Beyond binary search: parallel in-place construction of implicit search tree layouts

Kyle Berney, Henri Casanova, Alyssa Higuchi, Ben Karsin, and Nodari Sitchinava
Department of Information and Computer Sciences
University of Hawaii at Manoa
Honolulu, Hawaii
Email: {berneyk, henric, higuchi8, karsin, nodari}@hawaii.edu

***Abstract*—We present parallel algorithms to efficiently permute a sorted array into the level-order binary search tree (BST), level-order B-tree (B-tree), and van Emde Boas (vEB) layouts *in-place*. We analytically determine the complexity of our algorithms and empirically measure their performance. Results indicate that on both CPU and GPU architectures B-tree layouts provide the best query performance. However, when considering the total time to permute the data and to perform a series of search queries, our vEB permutation provides the best performance on the CPU. We show that, given an input of N=500M 64-bit integers, the benefits of query performance (compared to binary search) outweigh the cost of in-place permutation using our algorithms when performing at least 5M queries (1% of N) and 27M queries (6% of N), on our CPU and GPU platforms, respectively.**

***Index Terms*—permutation, searching, parallel, in-place**

I. Introduction

Searching is a fundamental computational problem and various pointer-based data structures have been proposed to optimize search queries. Binary search trees (BSTs) are the simplest such structures for one-dimensional data. On modern architectures with a multi-level memory hierarchy, I/O-efficient variations of search trees, e.g. B-trees, typically outperform BSTs due to their improved locality of reference.

A drawback of pointer-based search tree data structures is that they take up a constant factor more space than the data itself. In contrast, if the data is stored in sorted order, efficient search can be performed using binary search without using any extra space. The advantage of search trees lies in their efficient updates (insertions and deletions of elements). However, in the case of *static* data, storing data in sorted order and performing binary search seems to be the preferred approach. For example, Khuong and Morin [23] observe that binary searches account for 10% of the computation time for the AppNexus ad-bidding engine; and searches on static data arise in various domains such as finance [5], sales [20], advertising [23], and numerical analysis [8].

Despite its simplicity and optimal time complexity in the classical RAM model, binary search is not the most efficient approach on modern memory hierarchies. This is easily seen by viewing a sorted array as an *implicit* binary search tree. In an *implicit tree*, data is stored in an array and the locations within the array define a one-to-one mapping to the vertices of the corresponding (pointer-based) tree so that child-parent relationships of the vertices is determined via index arithmetic. For example, the i-th entry of a sorted array of size N corresponds to the i-th vertex visited during the in-order traversal of a complete BST on N vertices. Similarly, entries accessed during a binary search for key x in a sorted array correspond to the keys stored in the vertices on the root-to-leaf path when searching for x in the corresponding BST. Since queries on B-trees are more cache-efficient than on BSTs, it is not surprising that querying data stored in an implicit B-tree layout also outperforms binary search both in theory and practice on modern architectures [7], [23], [25].

Given the abundance of available parallelism in modern CPUs and GPUs, in this paper we study efficient *parallel* transformations of a static sorted array into various *implicit* search tree layouts (defined in Section I-A). Moreover, since binary search on already sorted data does not require any additional space, we are interested in performing these transformations *in-place*.

A. Memory Layouts of Static Search Trees

We say a layout is a *BST layout* if it is defined by the breadth-first left-to-right traversal of a complete binary search tree.[1] Given the index i of a node v in the BST layout, the indices of the left and right children of v can be computed in $O(1)$ time as $2i+1$ and $2i+2$ (using 0-indexing), respectively.

A *complete B-tree* [4] is a complete multi-way search tree, where each node (except possibly the last leaf node) contains exactly B elements and every internal node (except possibly the last one) has exactly $B+1$ children. The *B-tree layout* is defined by the breadth-first left-to-right traversal of a complete B-tree.

The *van Emde Boas (vEB) layout* [28] is defined recursively as follows. The vEB layout of a tree with a single vertex is the vertex itself. Given a complete binary search tree $\mathcal{T}$ with N vertices and height $h = \lfloor \log N \rfloor > 0$, consider the top subtree $\mathcal{T}_0$ of height $\lceil (h-1)/2 \rceil$ containing $r = 2^{\lceil (h-1)/2 \rceil} - 1$ vertices, and up to $r+1$ bottom subtrees $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_{r+1}$, each of height $\lfloor (h-1)/2 \rfloor$ and each rooted at the children of the $(r+1)/2$ leaves of $\mathcal{T}_0$. The vEB layout of $\mathcal{T}$ is defined

This material is based upon work supported by the National Science Foundation under Grant No. 1533823.

[1] In a complete binary tree every level, except possibly the last one, is completely filled and all leaves on the last level are as far left as possible.

recursively as the vEB layout of $\mathcal{T}_0$, followed by the vEB layouts of each $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_{r+1}$.

The I/O complexity (defined in Section II-A) of performing a search query on an array of size N in the BST layout is $O(\log(N/B))$, and $\Theta(\log_B N)$ in the B-tree and vEB layouts [7], [28]. In theory, because the definition of the vEB layout does not make use of the parameter B, i.e., it is *cache-oblivious* [19], querying the vEB layout on architectures with multiple levels of cache will result in the asymptotically optimal number of accesses at every level of the memory hierarchy [28].

The relative performance of querying each of these search tree layouts has been studied empirically. Ladner *et al.* [25] measure the cache performance and instruction count of querying the B-tree and vEB layouts, with results indicating that the B-tree layout achieves the best performance on CPUs. The experimental results of Brodal *et al.* [7] indicate that the performance of the vEB and B-tree layouts are comparable, both outperforming the BST layout. These results are contradicted, however, by Khuong and Morin [23], who show that, by using explicit prefetching and other optimizations, the BST layout can outperform both the B-tree and vEB layouts.

B. Previous Work on Permutations

The transformation from sorted order to an implicit search tree layout is a special case of permuting an array of N elements. Let $\pi : [N] \to [N]$ be an arbitrary permutation. For the purpose of this paper, we assume that π is given as a function that can be described concisely in $O(1)$ space (e.g., not as a table that explicitly gives $\pi(i)$ for each i). Let τ_π be the time it takes to evaluate $\pi(i)$. For example, while $\tau_\pi = O(1)$ for the BST and B-tree layouts, it is not obvious how to compute τ_π faster than $O(\log\log N)$ time for the vEB layout.

Note that for the problem of permuting N elements using P processors, $\Omega((N/P) \cdot \tau_\pi)$ is the trivial lower bound. If there is no in-place requirement, any permutation π can be implemented in $O(\lceil N/P \rceil \cdot \tau_\pi)$ time in parallel: each entry $A[i]$ can be copied to $B[\pi(i)]$ independently of each other. Thus, the BST and B-tree layouts can be constructed from sorted data in $O(\lceil N/P \rceil)$ time and the vEB layout can be constructed in $O(\lceil N/P \rceil \log\log N)$ time.

It is well known that every permutation can be decomposed into disjoint cycles. A cyclic permutation can be implemented sequentially in-place trivially by starting at a single vertex and following the cycle. However, for a general permutation this approach still needs additional space to mark the elements that have already been permuted, unless it can identify all cycles up front.

When it comes to *in-place* permutations, Fich *et al.* [18] showed that every permutation π can be implemented sequentially in-place in $O((N \log N) \cdot \tau_\pi)$ time. For a special case when the data is permuted from a sorted order, they observed that they can check if an element has already been moved by computing the inverse permutation π^{-1} to determine if the element is not in its original sorted order. Thus, for this special case, the time can be reduced to $O(N \cdot (\tau_\pi + \tau_{\pi^{-1}}))$. However, it is not obvious how to parallelize their algorithm, nor is it trivial to compute π^{-1} for the vEB layout.

