Beyond Binary Search: Parallel In-place Construction of Implicit Search Tree Layouts

Kyle Berney, Henri Casanova, Ben Karsin, and Nodari Sitchinava

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. When considering the total time to permute the data in-place and to perform a series of search queries, the vEB layout provides the best performance on the CPU. Given an input of N=537 million 64-bit integers, the benefits of query performance (compared to binary search) outweigh the cost of in-place permutation when performing as few as 0.37% of N queries. On the GPU, results depend on the particular architecture, with the B-tree and vEB layouts performing the best. The number of queries necessary to reach the break-even point with binary search ranges from 1.3% to 8.9% of N=1,074 million 32-bit integers.

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

1 Introduction

Searching is a fundamental computational problem that arises in many applications. When many queries are expected to be performed, data is often stored in a data structure that is conducive to efficient searching. Pointer-based search trees are an example of such a data structure; for example, a *binary search tree (BST)* is a binary tree such that for every vertex v, the key stored at v is greater than all the keys stored in the subtree rooted at v's left child and smaller than all the keys stored in the subtree rooted at v's right child. 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, which can be prohibitive in limited-memory environments. 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 (i.e., data which will not change in the future), storing data in sorted order and performing binary search seems to be the preferred approach. For example, Khuong and Morin [1] observe that binary searches on static sorted arrays 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 [2], sales [3], advertising [1], and numerical analysis [4].

Despite its simplicity and optimal $\Theta(\log N)$ complexity in the classical RAM model, binary search on a sorted array of size N is not the most efficient approach on modern memory hierarchies. This is due to the fact that the complexity metric of the RAM model does not capture an algorithm's locality of reference, which is key to efficient implementations on modern memory designs. One way to see the inefficiency of binary search is to view the 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. In this way, child-parent relationships of the vertices are 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[1] 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. Thus, when performing binary search on a sorted array of size N, the i-th vertex visited, for $i \in \{2, ..., \lfloor \log N \rfloor\}$, is $n/2^{i+1}$ array locations away from the $(i-1)$-th (previous) vertex visited.

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 [1], [5], [6]. Consequently, if many search queries need to be performed, it may be worth spending extra time to re-arrange the sorted data into a memory layout that will increase locality of reference of each query.

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 1.2) and the minimum number of queries needed to justify the extra time to perform such transformations in practice. Moreover, since binary search on already sorted data does not require any additional space, we require that these transformations be performed *in-place*.

This material is based upon work supported by the National Science Foundation under Grant No. CCF-1533823 and CCF-1911245.

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.

1.1 Models of Computation

In this work, we analyze the time complexity of our parallel algorithms in the *Parallel Random Access Machine (PRAM)* model [7]. Given an input of size N, the PRAM model defines two complexity metrics of an algorithm: *work*, denoted $W(N)$, is the total number of operations performed by all processors, and *depth* (also known as *span* or *critical path length*), denoted $D(N)$, 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 [8] 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 [9] – a parallel extension of the (sequential) *External Memory (EM)* model [10]. 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 the 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 [11], [12]. 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, denoted $Q(N,P)$, is defined as the maximum number of blocks transferred by any one of the processors throughout the computation.

1.2 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. 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. Figure 1 depicts an example 15-node BST layout.

A *complete B-tree* [13] 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. B-trees exhibit improved cache efficiency (compared to a BST) when employing a B value that coincides with the cache line size of the particular machine. The *Level-order B-tree layout* is defined by the breadth-first left-to-right traversal of a complete B-tree. Figure 2 depicts the B-tree layout for $N = 26$ and $B = 2$.

The *van Emde Boas (vEB) layout* [14] 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 $\lfloor (h-1)/2 \rfloor$ containing $r = 2^{\lfloor (h-1)/2 \rfloor} - 1$ vertices, and $r+1$ bottom subtrees $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_{r+1}$, each of height $\lceil (h-1)/2 \rceil$ and each rooted at the children of the $(r+1)/2$ leaves of $\mathcal{T}_0$. The vEB layout of $\mathcal{T}$ is defined recursively as the vEB layout of $\mathcal{T}_0$, followed by the vEB layouts of each $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_{r+1}$. Figure 3 depicts an example vEB layout with 15 nodes.

