, \ldots \rangle$ and $R'_t = \langle \ldots, x_j \rangle$. The way we choose a splitter for v depends on whether v is at an odd level or even level in T. If v is at an even level (assuming the root of T is at level 0), we choose $m = \lfloor (i+j)/2 \rfloor$ and let R'_u denote the super-run that includes x_m. Then we choose the split value for v to be the index of R'_u closest to m. That is, on even levels we split the total sizes of the sets of the super-runs roughly in half. On the other hand, if v is at an odd level, then we compute m as above but we choose our splitter as the index of $R'_{\lfloor (s+t)/2 \rfloor}$ closest to m. That is, on odd levels we divide by two the number of super-runs being considered. If we have a processor assigned to each super-run in $\mathcal{R}'_v$, we can do all of these steps in $O(1)$ span in the CREW PRAM, since concurrent reads may be required for all processors to learn of the splitter values. We then recursively determine splitters for each of v's two children if it is associated with more than two runs in $\mathcal{R}'_v$ after performing the split for R_u. Once we reach a node in T that is associated with a single super-run, we splice in at this node the near-optimal binary tree that we constructed sequentially for this super-run. See Figure 1.

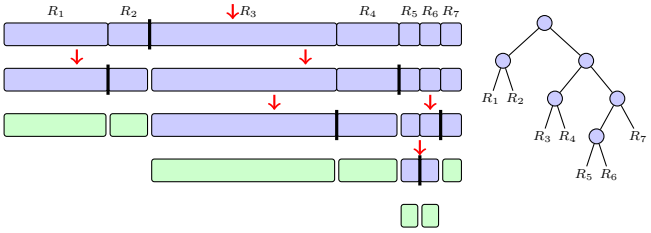

Fig. 1. Construction of the merge-tree for organic mergesort with seven groups of runs. On the left, for each level, the red arrows indicate the middle elements x_m for each subdivision, and the vertical black lines the selected splitters. The resulting merge tree is shown on the right.

Note that because of the even-level splits, the depth of T is $O(\log \rho(X))$. Moreover, based on an analysis similar to that given by Mehlhorn [32] for Method 1, because of the odd-level splits, the total weight of T is $O(N(1+\mathcal{H}(X)))$. Moreover, the total span for this algorithm is at most $O(\log \rho(X))$ and the total work is $O(\rho(X))$. Thus, we have:

Theorem 4. *Given a sequence, $X = \langle x_1, x_2, \ldots, x_N \rangle$, and a set of runs for X, $\mathcal{R} = (R_1, \ldots, R_{\rho(X)})$, we can construct a merge tree, T, with height $O(\log \rho(X))$ and weight $O(N(1+\mathcal{H}(X)))$ in $O(\log \rho(X))$ span and $O(\rho(X))$ work in the CREW PRAM model.*

B. Performing the Merge Operations in Waves

We first give a CREW PRAM sorting algorithm that can run in sublogarithmic time, depending on $\rho(X)$. Let us describe this algorithm assuming we are given a sequence, $X = (x_1, x_2, \ldots, x_N)$, a set of maximal increasing runs, $\mathcal{R} = (R_1, \ldots, R_{\rho(X)})$, for X, and a merge tree, T, with height $O(\log \rho(X))$ and weight $O(N(1+\mathcal{H}(X)))$, as described above.

Given these inputs, our bottom-up level-by-level algorithm is a simple method that has $O(\log \rho(X) \cdot \log\log N)$ span and $O(N(1+\mathcal{H}(X)))$ work in the CREW PRAM model. The algorithm is simply to perform the merges defined by T level-by-level in a bottom-up fashion, starting with the lowest level in T. By an algorithm due to Kruskal [27], each merge of two sorted arrays of total size m can be performed with $O(\log\log m)$ span and $O(m)$ work. Thus, we have:

Theorem 5. *Given a sequence, X, of N comparable elements that can be partitioned into a set of runs, $\mathcal{R} = (R_1, \ldots, R_{\rho(X)})$, we can sort X in $O(N(1+\mathcal{H}(X)))$ work and $O(\log N + \log \rho(X) \log\log N)$ span in the CREW PRAM model.*

C. Performing the Merge Operations in a Pipelined Fashion

In this subsection, we describe our parallel implementation of the organic mergesort algorithm that runs in $O(\log N)$ time using $O(N(1+\mathcal{H}(X)))$ work in the CREW PRAM model. There are several challenges needing to be overcome in order to achieve these bounds, and our CREW PRAM method is a generalization of non-adaptive pipelined parallel mergesort algorithms [4], [16], [22] to an organic merge tree where we start with runs at its leaves instead of singleton sets.

Given a sequence, $X = (x_1, x_2, \ldots, x_N)$, of comparable elements, let us assume we are given a set of maximal runs, $\mathcal{R} = (R_1, \ldots, R_{\rho(X)})$, and a merge tree, T, for $\mathcal{R}$ such that the total weight of T is $O(N(1+\mathcal{H}(X)))$. Each run, R_i, is associated with a leaf, v_i, and for each node v in T, we also define S_v to be the set of all elements stored in descendants of v in T, including v itself. So $S_{v_i} = R_i$ if v_i is a leaf.

Previous (non-adaptive) pipelined mergesort algorithms [4], [16], [22] assume that each leaf holds only a single element and that T is a complete binary tree; hence, they begin their pipelined merge operations in a level-by-level fashion starting from the leaves. In our case, however, we must delay starting the merges for some leaves so as to coordinate the merges at internal nodes so that each internal node is receiving elements from its two children in as balanced a way as possible. Our method is an adaptation of the cascading divide-and-conquer technique of Atallah, Cole, and Goodrich [4] to parallel organic mergesort using T.

Atallah, Cole, and Goodrich [4] showed how to construct a sorted array, $U(v)$, for each node in a merge tree, T, where each leaf stores a singleton and $U(v)$ consists of the union of all the elements stored at v, including v itself. Their method runs in $O(h(T))$ time using $O(w(T))$ work in the CREW PRAM model, where $h(T)$ denotes the height of T. It achieves these bounds by taking samples of the lists at one level in T and using these samples to begin performing the merges at the next level up in T. Their method consists of $O(h(T))$ stages, each of which can be implemented in $O(1)$ time, and we follow this same approach.