Yang *et al.* [33] observed that every permutation is the product of two involutions. A permutation π is an *involution* if it is its own inverse, i.e., $\pi(\pi(i)) = i$ for all i. Moreover, every involution is composed of disjoint cycles of length at most 2, i.e., can be implemented in parallel and in-place by swapping pairs of elements. Thus, if the two involutions of a permutation are known, this permutation can be implemented in parallel and in-place. This result is non-constructive, i.e., given an arbitrary permutation π it is not clear how to determine the two involutions that define π; however, the authors show how to determine the involutions of a cyclic permutation.

One permutation of particular interest for this work is the *perfect shuffle* [13]: a permutation in which two lists of equal length are interleaved perfectly. A generalization is the *k-way perfect shuffle*, where k equal-length lists are interleaved perfectly [29]. These permutations have many applications (e.g., parallel processing [30], Fast Fourier Transforms (FFT) [11], [30], Kronecker products [11], [12], encryption [31], sorting [30], and merging [10], [14], [17]). Ellis *et al.* [15], [16] use a number-theoretic approach to compute representative elements of the disjoint cycles of the perfect shuffle and the k-way perfect shuffle, thus making a sequential in-place approach possible. Jain [21] relies on the fact that 2 is primitive root of 3^k for any $k \geq 1$, which makes it possible to compute the representative elements of the disjoint cycles recursively for any N. Finally, Yang *et al.* [33] use the product of involutions approach and describe the involutions for the k-way perfect shuffle for two cases: (i) $N = k^d$ and (ii) $N = kd$ for some integer $d > 1$. For (i), the involutions involve reversing the base-k representation of element indices. For (ii), the involutions involve computing modular inverses of element indices and finding greatest common divisors. We use these results of Yang *et al.* [33] for designing our involution-based permutation algorithms.

C. Our Results

We present parallel algorithms for the in-place permutation of a sorted array into the BST, B-tree, and vEB layouts, and analyze their time and I/O complexities. We propose two types of algorithms:

1) Building on the work of Yang *et al.* [33] and Fich *et al.* [18], we determine the pairs of *involutions* required to permute a sorted array into the BST layout. We also determine the $\log_{B+1} N$ pairs of involutions required to permute a sorted array into the B-tree layout. We can use the B-tree involutions to permute a sorted array into the vEB layout.
2) Using a *cycle-leader* approach, we develop an efficient parallel in-place algorithm to permute a sorted array into the vEB layout. By recursively applying this approach, we are able to design algorithms for permuting a sorted array into the B-tree layout. The B-tree layout algorithm can be used to obtain the BST layout by setting $B = 1$.

TABLE I

ASYMPTOTIC RUNTIME AND I/O COMPLEXITY BOUNDS OF EACH OF OUR ALGORITHMS. N IS THE INPUT SIZE, P IS THE NUMBER OF PROCESSORS, M AND B ARE THE SIZES OF THE INTERNAL MEMORY AND THE TRANSFER BLOCK, RESPECTIVELY, IN THE PARALLEL EM (PEM) MODEL. $K = \text{MIN}(\frac{N}{P}, M)$ AND $T_{REV_k}(N)$ IS THE TIME COMPLEXITY OF REVERSING THE DIGITS OF NUMBER N IN THE BASE-k REPRESENTATION.

Algorithm	Time complexity	I/O complexity
Involution BST	$O\left(\frac{N}{P} \cdot T_{REV_2}(N)\right)$	$O\left(\frac{N}{P}\right)$
Involution B-tree	$O\left(\left(\frac{N}{P} + \log_{B+1} N\right) \log N\right)$	$O\left(\frac{N}{P} + B \log_{B+1} \frac{N}{K}\right)$
Involution vEB	$O\left(\frac{N}{P} \log N\right)$	$O\left(\frac{N}{P} \log \log_K N\right)$
Cycle-leader BST	$O\left(\left(\frac{N}{P} + \log N\right) \log N\right)$	$O\left(\left(\frac{N}{PB} + \log \frac{N}{K}\right) \log \frac{N}{K}\right)$
Cycle-leader B-tree	$O\left(\left(\frac{N}{P} + \log_{B+1} N\right) \log_{B+1} N\right)$	$O\left(\left(\frac{N}{PB} + \log_{B+1} \frac{N}{K}\right) \log_{B+1} \frac{N}{K}\right)$
Cycle-leader vEB	$O\left(\frac{N}{P} \log \log N\right)$	$O\left(\frac{N}{PB} \log \log_K N\right)$

The involution-based approach entails reversing a subset of the digits of numbers represented in an arbitrary base-k (for BST $k = 2$, for B-tree $k = B + 1$). If implemented in software, the worst-case complexity of this operation is linear with the number of digits, i.e., $O(\log_k N)$. However, some architectures[2] provide it as a built-in hardware primitive, i.e., it takes $O(1)$ time. Therefore, we parameterize the time of this operation as $T_{REV_k}(N)$.

To the best of our knowledge our algorithms are the first parallel in-place algorithms for permuting a sorted array into search tree layouts. We analyze the time and I/O complexities of our algorithms, which are summarized in Table I. Our cycle-leader algorithms exhibit better spatial locality, while our involution-based algorithms are much simpler and trivial to parallelize.

We evaluate these algorithms experimentally on multicore CPU and GPU architectures. We find that, compared to a binary search on non-permuted input, the permutation overhead of our permutation algorithms is offset by the query time for as few as $0.01N$ search queries.

The remainder of this paper is organized as follows. Section II provides background on parallel models of computation. Section III, resp. Section IV, presents our involution-based, resp. cycle-leader, algorithms. For ease of exposition, in Sections III and IV we consider only *perfect* trees, i.e., complete trees in which every level is full. Section V analyzes the I/O complexity of our algorithms. Section VI discusses extensions of our algorithms to non-perfect trees. Section VII presents experimental results. Finally, Section VIII concludes with a summary and perspectives on future work.

II. PRELIMINARIES

A. Models of Computation

In this work, we analyze the running time of our parallel algorithms in the *Parallel Random Access Machine (PRAM)* model [22]. Given an input of size N, the PRAM model defines two complexity metrics of an algorithm: *work*, $W(N)$, is the number of total operations performed by all processors, and *depth* (also known as *span* or *critical path length*), $D(N)$, is the maximum number of operations performed by any one processor if the algorithm is executed using an infinite number of processors. Then, the runtime of an algorithm on P processors can be computed using Brent's Scheduling Principle [6] as $T(N, P) = O\left(\frac{W(N)}{P} + D(N)\right)$. In this work, we consider the CREW PRAM model, which allows simultaneous read accesses, but disallows simultaneous write accesses to the same data by multiple processors.

We analyze the I/O complexity of our parallel algorithms in the Parallel External Memory (PEM) model [3] – a parallel extension of the (sequential) *External Memory (EM)* model [1]. In the EM model, a processor contains fast internal memory of size M and data initially resides in (much larger) external memory. To process data, that data must first be transferred into internal memory using contiguous blocks of size B. The complexity metric of the EM model, *I/O complexity*, is the number of such blocks transferred during computation. The EM model has also been used to model a single level of cache in modern processor design: external memory represents main memory (DRAM), internal memory represents cache, transfer blocks represent cache-lines, and I/O complexity of an algorithm represents the number of accesses to slow DRAM. In practice, it has been shown that I/O-efficient algorithms outperform efficient RAM algorithms [2], [9]. In the PEM model, each of P processors contains private memory of size M and share the external memory. The data is still transferred between external memory and an individual processor's internal memory in blocks of size B. The (parallel) I/O complexity, $Q(N, P)$, is defined as the maximum number of blocks transferred by any one of the processors throughout the computation.