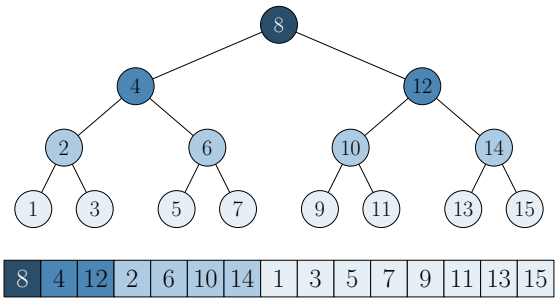

Fig. 1. BST layout for $N = 15$.

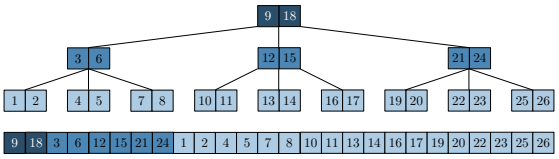

Fig. 2. Level-order B-tree layout for $N = 26$ and $B = 2$.

The above definition of the vEB layout for $N \neq 2^{h+1} - 1$ complicates the permutation algorithms described in Sections 3.3 and 4.1 because the number of vertices in each bottom subtree may be different. Instead, in this work, we modify the definition of the vEB layout for $N \neq 2^{h+1} - 1$ as follows. Let $r = 2^{\lfloor (h-1)/2 \rfloor} - 1$ and $l = 2^{\lceil (h-1)/2 \rceil} - 1$. The top subtree $\mathcal{T}_0$ of the vEB layout will always contain r vertices and the remaining $N - r$ elements will form $x = \lceil (N-r)/l \rceil$ bottom subtrees, $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_x$. Each of the first $y = \lfloor (N-r)/l \rfloor$ bottom subtrees, $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_y$, will consist of exactly l vertices. If $N - r$ is not a multiple of l, i.e., $x = y + 1$, then the last bottom subtree, $\mathcal{T}_x$, will contain $1 \leq l' < l$ vertices. As in the standard definition, the vEB layout consists of $\mathcal{T}_0$, immediately followed by $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_x$, with each subtree laid out recursively ($\mathcal{T}_0, \mathcal{T}_1, \ldots, \mathcal{T}_y$ uses the standard vEB layout and if $x = y+1$, then $\mathcal{T}_x$ uses this modified approach).

In our definition of the vEB layout, at each recursive level, there is at most one bottom subtree that contains a different number of vertices than all other bottom subtrees and is always located at the end of the layout. This observation allows us to easily adapt the permutation algorithms described in Sections 3.3 and 4.1 and query optimizations described in Section 7.1 to work with arrays of sizes $N \neq 2^{h+1} - 1$, without affecting the asymptotic analysis.

The I/O complexity 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 [5], [14]. In theory, because the definition of the vEB layout does not make use of the parameter B, i.e., it is *cache-oblivious* [15], querying the vEB layout on architectures with multiple levels of cache

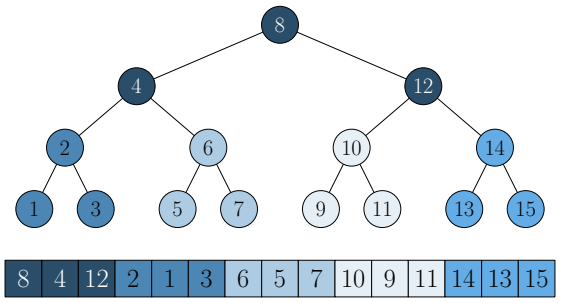

Fig. 3. van Emde Boas (vEB) layout for $N = 15$.

will result in the asymptotically optimal number of accesses at every level of the memory hierarchy [14].

The relative performance of querying each of these search tree layouts has been studied empirically. Ladner *et al.* [6] 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.* [5] 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 [1], who show that, by using explicit prefetching and other optimizations, the BST layout can outperform both the B-tree and vEB layouts.

1.3 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 $\pi(i)$ 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 in the PRAM model. 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 disjoint cycles up front.

When it comes to *in-place* permutations, Fich *et al.* [16] 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.* [17] 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* [18]: 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 [19]. These permutations have many applications (e.g., parallel processing [20], Fast Fourier Transforms (FFT) [20], [21], Kronecker products [21], [22], encryption [23], sorting [20], and merging [24], [25], [26]). Ellis *et al.* [27], [28] 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 [29] 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.* [17] 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.* [17] for designing our involution-based permutation algorithms.

1.4 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) [26]. 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* [24], [30]. When it comes to parallel algorithms, 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.

2 Contributions

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.* [17] and Fich *et al.* [16], 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. The B-tree involutions can be used in order 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 $B = 1$.