Let a, b, and c be three items, with $a \leq b$. We say c is ***between*** a and b if $a < c \leq b$. Let two sorted lists, $A = (a_1, a_2, \ldots, a_n)$ and $B = (b_1, b_2, \ldots, b_m)$, be given. Given an element a we define the ***predecessor*** of a in B to be the greatest element in B that is less than or equal to a. If $a < b_1$,

then we say that the predecessor of a is $-\infty$. We define the ***rank*** of a in B to be the rank of the predecessor of a in B ($-\infty$ has rank 0). We say that A is ***ranked*** in B if for every element in A we know its rank in B. We say that A and B are ***cross ranked*** if A is ranked in B and B is ranked in A. We define two operations on sorted lists. We define $A \cup B$ to be the sorted ***merged*** list of all the elements in A or B. If B is a subset of A, then we define $A - B$ to be the sorted list of the elements in A that are not in B.

Let T be a binary merge tree. For any node v in T we let $parent(v)$, $sibling(v)$, $lchild(v)$, $rchild(v)$, and $depth(v)$ denote the parent of v, the sibling of v, the left child of v, the right child of v, and the depth of v (the root is at depth 0), respectively. We also let $root(T)$ denote the root node of T. Define the ***weighted height***, $h'(T)$, of T as follows:

$$h'(v) = \begin{cases} \lceil \log |U(v)| \rceil, \text{ if } v \text{ is a leaf with sorted set } U(v) \\ \max\{h'(lchild(v)), h'(rchild(v))\} + 1, \text{ else} \end{cases}$$

The ***weighted altitude*** of a node, v in T, is defined $alt(v) = h'(T) - depth(v)$. We say that a sorted list L is a ***c-cover*** of a sorted list J if between each two adjacent items in $(-\infty, L, \infty)$ there are at most c items from J (where $(-\infty, L, \infty)$ denotes the list consisting of $-\infty$, followed by the elements of L, followed by ∞). We let $SAMP_c(L)$ denote the sorted list consisting of every c-th element of L, and call this list the ***c-sample*** of L. That is, $SAMP_c(L)$ consists of the c-th element of L followed by the $(2c)$-th element of L, and so on.

The algorithm for constructing $U(v)$ for each internal node, $v \in T$, proceeds in stages. Intuitively, in each stage we will be performing a portion of the merge of $U(lchild(v))$ and $U(rchild(v))$ to give the list $U(v)$. After performing a portion of this merge we will gain some insight into how to perform the merge at v's parent, and we will pass some of the elements formed in the merge at v to v's parent, so we can begin performing the merge at v's parent. Specifically, we denote the list stored at a node v in T at stage s by $U_s(v)$. Initially, $U_0(v)$ is empty for every node except the leaf nodes of T, in which case $U_0(v)$ is the sorted run, $U(v)$, stored at the leaf node v. Each internal node, v in T, is initially ***frozen*** and it becomes ***active*** in the stage in which it is to receive elements from its children. Likewise, v returns to being frozen in the stages after the stage in which it has passed all of its elements in $U(v)$ to its parent. In particular, we say that a node v is ***active*** at stage s if $\lfloor s/3 \rfloor \leq alt(v) \leq s$. We say that an internal node v is ***full*** at stage s if $U_s(v) = U(v)$. For each active internal node $v \in T$ we define the list $U'_{s+1}(v)$ as follows:

$$U'_{s+1}(v) = \begin{cases} SAMP_4(U_s(v)) & \text{if } alt(v) \geq s/3 \\ SAMP_2(U_s(v)) & \text{if } alt(v) = (s-1)/3 \\ SAMP_1(U_s(v)) & \text{if } alt(v) = (s-2)/3. \end{cases}$$

Suppose a leaf, v in T, becomes active at stage $s = 3(alt(v))$, i.e., because $alt(v) \geq \lfloor s/3 \rfloor$. Then let $U'_{s+1}(v) = SAMP_c(U_s(v))$, where $c = \lceil |U(v)|/2^{s-3(alt(v))} \rceil$. For example, if v is a leaf and $|U(v)| \geq 8$ is a power of 2, such that v becomes active in stage s, then $U'_{s+1}(v)$ contains 1 element, $U'_{s+2}(v)$ contains 2 elements, $U'_{s+3}(v)$ contains 4 elements, $U'_{s+4}(v)$ contains 8 elements, and so on. Moreover, the total work to deliver these samples to v's parent is $O(|U(v)|)$. At stage $s+1$ we perform the following computation at each internal node v that is currently active.

Per-Stage Computation$(v, s+1)$**:**

> Form the two lists $U'_{s+1}(lchild(v))$ and $U'_{s+1}(rchild(v))$, and compute the new list $U_{s+1}(v) := U'_{s+1}(lchild(v)) \cup U'_{s+1}(rchild(v))$.

This formalizes the notion that we pass information from the merges performed at the children of v in stage s to the merge being performed at v in stage $s+1$. Note that until an internal node, v, becomes full $U'_{s+1}(v)$ will be the list consisting of every fourth element of $U_s(v)$. This continues to be true about $U'_{s+1}(v)$ up to the point that v becomes full. If s_v is the stage at which an internal node, v, becomes full (and $U_s(v) = U(v)$), then at stage $s_v + 1$, $U'_{s+1}(v)$ is the two-sample of $U_s(v)$, and, at stage $s_v + 2$, $U'_{s+1}(v) = U_s(v)$ ($= U(v)$). Thus, at stage $s_v + 3$, $parent(v)$ is full. Therefore, after $3 \cdot h'(T)$ stages every node has become full and the algorithm terminates. Given the above definitions for sampling, we can perform each merge in $O(1)$ time and linear work, e.g., using adaptations of previous methods for pipelined merging [4], [16], which, for completeness, we give in Section VIII. Thus, we have:

Theorem 6. *Given a sequence, X, of N comparable elements, for which it is possible for X to be partitioned into a set of maximal increasing runs, $\mathcal{R} = (R_1, \ldots, R_{\rho(X)})$, we can sort X with $O(\log N)$ span and $O(N(1 + \mathcal{H}(X)))$ work in the CREW PRAM model.*

V. Cache-Aware Organic Mergesort in External Memory

In this section, we describe our cache-aware organic mergesort algorithm. Given an input sequence, X, of length N, we begin by identifying the maximal increasing or decreasing runs in X, as described above. We then do an additional scan of the list of runs to merge consecutive runs of size smaller than M, so that each remaining run has size at least M. Let $\mathcal{R} = \{R_1, R_2, \ldots, R_{\rho(X)}\}$ be the resulting set of runs, and let $r_i = |R_i|$.