B. Parallel In-place Computations

There is a bit of ambiguity in the literature when it comes to the definition of *in-place* algorithms. Strictly speaking, a (sequential) algorithm is said to be *in-place* if it uses at most $\Theta(1)$ additional space (a processor needs at least one register to perform any useful work) [17]. However, for a recursive algorithm, at least $\Omega(\log N)$ additional space is needed to implement the recursion stack of a balanced recursion. Therefore, it is reasonable for an in-place algorithm to use up to $O(\log N)$ additional space, although often such algorithms are called *in-situ* [14], [24]. When it comes to parallel algorithms,

[2]E.g., NVidia K40 GPU implements this operation in hardware for $k = 2$.

there is an additional complication. Each of the P processors needs to have $\Omega(1)$ space to perform any meaningful work. Moreover, for asynchronous recursion, $\Omega(\log N)$ space is needed per processor, i.e., a total of $\Omega(P \log N)$ additional space. Therefore, if $P = \frac{N}{\log N}$, the total additional space becomes $\Omega(N)$ and trivially non-in-place algorithms could be viewed as being in-place. To avoid this situation, we define *in-place parallel* computation as follows:

Definition 1. *A parallel algorithm running on P processors each having an internal memory of size M is called* in-place *if it uses at most $O(P(M + \log N))$ additional space and works correctly for any $P \geq 1$ processors.*

In the PRAM model, $M = O(1)$ is the number of registers per processor so it reduces to $O(P \log N)$; while in the PEM model, M is the size of each processor's internal memory. Note that the requirement for an algorithm to work correctly for any $P \geq 1$ precludes the view of trivially non-in-place algorithms designed for large P as being in-place.

III. Involution Approach

A. BST Layout

A perfect BST contains $N = 2^d - 1$ vertices. As mentioned in Section I, Fich *et al.* [18] propose a sequential in-place algorithm to permute a sorted array into the BST layout. They note that the permutation satisfies the property that for a given index $i = (x10^j)_2$ in binary representation, the index of that element in the BST layout is $\pi(i) = (0^j 1x)_2$. Let $\text{REV}_k(b, i)$ be the operation that reverses b least significant digits of the base-k representation of the integer i. The previously mentioned permutation can be computed as $\pi(i) = \text{REV}_2(d - (j+1), (\text{REV}_2(d, i))$. Since REV_2 is an involution [18], we can perform the permutation π in parallel in just two rounds of $O(N)$ independent swaps.

The time to compute $\pi(i)$ depends on the time to perform the REV_2 operation, which we quantify precisely for our particular hardware platforms in Section VII. Thus, this algorithm has depth $D(N) = O(T_{\text{REV}_2}(N))$ and work $W(N) = O(N \cdot T_{\text{REV}_2}(N))$.

B. B-tree Layout

The B-tree layout algorithm relies on the k-way perfect shuffle involution approach developed by Yang *et al.* [33]. Let us first review their results.

Let $J_r(i) = g(r(\frac{i}{g})^{-1} \pmod{\frac{N-1}{g}})$ where g is the greatest common divisor of i and $N-1$. Yang *et al.* [33] show that for $N = k^d$ and $N = kd$ the k-way perfect shuffle can be implemented as $\Xi_1(i) = \text{REV}_k(d, \text{REV}_k(d-1, i))$ and $\Xi_2(i) = J_k(J_1(i))$, respectively, for any integer $d > 1$. We note that the k-way "un-shuffle," which we use, can be performed by simply reversing the order in which the involutions are performed.

A perfect B-tree has $N = (B+1)^d - 1$ elements, for some $d > 1$. Since each leaf node contains B contiguous elements from the sorted array, every $(B+1)$-th element is stored in a non-leaf (internal) node. Let S_i, for $i \in \{0, 1, 2, ..., B\}$, denote the list of elements at locations $i + j(B+1)$, for $j \in \{0, 1, ..., \left\lfloor \frac{N}{(B+1)} \right\rfloor\}$. In other words, each S_i is comprised of the elements starting at i, strided by $B+1$. By this definition, S_B contains all internal elements and S_l, for $0 \leq l \leq B-1$, contains the l-th element of each leaf node. We first perform the $(B+1)$-way perfect un-shuffle (via Ξ_1 while using or simulating 1-indexing), which will gather each S_i into contiguous space and lay them out in sequence. We then apply the B-way perfect shuffle (via Ξ_2 while using or simulating 0-indexing) on all S_l lists to interleave the leaf elements back into their corresponding leaf nodes, i.e., into their correct positions. All leaf elements are thus correctly permuted and we recurse on S_B.

Recall that REV_k can take up to $O(\log_k N)$ time. Finding the modular inverse, however, requires using the extended Euclidean algorithm [33], which takes $O(\log N)$ time. The latter dominates the running time, resulting in $O(\log N)$ time for both operations. Therefore, the work and depth complexities of our B-tree permutation algorithm are:

$$\begin{aligned} W(N) &= W\left(\frac{N}{B+1}\right) + O(N \log N) = O(N \log N) \ , \\ D(N) &= D\left(\frac{N}{B+1}\right) + O(\log N) = O(\log_{B+1} N \cdot \log N) \ . \end{aligned}$$

C. van Emde Boas Layout

We are able to apply the B-tree layout algorithm for the vEB layout, by using $B = 2^x - 1$ for $N = 2^{2x} - 1$ (trees of odd height) and $B = 2^{x-1} - 1$ for $N = 2^{2x-1} - 1$ (trees of even height) and by recursing on each subtree of the vEB layout. The resulting work and depth complexities are:

$$\begin{aligned} W(N) &= \sqrt{N} \cdot W\left(\sqrt{N}\right) + O(N \log N) = O(N \log N) \ , \\ D(N) &= D\left(\sqrt{N}\right) + O(\log N) = O(\log N) \ . \end{aligned}$$

IV. Cycle-leader Approach

A. van Emde Boas Layout

Recall from Section I-A that we define $\mathcal{T}_i$ as the i-th subtree of size $O(\sqrt{N})$: $\mathcal{T}_0$ is the "root" subtree consisting of $r = 2^{\lceil (h-1)/2 \rceil} - 1$ vertices of the upper $\left\lceil \frac{h-1}{2} \right\rceil$ levels, where $h = \lfloor \log N \rfloor$, while $\mathcal{T}_1, \ldots, T_{r+1}$ are "leaf" subtrees consisting of $l = 2^{\lfloor (h-1)/2 \rfloor} - 1$ vertices each. Let $A[a_i : b_i]$ be the interval within the input array where the elements of $\mathcal{T}_i$ should be moved to. In particular, $a_0 = 1$, $b_0 = r$ and for all $1 \leq j \leq r+1$, $a_j = r + (j-1)l + 1$ and $b_j = r + jl$. Our algorithm first moves each $\mathcal{T}_i$ into $A[a_i : b_i]$, which we call the *equidistant gather* operation, then recursively permutes each $A[a_i : b_i]$ into the vEB layout. (Our equidistant gather operation happens to be general enough to work for any $r \leq l$.)

We use $T_i[a : b]$ to denote the subset of nodes of $\mathcal{T}_i$ from the a-th smallest to the b-th smallest in the sorted order. E.g., $\mathcal{T}_i[1 : k]$ represents the first k smallest elements of $\mathcal{T}_i$.

The following proposition bounds the range in the input array, where the elements of the leaf subtrees T_j, for $j \geq 1$, are initially located:

Proposition 2. *For all $i = r - j + 2$, $1 \leq j \leq r$, $\mathcal{T}_j[i : l]$ are already in their destination interval $A[a_j : b_j]$. If $i > l$, then no elements of $\mathcal{T}_j$ are in their destination interval.*

Proof. Since the input is in sorted order, for all $1 \leq i, j \leq l$, $\mathcal{T}_j[i]$ is initially located at index $i_{orig} = (j-1)(l+1)+i$. Hence, we check if $i_{orig} \geq a_j = r+(j-1)l+1$. Solving for i results in $i \geq r-j+2$. □

From the above proposition, we know that $\mathcal{T}_1[r+1 : l], \mathcal{T}_2[r : l], ..., \mathcal{T}_{r+1}[1 : l]$ are already in their destination intervals and only $\mathcal{T}_1[1 : r], ..., \mathcal{T}_r[1]$ need to be moved.