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 in the base-k representation of the integer N being reversed, 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. The time and I/O complexities of our algorithms are summarized in Table 1. Our cycle-leader algorithms exhibit better I/O complexity, while our involution-based algorithms are much simpler and trivial to parallelize.

We evaluate these algorithms experimentally on multi-core 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 $0.0037N$ search queries on a CPU and $0.013N$ on a GPU.

The remainder of this paper is organized as follows. Section 3 presents our involution-based algorithms and Section 4 presents our cycle-leader algorithms, each section analyzing the time complexity of these algorithms. For ease of exposition, in Sections 3 and 4 we consider only *perfect* trees, i.e., complete trees in which every level is full. Section 5 analyzes the I/O complexity of our algorithms. Section 6 discusses extensions of our algorithms to non-perfect trees. Section 7 goes over experimental optimizations and Section 8 presents experimental results. Finally, Section 9 concludes with a summary.

2. E.g., Nvidia GPUs implement this operation in hardware for $k = 2$.

3 Involution Approach

3.1 BST Layout

A perfect BST contains $N = 2^d - 1$ vertices. As mentioned in Section 1, Fich *et al.* [16] 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 the 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 [16], 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 8. Thus, this algorithm has depth $D(N) = O(T_{\text{REV}_2}(N))$ and work $W(N) = O(N \cdot T_{\text{REV}_2}(N))$.

3.2 B-tree Layout

The B-tree layout algorithm relies on the k-way perfect shuffle involution approach developed by Yang *et al.* [17]. Let us first review their results.

Let $J_r(i) = g \cdot (r \cdot (\frac{i}{g})^{-1} \pmod{\frac{N-1}{g}})$ where g is the greatest common divisor of i and $N-1$. Yang *et al.* [17] 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 (i.e., 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.

TABLE 1
Asymptotic time 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 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	$O\left(\frac{N}{P} \log \log N\right)$	$O\left(\frac{N}{PB} \log \log_K N\right)$

Recall that REV_k can take up to $O(\log_k N)$ time. Finding the modular inverse, however, requires using the extended Euclidean algorithm [17], which takes $O(\log N)$ time. The latter dominates the running time, resulting in $O(\log N)$ time for both operations. The work and depth complexities of our B-tree permutation algorithm are given in Proposition 2 and 3, respectively.

Proposition 2.

$$W(N) = W\left(\frac{N}{B+1}\right) + O(N \log N) = O(N \log N) .$$

Proposition 3.

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

3.3 van Emde Boas Layout

We are able to apply the B-tree layout algorithm for the vEB layout of height h, by using $B = 2^{\lceil (h-1)/2 \rceil} - 1$ and recursing on each subtree of the vEB layout. The resulting work and depth complexities are:

Proposition 4.

$$W(N) = \sqrt{N} \cdot W\left(\sqrt{N}\right) + O(N \log N) = O(N \log N) .$$

Proposition 5.

$$D(N) = D\left(\sqrt{N}\right) + O(\log N) = O(\log N) .$$

4 Cycle-leader Approach

4.1 van Emde Boas Layout

Recall from Section 1.2 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 $r = 2^{\lfloor (h-1)/2 \rfloor} - 1$ vertices of the upper $\lfloor \frac{h-1}{2} \rfloor$ levels, where $h = \lfloor \log N \rfloor$, while $\mathcal{T}_1, \ldots, \mathcal{T}_{r+1}$ are "leaf" subtrees consisting of $l = 2^{\lceil (h-1)/2 \rceil} - 1$ 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 $1 \le j \le r+1$, $a_j = r + (j-1)l + 1$ 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 is general enough to work for any $r \le l$.)

We use $\mathcal{T}_i[a : b]$ 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 $\mathcal{T}_j$, for $j \ge 1$, are initially located:

Proposition 6. *For all $i = r - j + 2$, $1 \le j \le r + 1$, $\mathcal{T}_j[i : l]$ 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 $1 \le i, j \le l$, $\mathcal{T}_j[i]$ is initially located at index $i_{orig} = (j-1)(l+1) + i$. Hence, we check if $i_{orig} \ge a_j = r + (j-1)l + 1$. Solving for i results in $i \ge r - j + 2$. □

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 4 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 $\mathcal{T}_i$.