Let T' be the complete merge tree of height $O\Big(\log_{M/B}(N/B)\Big)$ that one would get by performing a standard β-way mergesort of X ignoring the runs, for $\beta = \lceil (M/B)^{1/2} \rceil$, with each leaf associated with a consecutive block of elements from X; see, e.g., [1]. Next, we do an additional scan of the runs in $\mathcal{R}$ to identify for each run $R_i = (x_j, \ldots, x_k)$ in $\mathcal{R}$ the node, v_i in T', that is lowest common ancestor of the leaves for x_j and x_k, respectively. Note that determining the node, v_i, is a simple calculation based on the values of j and k and does not need any I/Os. We create a corresponding node, v_i, in our organic merge tree, T, and we give v_i a child, u, that stores the run, R_i. Moreover, we define the ancestors in T to be the same ancestors of v_i in T'. We can therefore explicitly construct

T' by processing the nodes, $(v_1, v_2, \ldots, v_{\rho(X)})$, in order and creating ancestor nodes as needed. Such a method requires $O(\lceil \rho(X)/B \rceil)$ I/Os and gives us the following.

Lemma 1. *For each run, R_i in $\mathcal{R}$, the depth of the node storing R_i in T is $O\Big(\log_{M/B} \lceil N/(r_i B) \rceil\Big)$.*

Proof. Since the run $R_i = (x_j, \ldots, x_k)$ is of length $|R_i|$ and T' is a complete β-way merge tree built on top of the elements in X, the lowest common ancestor, v in T', of the leaves for x_j and x_k must have height at least $\lfloor \log_\beta r_i \rfloor$. Otherwise, v would not have enough leaf descendants to cover R_i, as would be required for the lowest common ancestor of x_j and x_k in T'. The depth of v, therefore, is at most $\lceil \log_\beta \lceil N/B \rceil \rceil - \lfloor \log_\beta r_i \rfloor$, which is $O\Big(\log_{M/B} \lceil N/(r_i B) \rceil\Big)$. □

Our cache-aware organic mergesort algorithm, then, is a simple generalization of the cache-aware β-way mergesort algorithm, where we perform the merges in T bottom-up, level by level. Each β-way merge we do at a node, v, with descendant runs of total size N_v takes $O(N_v/B)$ I/Os, by the classic β-way merge method; see, e.g., [1]. Thus, by Lemma 1, the total number of I/Os performed by our cache-aware algorithm is $O\big(\sum_{v \in T} \lceil N_v/B \rceil\big)$, which is

$$O\left(\left\lceil \frac{N}{B} \right\rceil + \sum_{i=1}^{\rho(X)} \left\lceil \frac{r_i}{B} \right\rceil \log_{M/B} \left\lceil \frac{N}{r_i B} \right\rceil\right)$$
$$= O\left(\left(\frac{N}{B}\right)\left(1 + \sum_{i=1}^{\rho(X)} \left(\frac{r_i}{N}\right) \log_{M/B} \left\lceil \frac{N}{r_i B} \right\rceil\right)\right).$$

Therefore, we have:

Theorem 7. *Given a sequence, X, of N comparable elements, for which it is possible for X to be partitioned into a set of maximal increasing runs, $\mathcal{R} = (R_1, \ldots, R_{\rho(X)})$, we can sort X using $O\big((N/B)(1 + \mathcal{H}_{\mathrm{I/O}}(X))\big)$ I/Os.*

VI. Cache-Oblivious Organic Mergesort

In this section we describe a cache-oblivious stable implementation of the organic mergesort algorithm, which sorts an input sequence $X = \langle x_1, x_2, \ldots, x_N \rangle$ with $\rho(X)$ increasing and decreasing runs using $O\big(\frac{1}{\varepsilon} N(1 + \mathcal{H}(X))\big)$ comparisons and $O\big(\frac{1}{\varepsilon}\frac{N}{B}\big(1 + \mathcal{H}_{\mathrm{I/O}}(X)\big)\big)$ I/Os, under the tall cache-assumption $M \geq B^{1+\varepsilon}$.

The algorithm proceeds in two phases: In the first phase, while identifying runs and long decreasing runs are reversed into increasing runs, short runs of size at most $\tau = N^{1-\Theta(\varepsilon)}$ are identified and combined to longer increasing runs using a (non-adaptive) sorting algorithm. In the second phase the resulting $O(N/\tau)$ increasing runs are merged in a weight balanced fashion using an $O(N/\tau)$-merger borrowed from [8], [20].

A. Cache-Oblivious Preliminaries

Frigo et al. [20] introduced the cache-oblivious model and described the cache-oblivious sorting algorithm, ***funnelsort***, achieving $O((N/B)(1 + \log_M N))$ I/Os, under a tall cache assumption, $M \geq B^2$. Brodal and Fagerberg [8] relaxed this tall cache assumption to $M \geq B^{1+\varepsilon}$, for any constant $\varepsilon > 0$, and showed that the resulting I/O cost for sorting is $O\big(\frac{1}{\varepsilon}(N/B)(1 + \log_M N)\big)$. Under the relaxed tall cache assumption, the I/O overhead of a factor $\frac{1}{\varepsilon}$ for $M \gg B$ was proved necessary in [10]. Both funnelsort algorithms [8], [20] are stable, since they essentially both perform binary mergesort.