We first consider a sequential strategy to perform the equidistant gather in-place: we perform r rounds of swapping, where after round i, all elements in the subtree $\mathcal{T}_i$ are in $A[a_i : b_i]$. Figure 1 illustrates the first few rounds of swapping for $r = l$. We see that, initially, all elements of $\mathcal{T}_0$ are distributed throughout the array. After the first round, $\mathcal{T}_0$ is in $A[a_0 : b_0]$ and $\mathcal{T}_1$ becomes distributed throughout the array. After repeating this process r times, each $\mathcal{T}_i$ is in $A[a_i : b_i]$, however, the elements in each $\mathcal{T}_i$ may not be in sorted order. Specifically, we need to perform a circular shift to the right by $r+1-i$ places (or equivalently $l-(r+1-i)$ to the left) on each $\mathcal{T}_i$.

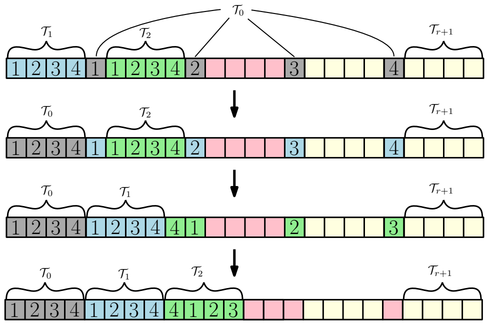


Fig. 1. Illustration of the series of swaps needed to sequentially perform the equidistant gather operation for $r = l$.

We can parallelize this algorithm by unrolling the r sequential swap rounds and identifying the resulting disjoint cycles. We identify r disjoint cycles of the following form:

$$\begin{aligned}
&\mathcal{T}_0[1] \mapsto \mathcal{T}_1[1] , \\
&\mathcal{T}_0[2] \mapsto \mathcal{T}_1[2] \mapsto \mathcal{T}_2[1] , \\
&\mathcal{T}_0[3] \mapsto \mathcal{T}_1[3] \mapsto \mathcal{T}_2[2] \mapsto \mathcal{T}_3[1] , \\
&\quad\vdots \\
&\mathcal{T}_0[r] \mapsto \mathcal{T}_1[r] \mapsto ... \mapsto \mathcal{T}_{r-1}[2] \mapsto \mathcal{T}_r[1] .
\end{aligned}$$

Since we can identify each element in each disjoint cycle, its position in the cycle, and the length of the cycle, as mentioned in Section I, we can implement circular shifts in parallel and in-place in $O(1)$ depth and $O(N)$ work using the involutions of Yang *et al.* [33]. Therefore, since both stages of our algorithm are comprised of disjoint circular shifts, we can perform the equidistant gather in $O(1)$ time and $O(N)$ work.

Depending on the height of the given tree, the value of r and l will vary in the vEB layout. For $N = 2^{2x}-1$, we have $r = l = 2^x - 1$, hence we can apply the above equidistant gather to permute each subtree into their correct interval, then we recurse on each subtree in parallel. For $N = 2^{2x-1}-1$, we have $r = 2^x - 1$ and $l = 2^{x-1}-1$. Since $r = 2l+1$, we split the array in half and perform the above algorithm separately on $A[1 : \lfloor \frac{N}{2} \rfloor]$ and $A[\lfloor \frac{N}{2} \rfloor + 2 : N]$ in parallel. We then combine these two partitions by performing a circular shift on $A[l+1 : \lfloor \frac{N}{2} \rfloor + l + 1]$ to shift the $(l+1)$ elements of $\mathcal{T}_0$ in the second partition to the front. The work and depth complexities of this algorithm are:

$$\begin{aligned}
W(N) &= \sqrt{N} \cdot W(\sqrt{N}) + O(N) = O(N \log\log N) , \\
D(N) &= D(\sqrt{N}) + O(1) = O(\log\log N) .
\end{aligned}$$

B. B-tree Layout

The idea is similar to the above vEB cycle-leader approach, except we have $r = \left\lfloor \frac{N}{(B+1)} \right\rfloor$ and $l = B$. Therefore, we need to extend the equidistant gather operation for $r > l$. We call this version the *extended equidistant gather* operation.

In a perfect B-tree of height h, $N = (B+1)^{h+1} - 1$. Let $C = \left\lceil \frac{N}{(B+1)^2} \right\rceil$. To perform the extended equidistant gather, we partition the array into $(B+1)$ partitions, where each partition will contain C internal elements (except for the first one, which will contain $C-1$ internal elements) and BC leaf elements. We move the internal elements of each partition to the front of that partition by applying the extended equidistant gather recursively on each partition. We then move the internal elements to the front of the whole array by applying the equidistant gather while treating each chunk of C elements as a unit, and while ignoring the first $C-1$ internal elements of the first partition. At the base case of the recursion $C = 1$ and we can apply the equidistant gather directly to bring the internal elements to the front.

Since the equidistant gather takes $O(N)$ work and $O(1)$ depth, the extended equidistant gather takes:

$$W'(N) = (B+1) \cdot W'\left(\frac{N}{B+1}\right) + O(N) = O(N \log_{B+1} N) ,$$

$$D'(N) = D'\left(\frac{N}{B+1}\right) + O(1) = O(\log_{B+1} N) .$$

Once all the internal elements are gathered to the front of the array, we recursive on the internal elements, resulting in the following complexities:

$$W(N) = W\left(\frac{N}{B+1}\right) + W'(N) = O(N \log_{B+1} N) ,$$

$$D(N) = D\left(\frac{N}{B+1}\right) + D'(N) = O(\log^2_{B+1} N) .$$

C. BST Layout

We can apply the algorithm in the previous section to the BST layout by setting $B = 1$, resulting in $O(N \log N)$ work and $O(\log^2 N)$ depth. Although this is worse than the involution-based algorithm from Section III, the cycle-leader algorithm exhibits better spatial locality, which we analyze in the next section.

V. I/O OPTIMIZATIONS

In this section, we analyze the I/O complexity of our proposed algorithms in the *parallel external memory (PEM)* model [3] – a parallel extension of the EM model. When applicable, we present additional modifications to the algorithms to improve the I/O efficiency. Let $K = \text{MIN}\left(\frac{N}{P}, M\right)$ and assume that $P \leq \frac{N}{B}$, i.e., each processor processes at least one block, and $M \geq 2B + O(1)$, i.e., each processor can swap at least two blocks. In Section V-B, we strengthen this assumption to $M \geq B^2$ (standard tall-cache assumption), which is needed for matrix transposition, and consequently $P \leq \frac{N}{B^2}$.

A. *Involution-based Algorithms*

We first consider the involution-based algorithms described in Section III. The swaps performed by these algorithms can be an arbitrary distance away from each other. Hence, in the worst case these algorithms will perform $O(1)$ I/Os per swap. Thus, each iteration of an involution performs $O(\frac{N}{P})$ I/Os. For the B-tree and vEB layouts, however, once the subproblem is of size M or less, it will fit in internal memory. Therefore, for B-trees we have:

$$\begin{aligned} Q(N,P) &= \begin{cases} O\left(N/B\right) & \text{if } N \leq M \text{ and } P = 1 \\ Q\left(\frac{N}{B+1}, \text{MIN}\left(P, \frac{N}{B(B+1)}\right)\right) + O\left(\frac{N}{P}\right) & \text{otherwise} \end{cases} \\ &= O\left(\frac{N}{P} + B \log_{B+1} \frac{N}{K}\right) , \end{aligned}$$

and for vEBs we have:

$$\begin{aligned} Q(N,P) &= \begin{cases} O\left(N/B\right) & \text{if } N \leq M \text{ and } P = 1 \\ \left\lceil \frac{\sqrt{N}}{P} \right\rceil Q\left(\sqrt{N}, \left\lceil \frac{P}{\sqrt{N}} \right\rceil\right) + O\left(\frac{N}{P}\right) & \text{otherwise} \end{cases} \\ &= O\left(\frac{N}{P} \log\log_K N\right) . \end{aligned}$$

B. *vEB Cycle-leader Algorithm*

For the cycle-leader approach, we rely on performing parallel circular shifts of elements. We perform the circular shifts using the technique presented by Yang *et al.* [33], which involves two rounds of array reversals. We can reverse k elements in-place and in parallel by performing $\left\lfloor \frac{k}{2} \right\rfloor$ independent swaps. Specifically, index i swaps with index $k - i - 1$ (using 0-indexing). Thus, to optimize for I/Os, we can swap elements in groups of B, provided that every group of B elements are located in contiguous memory locations. Therefore, we can perform a circular shift of N elements in $O(\frac{N}{PB})$ I/Os. For the remainder of the section, assume every circular shift uses this optimization when applicable.