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

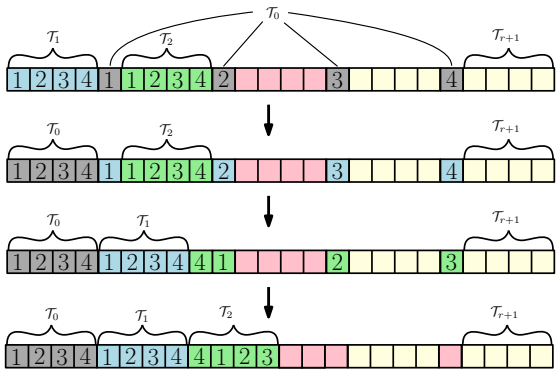

Fig. 4. Illustration of the series of swaps needed to sequentially perform the equidistant gather operation for $r = l$.

mentioned in Section 1.3, 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.* [17]. 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. The work and depth complexities of this algorithm are:

Proposition 7.

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

Proposition 8.

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

4.2 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 single 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:

Proposition 9.

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

Proposition 10.

$$\begin{aligned} D'(N) &= D'\left(\frac{N}{B+1}\right) + O(1) \\ &= O(\log_{B+1} N) \end{aligned}$$

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

Proposition 11.

$$\begin{aligned} W(N) &= W\left(\frac{N}{B+1}\right) + W'(N) \\ &= W\left(\frac{N}{B+1}\right) + O(N \log_{B+1} N) \\ &= O(N \log_{B+1} N) \end{aligned}$$

Proposition 12.

$$\begin{aligned} D(N) &= D\left(\frac{N}{B+1}\right) + D'(N) \\ &= D\left(\frac{N}{B+1}\right) + O(\log_{B+1} N) \\ &= O(\log^2_{B+1} N) \end{aligned}$$

4.3 BST Layout

We can apply the B-tree cycle leader algorithm (Section 4.2) 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 3.1, the cycle-leader algorithm exhibits better spatial locality, which we analyze in the next section.

5 I/O Optimizations

In this section, we analyze the I/O complexity of our proposed algorithms in the *parallel external memory (PEM)* model [9] – 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 5.2, we increase this assumption to $M \geq B^2$ (standard tall-cache assumption) and consequently $P \leq \frac{N}{B^2}$.

5.1 Involution-based Algorithms

We first consider the involution-based algorithms described in Section 3. 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. Proposition 13 and 14 provides the I/O complexity of the B-tree layout and vEB layout, respectively.

Proposition 13.

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

Proposition 14.

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

5.2 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.* [17], which involves two rounds of array reversals. We can reverse k elements in-place and in parallel by performing $\lfloor \frac{k}{2} \rfloor$ 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.

The vEB cycle-leader approach, described in Section 4.1, 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 an $(r+1) \times (l+1)$ 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 $\mathcal{T}_0$. Additionally for $r < l$, we ignore the remaining rightmost $(l - r)$ columns, as these elements do not participate in any cycles.

Thus, we consider a square matrix of size $r \times r$. Figure 5 illustrates this representation for $r = l$, 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 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 $P \le N/B^2$ and $M \ge B^2$, we can perform matrix transposition in $O(\frac{r^2}{PB})$ I/Os by tiling the matrix into sub-matrices of size $B \times B$ [10], [31].

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 $\frac{1}{P}\sum_{i=1}^{r}(1 + O(\frac{i}{B})) = O\left(\frac{\sqrt{N}}{P} + \frac{N}{PB}\right) = O(\frac{N}{PB})$ I/Os, assuming that $B \le \sqrt{N}$.

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

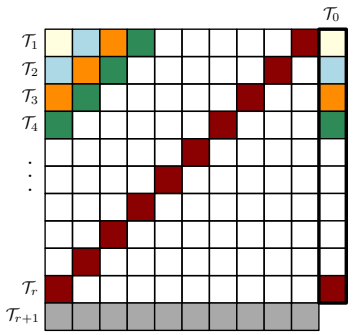

Fig. 5. Illustration of the distinct cycles of the equidistant gather operation for $r = l$. We consider memory as an $(r+1) \times (l+1)$ matrix. Shifting each row and transposing the inner $r \times r$ matrix lets us perform each cycle I/O efficiently.

performing a left circular shift on each subtree $\mathcal{T}_i$ by $i - 1$ positions. As outlined in Section 4.2, the equidistant gather operation is applied recursively to perform the vEB layout permutation. The I/O complexity of this algorithm is:

Proposition 15.

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

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 $P \le \sqrt{N}$ and the first term will dominate, resulting in the same asymptotic complexity.

5.3 B-tree Cycle-leader Algorithm

Recall from Section 4.2 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 $C \ge B$, 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 $M \ge (B+1)^2 - 1 = \Theta(B^2)$, 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:

Proposition 16.

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

Proposition 17.