Central to funnelsort is the concept of a k-***merger***. We briefly recall the construction and its usage in lazy funnelsort [8]. The job of a k-merger is to merge k sorted input buffers into a sorted output buffer of size k^d, where $d \geq \max\{2, 1 + 2/\varepsilon\}$ (see Figure 2). Merging is done in a balanced binary tree fashion, where each node merges its two input buffers and outputs it into a buffer that again is the input to its parent. This resembles binary mergesort, but here each node would complete merging before another node is invoked. Crucial to a k-merger is that the buffers have limited capacity, and when a node is invoked it continues merging until its output buffer is full. If an input buffer becomes empty during the merging, the merging at the node is paused while the input buffer is recursively filled by the child merger. Assuming k is a power of two, a k-merger is constructed recursively as follows. If $k = 2$ it is simply a binary node storing the position of the next elements to read from its two input buffers and the position of the next available position in the output buffer. If $k > 2$, we recursively construct a top-merger that is a k'-merger, $k' = 2^{\lceil \frac{1}{2} \log k \rceil} \approx \sqrt{k}$, and k/k' bottom-mergers that are k/k'-mergers, where $k/k' \approx \sqrt{k}$. The mergers are connected using middle buffers of capacity $(k/k')^d \approx k^{d/2}$ (see Figure 2). If the recursive k-mergers are laid out consecutively in memory, an invocation of a k-merger generating $O(k^d)$ output requires the following number of I/Os.

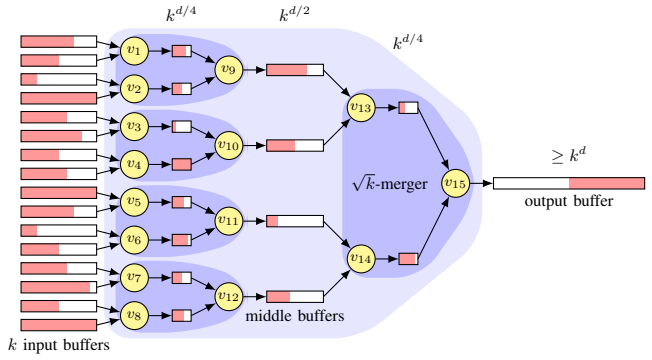

Fig. 2. A k-merger (light blue) recursively partitioned into $\sqrt{k}$-mergers (darker blue). Red are elements in buffers.

B. Entropy Adaptive Sorting Algorithm

In the following, we let $\tau = \lceil 3N^{1-1/d} \rceil$ be the threshold between a run being long or short. In the first phase (see Figure 3), we scan through the sequence to identify all increasing and decreasing runs. We denote a run as ***short*** if it has

size $\leq \tau$, and ***long*** otherwise. Consecutive short runs are made into a single increasing run, possibly long run, by applying a stable cache-oblivious sorting algorithm, e.g., funnelsort [8], [20]. Decreasing long runs are reversed to become increasing long runs. To ensure stability, consecutive elements with equal value should appear in the same order after reversing, i.e., we reverse consecutive elements with equal value back to the initial order. The total cost for the first phase is linear for the scanning and reversing long decreasing runs plus the cost for sorting all short runs, i.e., $O(N + N_s \log N_s)$ comparisons and $O(N/B + \mathrm{Sort}(N_s))$ I/Os, where N_s is the total length of the short runs.

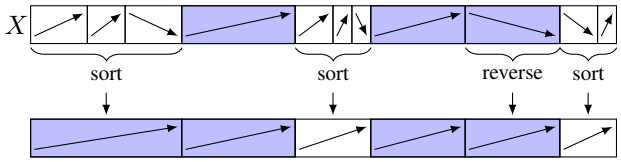

Fig. 3. Cache-oblivious adaptive sorting, first phase. Long runs are shaded blue, arrows indicate if a run is increasing or decreasing

Theorem 8 (Brodal and Fagerberg [8, Lemma 1])**.** *Let $d \geq 2$. The size of a k-merger (excluding its output buffer) is bounded by $c \cdot k^{(d+1)/2}$ for a constant $c \geq 1$. Assuming $B^{(d+1)/(d-1)} \leq M/2c$, a k-merger performs $O\Big(\frac{k^d}{B} \log_M(k^d) + k\Big)$ I/Os during an invocation.*

After the first phase all runs are increasing. There are at most N/τ initial long runs, and the short runs are combined to at most $1 + N/\tau$ (short or long) runs, i.e., the total number of runs after the first phase is at most $k' = 1 + 2N/\tau$ runs. By assigning each long run weight equal to its length and each short run weight τ (and treating them as long runs), the total weight W is at most $\tau + 2N$. We can now construct a weight balanced merge-tree where a node of weight w has depth at most $2 + \log \frac{W}{w}$, e.g., using the cache-efficient algorithm by Munro and Wild from Powersort that uses a stack and causes $O(k'/B)$ I/Os [33]. Note that assigning short runs weight τ, the depth of a short run in the resulting merge-tree is at most $O(1) + \log \frac{N}{\tau}$. Using a $2^{\lceil 2+\log(W/w)\rceil}$-merger, we can embed the weight-balanced tree into the merger and connect the sorted runs directly as input buffers to internal nodes instead at the leafs of the tree (and ignore/prune the corresponding subtrees from the merger). The resulting merger is a weight-balanced merger, where the invocation of the root completely merges all N elements into an output buffer of size N. The bounds of Theorem 8 apply when $k'^d \leq N$, which holds for $\tau \geq 3N^{1-1/d}$, since $k'^d \leq (1+2N/\tau)^d \leq (3N/\tau)^d \leq N$. The total number of comparisons becomes $O(N)$ comparisons to identify the runs, $O(N_s \log N_s)$ comparisons to sort the short runs, and $\sum_{i=1}^{k'} O\Big(r_i \lg \frac{N}{r_i}\Big)$ comparisons to merge the runs. Since a short run of size r satisfies $r \leq \tau = N^{1-\Theta(\varepsilon)}$, we have $\lg \frac{N}{r} = \Theta(\lg N)$. Accordingly, the total number of comparisons becomes $O(N(1 + \mathcal{H}(X)))$. If $N \leq M$ all computation takes place internally after reading the input using $O(N/B)$ I/Os, otherwise $N > M$ and $k' = O(N/B)$. The first phase requires $O(N/B + \mathrm{Sort}(N_S)) = O\Big(N/B + \sum_{r_i \leq \tau} \frac{r_i}{B} \log_M N_S\Big) = O\Big(\frac{N}{B}\big(1 + \sum_{r_i \leq \tau} \frac{r_i}{N} \log_M \frac{N}{r_i}\big)\Big)$ I/Os, since $\log N_S \leq \log N = \Theta\Big(\log \frac{N}{r_i}\Big)$ for $r_i \leq \tau$. An element in a run with weight w moves through $2 + \log \frac{W}{w}$ nodes of the merger of which only every $\Omega(\log M)$ incurs I/Os, contributing $O\big(1 + \frac{w}{B} \log_M \frac{W}{w}\big)$ I/Os to the merging. The total number of I/Os is