The vEB cycle-leader approach, described in Section IV-A, employs the equidistant gather operation, which relies on circular shifts. However, the equidistant gather performs a circular shift on elements strided by distance $O(\sqrt{N})$ and are thus not in contiguous memory. To avoid the I/O inefficiency of such an access pattern, we propose an initial transposition phase to block elements in each disjoint cycle together.

We can view the sorted array as a $(r+1) \times (r+1)$ row-major matrix with the bottom-right element removed. We can ignore the last row, which contains $\mathcal{T}_{r+1}$, since these elements do not move during the cycles. We also ignore the last column of the matrix, which contains the r elements of $\mathcal{T}_0$. Thus, we consider a square matrix of size $r \times r$. Figure 2 illustrates this representation, how each subtree is contained therein, and what elements are contained in each disjoint cycle of the gather. To improve I/O efficiency, we perform a circular shift on each

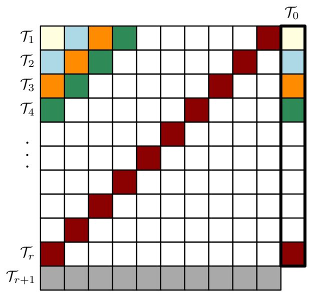


Fig. 2. Illustration of the distinct cycles of the equidistant gather operation when considering memory as a $r \times r$ matrix. Shifting each row and transposing the matrix lets us perform each cycle I/O efficiently.

row i by i positions to the right, which aligns the elements in each disjoint cycle into columns. We then transpose the square matrix to align the elements in each cycle into rows, placing all elements of each cycle into contiguous memory.

Shifting r rows of r elements requires $O(\frac{r^2}{PB})$ I/Os. Assuming that $P \leq N/B^2$ and $M \geq B^2$, we can perform matrix transposition in $O(\frac{r^2}{PB})$ I/Os by tiling the matrix into submatrices of size $B \times B$ [1], [32].

With each disjoint cycle in contiguous memory, we can now permute the first set of cycles of the equidistant gather I/O efficiently and in parallel. Thus for $r = O(\sqrt{N})$, this permutation takes $\frac{1}{P}\sum_{i=1}^{r}\left(1 + O\left(\frac{i}{B}\right)\right) = O\left(\frac{\sqrt{N}}{P} + \frac{N}{PB}\right) = O\left(\frac{N}{PB}\right)$ I/Os, assuming that $B \leq \sqrt{N}$.

After performing the first set of disjoint cycles, we perform the inverse of the above transposition to permute the elements back into their original order (this places each $\mathcal{T}_i$ into contiguous memory). To do this, we transpose the $r \times r$ matrix and perform a left circular shift on each row i by i positions. We complete the equidistant gather operation by performing a left circular shift on each subtree $\mathcal{T}_i$ by $i - 1$ positions. As outlined in Section IV-A, we recursively apply the equidistant gather operation to perform the vEB layout permutation. The I/O complexity of this algorithm is:

$$\begin{aligned} Q(N,P) &= \begin{cases} O(N/B) & \text{if } N \leq M \text{ and } P = 1 \\ \left\lceil \frac{\sqrt{N}}{P} \right\rceil Q\left(\sqrt{N}, \left\lceil \frac{P}{\sqrt{N}} \right\rceil\right) + O\left(\frac{N}{PB}\right) & \text{otherwise} \end{cases} \\ &= O\left(\frac{N}{PB} \log\log_K N\right) . \end{aligned}$$

Alternatively, a simpler solution would be to forgo the above described transposition phase and assign each processor a group of $O(B)$ cycles to permute sequentially. This can be done I/O efficiently since B consecutive elements will always contain elements from the same B cycles. The resulting I/O complexity is $O\left(\left(\frac{N}{PB} + \frac{\sqrt{N}}{B}\right) \log\log_K N\right)$. Although not as asymptotically efficient for large values of P, in practice for most architectures $P \leq \sqrt{N}$ and the first term will dominate, resulting in the same asymptotic complexity.

C. *B-tree Cycle-leader Algorithm*

Recall from Section IV-B that the B-tree cycle-leader algorithm is recursive, performing the equidistant gather operation while considering chunks of C elements as single units. Thus, as long as $C \geq B$, every swap of C elements will be I/O-efficient. Since $C = \left\lceil \frac{N}{(B+1)^2} \right\rceil$ for $N = (B+1)^{h+1} - 1$, only the base case ($C = 1$) will have a chunk size less than B. However, assuming that $M \geq (B+1)^2 - 1 = \Theta(B^2)$, we can simply load the base case into internal memory to perform the permutation in $O(B)$ I/Os. All other recursive levels are performed I/O efficiently, thus:

$$Q'(N,P) = \begin{cases} O(N/B) & \text{if } N \leq M \text{ and } P = 1 \\ \left\lceil \frac{B+1}{P} \right\rceil Q'\left(\frac{N}{B+1}, \left\lceil \frac{P}{B+1} \right\rceil\right) + O\left(\frac{N}{PB}\right) & \text{otherwise} \end{cases}$$
$$= O\left(\frac{N}{PB} \log_{B+1} \frac{N}{K}\right) ,$$
$$Q(N,P) = \begin{cases} O(N/B) & \text{if } N \leq M \text{ and } P = 1 \\ Q\left(\frac{N}{B+1}, \text{MIN}(P, \frac{N}{B(B+1)})\right) + Q'(N,P) & \text{otherwise} \end{cases}$$
$$= O\left(\left(\frac{N}{PB} + \log_{B+1} \frac{N}{K}\right) \log_{B+1} \frac{N}{K}\right) .$$

The BST algorithm is a special case of the B-tree algorithm with $B = 1$, which results in $O\left(\left(\frac{N}{PB} + \log \frac{N}{K}\right) \log \frac{N}{K}\right)$ I/O's.

VI. Extensions to non-perfect trees

Since the array is given in sorted order, any arbitrary size tree will be complete (though not necessarily perfect). For BSTs and B-trees, we can first permute the non-full level of leaves to the end of the array. For a tree of height h, the number of *full elements*, i.e., all the elements in the full levels, in a BST is $I = 2^h - 1$, and in a B-tree $I = (B+1)^h - 1$. The number of *non-full elements*, i.e., the elements on the non-full level, is $L = N - I$. The parents of non-full elements are striped across the non-full elements. We gather them to the front of the array and shift the non-full elements to the end of the array via a circular shift. To perform this gather, we apply a $(B+1)$-way un-shuffle (and additionally a B-way shuffle on the non-full elements for B-trees) as seen in Section III. Alternatively, we can apply the extended equidistant gather operation described in Section IV-B. This process takes $D(N) = O(T_{REV_2}(L))$ depth and $W(N) = O(L \cdot T_{REV_2}(L) + N)$ work for BSTs (via 2-way un-shuffle); and $D(N) = O(\log_{B+1} L)$ depth and $W(N) = O(L(B+1) \cdot \log_{B+1} L + N)$ work for B-trees (via extended equidistant gather). After applying this initial stage, we can proceed with the algorithm on the full elements which form a perfect tree of height $h - 1$.

To permute non-perfect vEBs, we first apply the BST approach to permute the non-full elements to the end of the array. If h is odd, removing the non-full elements will not impact the size of the root subtree ($\mathcal{T}_0$), allowing us to employ the standard vEB layout permutation algorithm for a full tree of height $h-1$. However if h is even, the size of $\mathcal{T}_0$ will change if we remove the non-full level. To account for this, we apply the extended equidistant gather for $l' = \frac{l}{2}$ (recall that l is the size of leaf subtrees in the vEB layout). Since this ignores the last non-full level, we then permute the non-full elements back into their correct subtree. To do this, we consider the $i \leq r$ perfect subtrees that have a full level of $l+1$ leaves that were ignored. We shift the $i(l+1)$ leaves to be directly after the internal nodes of $\mathcal{T}_i$ via a circular shift. We perform a 2-way shuffle where the elements in the first deck are in chunks of size l and the elements in the second deck are in chunks of size $l + 1$. This results in the $l + 1$ leaf elements being placed directly after the l internal elements for each subtree being considered. For each of the i pairs of internal and leaf elements, we perform a 2-way *in-shuffle* [29] (i.e., placing the elements of the second set before the first). If $\mathcal{T}_{i+1}$ has a non-full final level, we perform a similar procedure where we shift the non-full elements and perform a 2-way shuffle of the non-full and full elements of $\mathcal{T}_{i+1}$. We then recurse on each subtree to permute it into the vEB layout. Since we perform a sequence of shifts and 2-way shuffles (and un-shuffles), the work of this generalized algorithm is described by the recurrence:

$$W(N) = \begin{cases} \sqrt{N} \cdot W(\sqrt{N}) + O(N) & \text{if perfect} \\ \sqrt{N} \cdot W(\sqrt{N}) + O(N + \sqrt{N} \cdot T_{REV_2}(N)) & \text{otherwise} \end{cases}$$

Since $T_{REV_2}(N) = O(\log N) = O(\sqrt{N})$, the algorithmic complexity remains unchanged and work remains $W(N) = O(N \log \log N)$. However, the depth now depends on $T_{REV_2}(N)$:

$$D(N) = \begin{cases} D(\sqrt{N}) + O(1) & \text{if perfect} \\ D(\sqrt{N}) + O(1 + T_{REV_2}(N)) & \text{otherwise} \end{cases}$$
$$= O(T_{REV_2}(N) \cdot \log \log N) .$$

As seen in Section VII, some architectures have hardware instructions that can be used to perform $T_{REV_2}(N)$ in constant time, making it possible to operate on non-perfect vEBs without asymptotic performance loss.

VII. Experimental Results

In this section we evaluate the performance of our search tree permutation algorithms on both GPU and CPU architectures experimentally. We also quantify the performance of querying each search tree layout to determine the overall practical applicability of our algorithms.

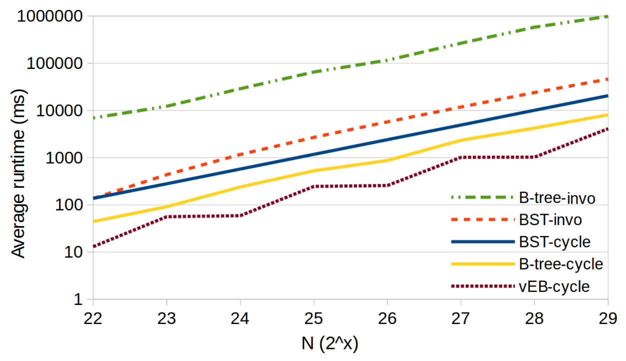
A. Methodology

Our CPU platform consists of two 10-core Intel Xeon E5-2680 and has 128GB of main memory and 25MB of L3 cache, with 64 byte cache lines. We use GCC 4.4.7 with the -O3 flag and use OpenMP [27] for parallelization. Our GPU platform is a nVidia Tesla K40 with 2,880 compute cores and 12GB of global memory. We use the CUDA 7.5 compiler. All shown experimental results are averages over 10 trials for arrays of 64-bit integer values ($B = 8$) and queries are randomly sampled from a uniform distribution.

B. CPU Results

For each search tree layout (BST, B-tree, and vEB), we measure the performance of the involution-based (Section III) and cycle-leader (Section IV) permutation algorithms.

Figure 3 plots sequential and parallel (using 20 threads) execution time vs. the input size (N). The results indicate that our cycle-leader approaches perform best in general, with the vEB cycle-leader algorithm outperforming all other approaches across the board. This is expected since this algorithm has lower I/O complexity than its competitors. Note



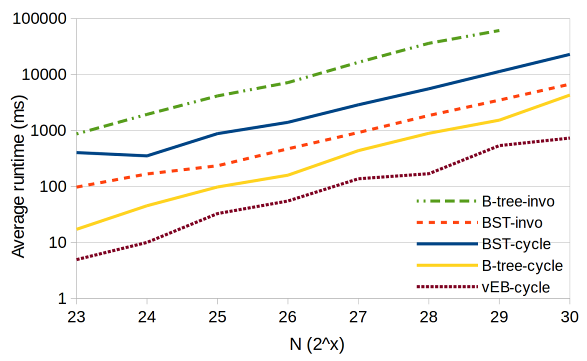


Fig. 3. Average time to permute a sorted array using each permutation algorithm on the CPU. Results using 1 thread (left) and using 20 threads (right) show our cycle-leader approaches performing the best overall.

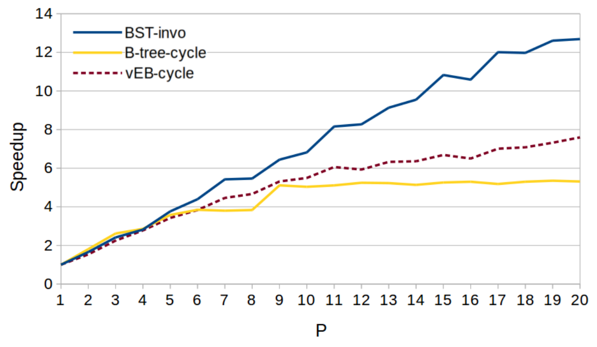


Fig. 4. Speedup of the fastest permutation algorithm for each layout on the CPU.

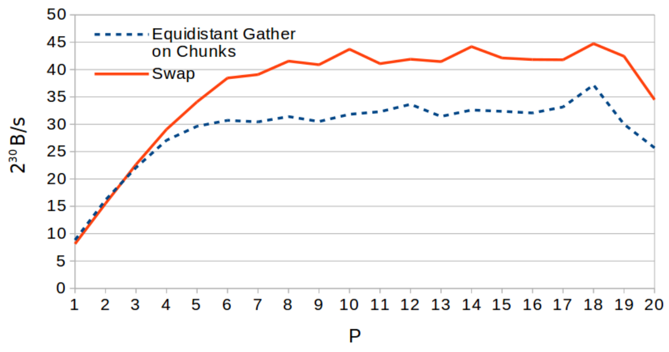


Fig. 5. Throughput of a single round of the equidistant gather on chunks of elements vs. swapping the first half of an array with the second half.