$$
\begin{aligned}
&O\left(\frac{N}{B} + \mathrm{Sort}(N_S) + \sum_{i=1}^{k'} \Big(1 + \frac{w_i}{B} \log_M \frac{W}{w_i}\Big)\right) \\
&= O\left(\frac{N}{B} + k' + \frac{N}{B} \sum_{i=1}^{k'} \frac{r_i}{N} \log_M \frac{N}{r_i}\right) \\
&= O\left(\frac{N}{B}\Big(1 + \sum_{i=1}^{k'} \frac{r_i}{N} \log_M \frac{N}{r_i}\Big)\right).
\end{aligned}
$$

Theorem 9. *Given a sequence, X, of N comparable elements, we can sort X in a cache-oblivious manner using $O(N(1 + \mathcal{H}(X)))$ comparisons and $O\big(\frac{N}{B}\big(1 + \mathcal{H}_{\mathrm{I/O}}(X)\big)\big)$ I/Os.*

VII. External-Memory Finger Buffer-Tree Sort

In this section, we introduce the finger buffer-tree, a variant of the buffer-tree that includes a finger pointer to the rightmost leaf node, designed to be a cache-aware search tree data structure. Our focus in this section is solely on the insertion operation, as it is the only operation required for the cache-aware lazy insertion sort algorithm.

The buffer-tree, as described by Arge [2], is a search high fan-out tree data structure, which utilizes "buffers" that are of size relative to the cache-size. These buffers allow for insertion operations to be performed in a "lazy" manner. The lazy insertion is broken up into two stages: the insertion and the tree sweep. Insertions are performed on a single node until a critical mass is reached, at which point we perform a "sweep" on the tree to perform many insertions at once to utilize the cache as efficiently as possible.

Let our finger buffer-tree be an (a, b)-tree [23], T, with $a = \frac{1}{4} \cdot \frac{M}{B}$ and $b = \frac{M}{B}$. T contains a pointer, which we call the ***finger pointer***, to the right most leaf node in the buffer-tree, ℓ_f. Each node in T contains a buffer of size $\Theta(M)$. For every node $v \in T$, let $\mathcal{B}_v$ be the buffer of node v.

For each insertion operation, we wait until B insertion operations have been called on T and stored in internal memory, before we begin the lazy insertion. This ensures that we can apply B insertion operations in a single I/O.

We then begin by placing each element into $\mathcal{B}_{\ell_f}$. If the buffer contains less than $M/2$ elements, it has not yet reached the critical mass of elements, and we are done. Otherwise, we begin the sweeping update process. The approach we use to update the finger buffer-tree is similar to the traditional buffer-tree [2], with the key difference being some elements in the

buffer must be pushed up to the parent node. Therefore, the sweeping update algorithm is split into three phases: emptying buffers up the tree, emptying buffers down the tree, and emptying the leaf node buffers.

The first phase is similar to the finger search in the binary search tree setting, where we move from the finger pointer up to the lowest common ancestor of the search location and the finger pointer. Let v be the node we are currently updating, and $p(v)$ be the parent of v. At the start of the algorithm, $v = \ell_f$. We also maintain a list of nodes to be emptied in the second phase, $\mathcal{L}$.

First, if either $|\mathcal{B}_v| < M/2$ or v is the root, then we add v to $\mathcal{L}$ if v is a root and $|\mathcal{B}_v| < M/2$, and end the first phase. Otherwise, we place any elements in $\mathcal{B}_v$ that belong in $p(v)$ to $\mathcal{B}_{p(v)}$. More precisely, recall that v must contain only values that are between two elements within $p(v)$, thus any element in $\mathcal{B}_v$ which is outside this range does not belong in v. The elements within $\mathcal{B}_{p(v)}$ are not considered at this stage. The process for identifying these elements in a cache-efficient way is straight forward. Load the first $M/2$ elements of $\mathcal{B}_v$ into internal memory, and sort the elements. Then identify the two elements v should contain the elements between and append any elements from $\mathcal{B}_v$ outside that range to $\mathcal{B}_{p(v)}$. If $|\mathcal{B}_v| > M/2$, we append v to $\mathcal{L}$. We then make a recursive call to $p(v)$, until we reach the base case.

The second phase is the process of emptying all nodes in $\mathcal{L}$. This process is performed in a near identical manner to the first phase, with the only difference being we check to see which child node the elements in the buffer should be pushed down to. By the end of the second phase, the only nodes containing buffers that contain more than $M/2$ elements are the leaf nodes.

The third phase is the process of emptying the leaf node buffers. Let ℓ be some leaf node in T with a buffer larger than $M/2$. During this phase, we load the first $M/2$ elements of $\mathcal{B}_\ell$ into internal memory, and insert them into ℓ in sorted order. If ℓ contains more than M elements, a rebalancing operation is performed.

The rebalancing done here is similar to the one in the (a, b)-tree, where the node is split, and a separator element is pushed up to the parent node. In the traditional buffer-tree, since the insertion operations were always occurring from the root to the leaf, recursively splitting nodes that contained more elements than their capacity could trivially ignore the buffer since all buffers on the path from the leaf to the root would be free. Since the finger buffer-tree only clears buffers along a subset of the path from the leaf to root, some nodes along that path may contain buffers with elements in them. The resolution to this is straight forward, by splitting the node v into v' and v'', where all elements in v' are less than v'', by assigning the elements in the buffer to v'' and making the buffer in v' empty, the I/O analysis remains the same.

Let v be a node containing more than M elements. If the $|\mathcal{B}_{p(v)}| > M/2$, then, similar to the process performed in the leaf node, the first $M/2$ elements of $\mathcal{B}_{p(v)}$ are loaded to internal memory, sorted, and inserted to $p(v)$. If the parent node contains more than M elements, then the rebalancing operation is performed recursively up the tree.

Theorem 10. *Let X be a sequence of N elements being inserted into an initially empty finger buffer-tree. The total cost of performing the insertions is $O\Big(N \log_{M/B} \frac{\mathrm{Inv}(X)}{N}\Big)$ I/Os operation, where $\mathrm{Inv}(X)$ is the number of inversions in the input sequence.*

Proof. To prove the above theorem, we perform an amortized analysis of the insertion operations. For each insertion operation, disregarding the cost of rebalancing, we assign a charge of $O\Big(\frac{\log_{M/B} d_i + \log_{M/B} y_i}{B}\Big)$, to the inserted element, where d_i is the number of elements currently in the tree greater than the i-th element at the time of insertion, and y_i is the number of elements that will be inserted after the i-th element that are smaller than the i-th element. [8] Summing this charge across all N inserted elements gives the total amortized cost

$$\sum_{i=1}^{N} \frac{\log_{\frac{M}{B}} d_i + \log_{\frac{M}{B}} y_i}{B} = O\left(\frac{N}{B}\left(1 + \log_{\frac{M}{B}} \frac{\mathrm{Inv}(X)}{N}\right)\right).$$

Since the above matches the bounds of the theorem, we seek to prove that the sweeping updates are covered by the initial charge paid for the start of the insertion operation.

The first and second phases of the sweeping update process, i.e., emptying the buffers up and down the tree, involve moving each element i along a path of length at most $O\Big(\log_{M/B} d_i\Big)$. Elements are transferred in blocks, and transfers are delayed until there are at least $M/2$ elements in the buffer to move. This batching ensures that a group of B elements are in aggregate moved in a single I/O. As a result, the cost of moving B element up or down the tree by a node is a constant, thus the initial charge is sufficient to cover the cost of the first two phases.

In the third phase, during the leaf node rebalancing, an element will only ever be pushed up the tree, and the distance it will be pushed up can be upper bounded by at most $O\Big(\log_{M/B}(d_i + y_i)\Big)$. Since elements are only pushed up when the node contains more than M elements, as in the previous two phases, moving a batch of B elements are moved in a single I/O, thus the initial charge is sufficient to cover the cost of each element being split and pushed up as well.

Therefore, the cost of moving an element from the leaf node to their final location in the tree is also covered by the initial charge. Similar to the first two phases, since the rebalancing operation is performed in a batch, the cost of moving each element is amortized over the B elements in the buffer. □

VIII. Pipelined Merging in an Organic Merge Tree

In this section, we describe the details for how to perform the pipelined merges in the CREW PRAM model, which provides the remaining details for the proof of Theorem 6. Our method uses constant-time CREW PRAM merging techniques

[8] $\sum y_i$ and $\sum d_i$ are equivalent, but we choose to separate them to make their purpose within the analysis clearer.

of Atallah, Cole, and Goodrich [4], which we include here for the sake of completeness. Given a sequence, $X = (x_1, x_2, \ldots, x_N)$, of comparable elements, let us assume we are given a set of maximal runs, $\mathcal{R} = (R_1, \ldots, R_{\rho(X)})$, and a merge tree, T, for $\mathcal{R}$ such that the total weight of T is $O(N(1 + \mathcal{H}(X)))$. Each run, R_i, is associated with a leaf, v_i, and for each node v in T, we also define S_v to be the set of all elements stored in descendants of v in T, including v itself. So $S_{v_i} = R_i$ if v_i is a leaf.

As described above, in the body, we have a formal notion for how we pass information from the merges performed at the children of v in stage s to the merge being performed at v in stage $s + 1$. Also, recall that until an internal node, v, becomes full $U'_{s+1}(v)$ will be the list consisting of every fourth element of $U_s(v)$. This continues to be true about $U'_{s+1}(v)$ up to the point that v becomes full. If s_v is the stage at which an internal node, v, becomes full (and $U_s(v) = U(v)$), then at stage $s_v + 1$, $U'_{s+1}(v)$ is the two-sample of $U_s(v)$, and, at stage $s_v + 2$, $U'_{s+1}(v) = U_s(v)$ $(= U(v))$. Thus, at stage $s_v + 3$, $parent(v)$ is full. Therefore, as noted above, after $3 \cdot h'(T)$ stages every node has become full and the algorithm terminates. Importantly, we also have the following.

Lemma 2. *For any stage $s \geq 0$ and any node $v \in T$, $|U_{s+1}(v)| \leq 2|U_s(v)| + 4$.*

Proof. The proof follows for internal nodes from a similar lemma (2.1) by Atallah, Cole, and Goodrich [4]. The proof follows for leaf nodes from the definition of $U_{s+1}(v)$. □

This gives us the following:

Lemma 3. *The list $(-\infty, U'_s(v), \infty)$ is a 4-cover for $U'_{s+1}(v)$, for all $s \geq 0$.*

Proof. The argument for internal nodes is the same as in the proof of Lemma 2.2 and Corollary 2.3 in [4]. The argument for leaves follows immediately from the definition of $U_{s+1}(v)$ for each leaf v in T. □

This, in turn, implies by an argument similar to that used to prove Theorem 2.5 in [4] that we can perform each stage s in $O(1)$ time and work that is equal to the total size of all $U_s(v)$ lists for nodes v in T. We include this argument below for the sake of completeness. The invariants we maintain at the start of each stage s is as follows:

1) For each item in $U'_s(v)$: its rank in $U'_s(sibling(v))$.
2) For each item in $U'_s(v)$: its rank in $U_s(v)$ (and hence, implicitly, its rank in $U'_{s+1}(v)$).

The lemma that follows shows that the above information is sufficient to allow us to merge $U'_{s+1}(lchild(v))$ and $U'_{s+1}(rchild(v))$ into the list $U_{s+1}(v)$ in $O(1)$ time using $|U_{s+1}(v)|$ processors.