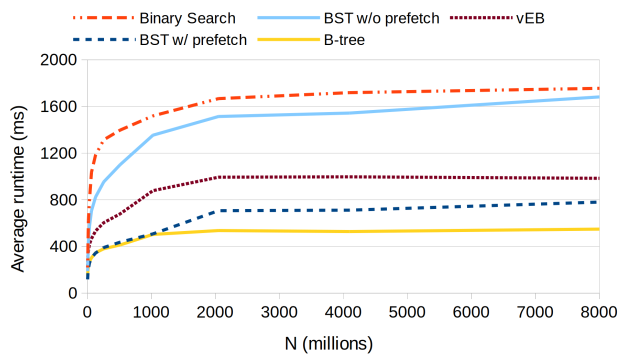
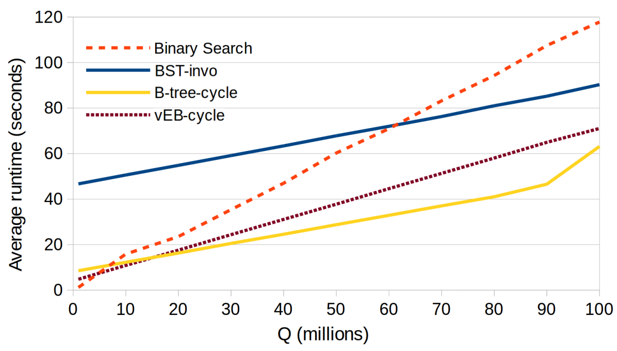


Fig. 6. Average time to perform 10^6 queries on each search tree layout on the CPU, for varying array size (N). While explicit prefetching improves BST search time, B-tree querying still provides the highest performance.

that the involution-based BST algorithm does not perform as well, despite being work-efficient in the PRAM model. This is because of the algorithm's poor spatial locality of memory accesses. Furthermore, since our CPU does not have a hardware primitive bit-reversal operation, $T_{REV_2}(N)$ takes $O(\log b)$ time, where b is the number of bits stored (i.e., 64).

Figure 4 plots the speedup of the fastest permutation algorithm for each of the layouts vs. the number of threads (P). We observe that the B-tree cycle-leader approach does not benefit from additional threads after $P = 9$. Since the B-tree cycle-leader approach repeatedly performs the equidistant gather on chunks of elements, to investigate the cause of the speedup performance, we measured the throughput of this procedure and compared it to the simplest analog: swapping the first half of an array with the second half of the same array; the results are plotted in Figure 5. We see that both procedures exhibit similar behavior, where after $P = 5$ the throughput does not increase substantially. Although the simple swap procedure achieves slightly higher throughput, we attribute this difference to the increased computation and the nested loop structure of the equidistant gather; we also note that the sharp decrease in speedup for $P = 20$ is due to a known non-deterministic performance issue specifically for $P = 20$ on the CPU platform used. In our B-tree cycle-leader implementation, the equidistant gather on chunks of elements is performed in parallel at depth 0 of the recursion tree for $P \geq 2$ and additionally at depth 1 of the recursion tree for $P \geq B + 1 = 9$; hence explaining the plateaus of speedups observed in Figure 4 at $P = 5$ and $P = 9$. Therefore, we conclude that the bottleneck of the B-tree cycle-leader approach is the throughput of swapping chunks of elements.

We compare the performance of each layout on a batch of search queries. Since each layout has benefits and drawbacks in terms of memory access patterns, we expect the more cache-efficient layouts, such as vEB and B-tree layouts, to provide better query performance. Figure 6 shows the average time to sequentially perform $Q = 10^6$ queries vs. the input size



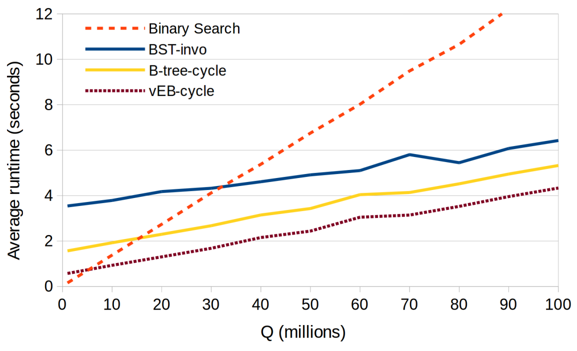


Fig. 7. Combined time of permuting and performing Q queries on an array of $N = 2^{29}$ elements on the CPU for $P = 1$ (left) and $P = 20$ (right).

(N). As a baseline, we include results for a binary search performed on the un-permuted sorted array. Unsurprisingly, the binary search performs the worst due to the lack of locality in memory access patterns. We include results for the BST layout querying both with and without explicit "prefetching," an optimization that was observed to improve BST query performance significantly in [23]. This observation is corroborated by our results with roughly a 2x improvement due to the use of prefetching. Nevertheless, the B-tree layout provides the best overall performance across the board. In spite of good locality, the vEB layout is on average 36.5% slower than the B-tree layout due to the more costly index computations.

An important practical question is: For how many queries is permutation worthwhile when compared to a no-permutation binary search approach? To answer this question, for each layout we measured the total runtime of permuting (using the fastest algorithm as previously determined) a sorted array of N elements, and then performing Q queries on the resulting layout. Figure 7 shows sequential (1 thread) and parallel (20 threads) results vs. the number of queries (Q), for an input size of $N = 2^{29}$. Sequentially, the B-tree layout provides the highest overall performance, with both fast query and permutation times. In parallel, however, the lower cost of vEB permutation offsets its increased permutation overhead. These results indicate that sequentially the cost of permuting to BST, B-tree, and vEB layouts is offset by the query runtime when $Q \geq 64M$ (12% of N), $Q \geq 8M$ (1.5% of N), and $Q \geq 4M$ (0.75% of N), respectively. In parallel, permuting into BST, B-tree, and vEB layouts is beneficial when $Q \geq 32M$ (6% of N), $Q \geq 15M$ (2.8% of N), and $Q \geq 5M$ (0.93% of N).

C. GPU Results

Graphics Processing Units (GPUs) are many-core architectures that are designed to provide high computational throughput, but designing algorithms and implementations that can approach peak performance is known to be challenging. Two key features of the GPU architecture make it compelling for this work: (i) GPUs have a relatively small memory (up to 16GB), making the in-place feature of our permutation algorithms crucial; and (ii) GPUs have high memory throughput and many compute cores, making them effective for a large number of independent search queries. We also note that the

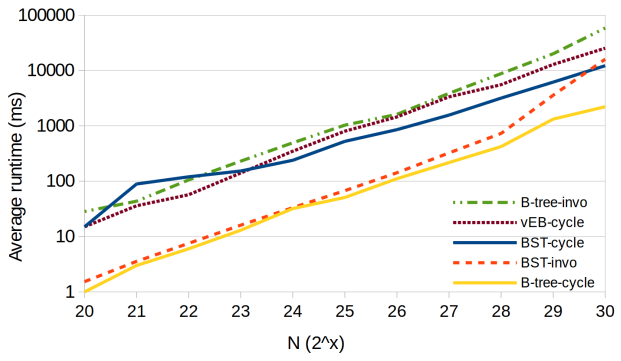


Fig. 8. Average time of performing each of our permutation algorithms on the GPU. As on the CPU, B-tree construction is fast. However, the recursion associated with vEB construction makes it perform poorly on the GPU.

cache line size for most modern GPUs is 128 bytes, i.e., larger than most CPUs. Thus, we employ $B = 32$ for our B-tree experiments hereafter. For more details on modern GPU architectures we refer interested readers to [26].

We developed GPU implementations of each of our permutation algorithms using standard good practices for writing fast GPU code [26]. While each of our algorithms provide a high degree of parallelism, synchronization and communication overheads can significantly degrade GPU performance. For this reason, we assign each thread to a query and have threads execute independently of each other.

Figure 8 plots the average permutation time vs. input size (N) and shows that the B-tree cycle-leader algorithm is the fastest, with the BST involution algorithm also providing good performance. Unlike our CPU, our GPU provides a hardware bit-reversal operation, allowing us to perform $T_{REV_2}(N)$ in $O(1)$ time. This is reflected in the performance results, where we see BST involutions performing well while B-tree involutions perform poorly. Interestingly, the vEB cycle-leader algorithm, which was the fastest algorithm on the CPU, performs significantly worse on the GPU. We attribute this slowdown to the fact that this algorithm uses recursion, which is known to cause performance degradation on GPUs. Developing an iterative version for the GPU, which we leave for future work, may improve its performance.