Lemma 4. ***(The Merge Lemma) [16].*** *Suppose one is given sorted lists A_s, A'_{s+1}, B'_s, B'_{s+1}, C'_s, and C'_{s+1}, where the following (input) conditions are true:*

1) *$A_s = B'_s \cup C'_s$;*
2) *A'_{s+1} is a subset of A_s;*
3) *B'_s is a c_1-cover for B'_{s+1};*
4) *C'_s is a c_2-cover for C'_{s+1};*
5) *B'_s is ranked in B'_{s+1};*
6) *C'_s is ranked in C'_{s+1};*
7) *B'_s and C'_s are cross ranked.*

Then in $O(1)$ time using $|B'_{s+1}| + |C'_{s+1}|$ processors in the CREW PRAM model, one can compute the following (output computations):

1) *the sorted list $A_{s+1} = B'_{s+1} \cup C'_{s+1}$;*
2) *the ranking of A'_{s+1} in A_{s+1};*
3) *the cross ranking of B'_{s+1} and C'_{s+1}.*

We apply this lemma by setting $A_s = U_s(v)$, $A'_{s+1} = U'_{s+1}(v)$, $A_{s+1} = U_{s+1}(v)$, $B'_s = U'_s(x)$, $B'_{s+1} = U'_{s+1}(x)$, $C'_s = U'_s(y)$, and $C'_{s+1} = U'_{s+1}(y)$, where $x = lchild(v)$ and $y = rchild(v)$. Note that assigning the lists of Lemma 4 in this way satisfies input conditions 1–4 from definitions. The ranking information we maintain from stage to stage satisfies input conditions 5–7. Thus, in each stage s we can construct the list $U_{s+1}(v)$ in $O(1)$ time using $|U_{s+1}(v)|$ processors. Also, the new ranking information (of output computations 2 and 3) gives us the input conditions 5–7 for the next stage. Moreover, we have that the constants c_1 and c_2 (of input conditions 3 and 4) are both equal to 4. Note that in stage s it is only necessary to store the lists for $s - 1$; we can discard any lists for stages prior to that.

The method for performing all these merges with a total of $|T|$ processors is basically to start out with $O(1)$ virtual processors assigned to each leaf node, and each time we pass k elements from a node v to the parent of v (to perform the merge at the parent), we also pass $O(k)$ virtual processors to perform the merge. When v's parent becomes full, then we no longer "store" any processors at v. There can be at most $O(n)$ elements present in active nodes of T for any stage s (where n is the number of leaves of T), since there are n elements present on the full level, at most $n/2$ on the level above that, $n/8$ on the level above that, and so on. Thus, we can perform the entire generalized cascading procedure using work proportional to the weight of T. This gives us the following:

Theorem 11. *Suppose one is given a binary merge tree, T, such that there is a sorted set $U(v)$ stored at each leaf. Then one can compute, for each node $v \in T$, the list $U(v)$, which is the union of all items stored at descendants of v, sorted in an array. This computation can be implemented in $O(h'(T))$ time using a total of $O(w(T))$ work in the CREW PRAM model, where $w(T)$ is the weight of T.*

Proof. Because we roughly at most double the size of $U_s(v)$ at each stage s, the total work to perform all the merges and samples at a node v in T is $O(n_v)$. Thus, the total work for the entire computation is $O(w(T))$. The time bound follows from the fact that we implement each stage in $O(1)$ time. □

Therefore, after we are done, the sorted set is $U(r)$, where r is the root of T. This completes the proof of Theorem 6.

Acknowledgements

This research was supported in part by NSF Grant 2212129 and NSF Grant 2432018, and Independent Research Fund Denmark grant 9131-00113B.

References

[1] Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988. doi:10.1145/48529.48535.

[2] Lars Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, 2003. doi:10.1007/S00453-003-1021-X.

[3] Lars Arge, Mikael Knudsen, and Kirsten Larsen. A general lower bound on the I/O-complexity of comparison-based algorithms. In Frank Dehne, Jörg-Rüdiger Sack, Nicola Santoro, and Sue Whitesides, editors, *Symp. Algorithms and Data Structures (WADS)*, pages 83–94. Springer, 1993. doi:10.1007/3-540-57155-8_238.

[4] Mikhail J. Atallah, Richard Cole, and Michael T. Goodrich. Cascading divide-and-conquer: A technique for designing parallel algorithms. *SIAM Journal on Computing*, 18(3):499–532, 1989. doi:10.1137/0218035.

[5] Nicolas Auger, Vincent Jugé, Cyril Nicaud, and Carine Pivoteau. On the worst-case complexity of timsort, 2019. arXiv:1805.08612, doi:10.48550/arXiv.1805.08612.

[6] Jérémy Barbay and Gonzalo Navarro. On compressing permutations and adaptive sorting. *Theoretical Computer Science*, 513:109–123, 2013. doi:10.1016/J.TCS.2013.10.019.

[7] Guy E. Blelloch and Magdalen Dobson. The geometry of tree-based sorting. In Kousha Etessami, Uriel Feige, and Gabriele Puppis, editors, *Int. Colloq. on Automata, Languages, and Programming (ICALP)*, volume 261 of *LIPIcs*, pages 26:1–26:19. Schloss Dagstuhl, 2023. doi:10.4230/LIPIcs.ICALP.2023.26.

[8] Gerth Stølting Brodal and Rolf Fagerberg. Cache oblivious distribution sweeping. In Peter Widmayer, Francisco Triguero Ruiz, Rafael Morales Bueno, Matthew Hennessy, Stephan J. Eidenbenz, and Ricardo Conejo, editors, *Int. Symp. on Automata, Languages and Programming (ICALP)*, volume 2380 of *LNCS*, pages 426–438. Springer, 2002. doi:10.1007/3-540-45465-9_37.

[9] Gerth Stølting Brodal and Rolf Fagerberg. Funnel heap - A cache oblivious priority queue. In Prosenjit Bose and Pat Morin, editors, *Int. Symp. on Algorithms and Computation (ISAAC)*, volume 2518 of *LNCS*, pages 219–228. Springer, 2002. doi:10.1007/3-540-36136-7_20.

[10] Gerth Stølting Brodal and Rolf Fagerberg. On the limits of cache-obliviousness. In Lawrence L. Larmore and Michel X. Goemans, editors, *ACM Symposium on Theory of Computing (STOC)*, pages 307–315, 2003. doi:10.1145/780542.780589.

[11] Gerth Stølting Brodal, Rolf Fagerberg, and Gabriel Moruz. Cache-aware and cache-oblivious adaptive sorting. In Luís Caires, Giuseppe F. Italiano, Luís Monteiro, Catuscia Palamidessi, and Moti Yung, editors, *32nd Int. Colloq. Automata, Languages and Programming (ICALP)*, volume 3580 of *LNCS*, pages 576–588. Springer, 2005. doi:10.1007/11523468_47.

[12] Gerth Stølting Brodal and Sebastian Wild. Funnelselect: Cache-oblivious multiple selection. In Inge Li Gørtz, Martin Farach-Colton, Simon J. Puglisi, and Grzegorz Herman, editors, *European Symposium on Algorithms (ESA)*, LIPIcs, pages 25:1–25:17. Dagstuhl, 2023. doi:10.4230/LIPIcs.ESA.2023.25.

[13] Sam Buss and Alexander Knop. Strategies for stable merge sorting. In *Symposium on Discrete Algorithms (SODA)*, pages 1272–1290. SIAM, 2019. doi:10.1137/1.9781611975482.78.

[14] Svante Carlsson and Jingsen Chen. An optimal parallel adaptive sorting algorithm. *Information Processing Letters*, 39(4):195–200, 1991. doi:10.1016/0020-0190(91)90179-L.

[15] Jingsen Chen and Christos Levcopoulos. Improved parallel sorting of presorted sequences. In Luc Bougé, Michel Cosnard, Yves Robert, and Denis Trystram, editors, *Conf. on Vector and Parallel Processing (CONPAR)*, volume 634 of *LNCS*, pages 539–544. Springer, 1992. doi:10.1007/3-540-55895-0_452.

[16] Richard Cole. Parallel merge sort. *SIAM Journal on Computing*, 17(4):770–785, 1988. doi:10.1137/0217049.

[17] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 4th edition, 2022.

[18] Vladimir Estivill-Castro and Derick Wood. A survey of adaptive sorting algorithms. *ACM Computing Surveys*, 24(4):441–476, 1992. doi:10.1145/146370.146381.

[19] Arash Farzan, Paolo Ferragina, Gianni Franceschini, and J. Ian Munro. Cache-oblivious comparison-based algorithms on multisets. In Gerth Stølting Brodal and Stefano Leonardi, editors, *European Symp. on Algorithms (ESA)*, volume 3669 of *LNCS*, pages 305–316. Springer, 2005. doi:10.1007/11561071_29.

[20] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. *ACM Transactions on Algorithms*, 8(1):4:1–4:22, 2012. doi:10.1145/2071379.2071383.

[21] William Cawley Gelling, Markus E. Nebel, Benjamin Smith, and Sebastian Wild. Multiway powersort. In *Symposium on Algorithm Engineering and Experiments (ALENEX)*, pages 190–200. SIAM, 2023. doi:10.1137/1.9781611977561.ch16.

[22] Michael T. Goodrich and S. Rao Kosaraju. Sorting on a parallel pointer machine with applications to set expression evaluation. *Journal of the ACM*, 43(2):331–361, 1996. doi:10.1145/226643.226670.

[23] Scott Huddleston and Kurt Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17:157–184, 1982. doi:10.1007/BF00288968.

[24] Joseph JáJá. *Parallel Algorithms*. Addison-Wesley, 1992.

[25] Vincent Jugé. Adaptive shivers sort: An alternative sorting algorithm. *ACM Transaction on Algorithms*, 20(4), August 2024. doi:10.1145/3664195.

[26] Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.

[27] C. P. Kruskal. Searching, merging, and sorting in parallel computation. *IEEE Transactions on Computers*, 32(10):942–946, 1983. doi:10.1109/TC.1983.1676138.

[28] De-lei Lee and Kenneth E. Batcher. On sorting multiple bitonic sequences. In *International Conference on Parallel Processing (ICPP)*, volume 1, pages 121–125, 1994. doi:10.1109/ICPP.1994.137.

[29] Christos Levcopoulos and Ola Petersson. A note on adaptive parallel sorting. *Information Processing Letters*, 33(4):187–191, 1989. doi:10.1016/0020-0190(89)90139-7.

[30] Christos Levcopoulos and Ola Petersson. Exploiting few inversions when sorting: Sequential and parallel algorithms. *Theoretical Computer Science*, 163(1):211–238, 1996. doi:10.1016/0304-3975(95)00256-1.

[31] Heikki Mannila. Measures of presortedness and optimal sorting algorithms. In Jan Paredaens, editor, *Int. Conf. Automata, Languages and Programming (ICALP)*, volume 172 of *Lecture Notes in Computer Science*, pages 324–336. Springer, 1984. doi:10.1007/3-540-13345-3_29.

[32] Kurt Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*, volume 1 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1984. doi:10.1007/978-3-642-69672-5.

[33] J. Ian Munro and Sebastian Wild. Nearly-optimal mergesorts: Fast, practical sorting methods that optimally adapt to existing runs. In Yossi Azar, Hannah Bast, and Grzegorz Herman, editors, *European Symposium on Algorithms (ESA)*, volume 112 of *LIPIcs*, pages 63:1–63:16. Dagstuhl, 2018. doi:10.4230/LIPICS.ESA.2018.63.

[34] Ola Petersson and Alistair Moffat. A framework for adaptive sorting. *Discrete Applied Mathematics*, 59(2):153–179, 1995. doi:10.1016/0166-218X(93)E0160-Z.

[35] Jens Kristian Refsgaard Schou and Bei Wang. PersiSort: a new perspective on adaptive sorting based on persistence. In *Canadian Conference on Computational Geometry (CCCG)*, pages 287–312, 2024. URL: https://cccg.ca/proceedings/2024/CCCG_2024_proceedings.pdf.

[36] Tadao Takaoka. Partial solution and entropy. In Rastislav Královic and Damian Niwinski, editors, *Mathematical Foundations of Computer Science (MFCS)*, volume 5734 of *LNCS*, pages 700–711. Springer, 2009. doi:10.1007/978-3-642-03816-7_59.