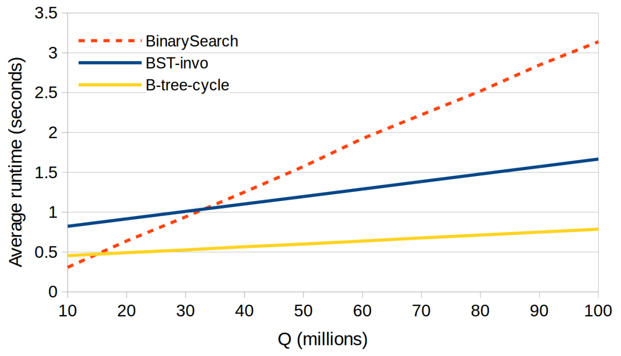


Fig. 9. Combined time to permute and query each layout on the GPU with $N = 2^{29}$. Despite the large N, few queries are needed to offset the overhead of permuting.

Figure 9 shows combined permutation and searching runtime vs. the number for queries (Q), for input size $N = 2^{29}$. We omit the results for the vEB layout because the high overhead of permuting it on the GPU makes it much slower than all other approaches. As in Figure 7 for the CPU, we include the binary search as a baseline. The permutation overhead for the BST and B-tree layouts is offset by the gain in querying performance when $Q \geq 34M$ (12.7% of N) and $Q \geq 15M$ (5.6% of N), respectively.

VIII. Conclusion

Implicit search tree layouts can improve search query performance by exploiting locality of reference and, consequently, cache efficiency. However, given initially sorted input, permuting it into a search tree layout requires extra space and can be costly, thereby bringing into question the usefulness of implicit search tree layouts in memory-constrained environments and/or when few search queries need to be performed.

In this work we present parallel in-place algorithms for permuting a sorted array into popular search tree layouts. Our algorithms exhibit the following features which make them exceptionally practical: 1) they operate in-place, making it possible to permute inputs that occupy all available space; 2) they are efficient in parallel, allowing the use of many-core architectures; and 3) our cycle-leader algorithms are I/O-efficient, resulting in implementations that effectively utilize the cache hierarchy. We measure the performance of our algorithms on both CPU and GPU platforms, and key results are as follows. On the CPU (resp. GPU), permuting an array of $N = 2^{29}$ 64-bit elements into a vEB layout (B-tree layout), and then searching for Q 64-bit queries, all in parallel, outperforms a parallel binary search when $Q \geq 5M$ ($Q \geq 27M$). On each platform, the number of queries beyond which input array permutation is worthwhile is less than 1% and 6% of the input size, respectively.

This work underscores the importance of I/O-efficient algorithms and techniques. As such, the use of efficient memory layouts beyond searching could prove an interesting direction for future research.

References

[1] Alok Aggarwal and Jeffrey S. Vitter. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988.

[2] Deepak Ajwani and Nodari Sitchinava. Empirical evaluation of the parallel distribution sweeping framework on multicore architectures. In *European Symposium on Algorithms*, pages 25–36, 2013.

[3] Lars Arge, Michael T. Goodrich, Michael Nelson, and Nodari Sitchinava. Fundamental parallel algorithms for private-cache chip multiprocessors. In *Proc. of 20th ACM SPAA*, pages 197–206, 2008.

[4] R. Bayer and E. McCreight. Organization and maintenance of large ordered indices. In *Proc. of ACM SIGFIDET Workshop on Data Description, Access and Control*, pages 107–141, 1970.

[5] Andrew Bradford. *The Investment Industry for IT Practitioners*. 2008.

[6] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21(2):201–206, 1974.

[7] G.S. Brodal, R. Fagerberg, and R. Jacob. Cache oblivious search trees via binary trees of small height. In *Proc. of 13th ACM-SIAM Symposium on Discrete Algorithms*, pages 39–48, 2002.

[8] Fabio Cannizzo. A fast and vectorizable alternative to binary search in o(1) with wide applicability to arrays of floating point numbers. *Journal of Parallel and Distributed Computing*, 113:37 – 54, 2018.

[9] Yi-Jen Chiang. Experiments on the practical I/O efficiency of geometric algorithms: Distribution sweep vs. plane sweep. In *Algorithms and Data Structures*, pages 346–357, 1995.

[10] M. Dalkilic, E. Haytaoglu, and G. Tokatli. A simple shuffle-based stable in-place merge algorithm. *Proc. Computer Science*, 3:1049 – 1054, 2011.

[11] D. D'Angeli and A. Donno. Shuffling matrices, kronecker product and discrete fourier transform. *Discrete Appl. Math*, 233:1 – 18, 2017.

[12] M. Davio. Kronecker products and shuffle algebra. *IEEE Transactions on Computers*, C-30(2):116–125, 1981.

[13] P. Diaconis, R.L Graham, and W.M Kantor. The mathematics of perfect shuffles. *Advances in Applied Mathematics*, 4(2):175 – 196, 1983.

[14] J. Ellis and M. Markov. In-situ, stable merging by way of the perfect shuffle. *The Computer Journal*, 43(1):40–53, 2000.

[15] John Ellis, Hongbing Fan, and Jeffrey Shallit. The cycles of the multiway perfect shuffle permutation. *Discrete Mathematics & Theoretical Computer Science*, 5(1):169–180, 2002.

[16] John Ellis, Tobias Krahn, and Hongbing Fan. Computing the cycles in the perfect shuffle permutation. *Information Processing Letters*, 75(5):217 – 224, 2000.

[17] John Ellis and Ulrike Stege. A provably, linear time, in-place and stable merge algorithm via the perfect shuffle. *CoRR*, 2015.

[18] Faith E. Fich, J. Ian Munro, and Patricio V. Poblete. Permuting in place. *SIAM Journal on Computing*, 24(2):266–278, 1995.

[19] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *40th FOCS*, pages 285–297, 1999.

[20] W.H. Inmon, Derek Strauss, and Genia Neushloss. *DW 2.0: The Architecture for the Next Generation of Data Warehousing*. 2010.

[21] Peiyush Jain. A simple in-place algorithm for in-shuffle. *CoRR*, 2008.

[22] Joseph JaJa. *Introduction to Parallel Algorithms*. 1992.

[23] Paul-Virak Khuong and Pat Morin. Array layouts for comparison-based searching. *J. Exp. Algorithmics*, 22:1.3:1–1.3:39, 2017.

[24] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. 1998.

[25] Richard E. Ladner, Ray Fortna, and Bao-Hoang Nguyen. A comparison of cache aware and cache oblivious static search trees using program instrumentation. In *Experimental Algorithmics*, pages 78–92. 2002.

[26] NVIDIA. CUDA programming guide 9.0, 2017.

[27] OpenMP Architecture Review Board. OpenMP application program interface version 3.0, 2008.

[28] Harald Prokop. Cache-oblivious algorithms. Master's thesis, MIT, 1999.

[29] Christian Ronse. A generalization of the perfect shuffle. *Discrete Mathematics*, 47:293 – 306, 1983.

[30] H. S. Stone. Parallel processing with the perfect shuffle. *IEEE Transactions on Computers*, C-20(2):153–161, 1971.

[31] S. Fahmeeda Sultana and D. Shubhangi. Video encryption algorithm and key management using perfect shuffle. *International Journal of Engineering Research and Applications*, 07:01–05, 2017.

[32] Jeffrey Scott Vitter. Algorithms and data structures for external memory. *Found. Trends Theor. Comput. Sci.*, 2(4):305–474, 2008.

[33] Qingxuan Yang, John Ellis, Khalegh Mamakani, and Frank Ruskey. In-place permuting and perfect shuffling using involutions. *Information Processing Letters*, 113(10):386 – 391, 2013.