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

Recall from Section 4.3 that the BST algorithm is a special case of the B-tree algorithm where the node size is a single element. In this case, the last $\Theta(\log B)$ rounds will have a chunk size less than B. Thus, once $N = O(B)$, we load the array into internal memory which results in $O\left(\left(\frac{N}{PB} + \log \frac{N}{K}\right) \log \frac{N}{K}\right)$ I/O's.

6 Extensions to non-perfect trees

Since the array is given in sorted order, any arbitrary size BST or B-tree will be complete (though not necessarily perfect). Hence 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., 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 in the non-full level, is $L = N - I$. In both trees, the parents of non-full elements are initially located in the array such that they partition 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 3.2. Alternatively, we can apply the extended equidistant gather operation described in Section 4.2. 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 $h-1$.

Recall from Section 1.2 that $r = 2^{\lfloor (h-1)/2 \rfloor} - 1$ is the size of the top subtree, $\mathcal{T}_0$, and $l = 2^{\lceil (h-1)/2 \rceil} - 1$ is the size of each of the first $y = \lfloor (N-r)/l \rfloor$ bottom subtrees, $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_y$. We first gather the first y elements of $\mathcal{T}_0$, which are initially located at every $(l+1)$-th array location, to the front of the array. Then we shift the remaining $r - y$ elements of $\mathcal{T}_0$, which reside at the end of the array, to the front of the array after the first y elements of $\mathcal{T}_0$. To perform this gather, we can either perform the equidistant gather operation described in Section 4.1; or we can apply an $(l+1)$-way un-shuffle followed by an l-way shuffle on the elements in $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_y$. The resulting work and depth of this process using the equidistant gather is $W(N) = O(N)$ and $D(N) = O(1)$. After this initial permutation, we recurse on subtrees $\mathcal{T}_0, \mathcal{T}_1, \ldots, \mathcal{T}_y$ using the algorithm for the perfect vEB layout. If $x = \lceil (N-r)/l \rceil = y+1$, then we additionally recurse on the last non-perfect subtree $\mathcal{T}_x$ using the algorithm just described for the non-perfect vEB layout.

7 Experimental Optimizations

7.1 Query Optimizations

To provide a fair and thorough comparison of each search tree layout, we consider query optimizations that have been shown to improve performance.

The work of Khuong and Morin [1] found that using explicit prefetching can significantly improve query performance on BST layouts. As explicit prefetching instructions are typically provided on modern CPUs, in Section 8.2 we consider performance both with and without this optimization to determine its efficacy in practice.

Brodal et al. [5] describe a vEB query approach that utilizes a precomputed table of size $O(\log N)$. Consider an arbitrary node in the vEB located at depth d and unfold the vEB recursion such that the considered node is the root of a vEB bottom tree. The precomputed table stores at index d the number of nodes in the corresponding bottom tree, the number of nodes in the corresponding top tree, and the depth of the root of the corresponding top tree. Thus, when performing a vEB query, the precomputed table can be used to calculate the index of the next node in $O(1)$ time. For non-perfect vEBs, the last leaf subtree may contain fewer elements than other leaf subtrees and thus, may have a different subtree height. Due to this, an additional table is needed to be able to query the non-perfect leaf subtree. This is needed for all non-perfect leaf subtrees for all recursive levels of the vEB. In the worst case, this requires $\sum_{i=0}^{\log \log N} \frac{\log N}{2^i} = O(\log N)$ space. This query optimization using the precomputed table(s) is used throughout Section 8.

7.2 Hybrid BST Layout

For permuting non-perfect BSTs, we must perform an initial permutation to gather and shift the non-full elements to the end of the array (as described in Section 6). To do this, we can either use the equidistant gather operation (i.e., cycle-leader approach) or a 2-way un-shuffle (i.e., involution approach).

In the conference version of this paper, we found that the involution approach performs better than the cycle-leader approach for perfect BSTs. However, recall from Section 3.1 that the BST involution approach uses a pair of involutions to permute the sorted array into the BST layout, which does not require the use of any shuffles or un-shuffles. In comparison, the involution approaches that do use shuffles and un-shuffles (e.g. the B-tree involution permutation) do not perform well. Therefore, we additionally consider the BST hybrid approach, which uses the extended equidistant gather for the initial permutation to gather the non-full elements, then uses the pair of involutions to permute the full elements into the BST layout. The BST hybrid approach is additionally used in Section 8.

7.3 Modified van Emde Boas Layout

In the conference version of this paper, results on the vEB permutation algorithms showed poor performance on GPUs [32]. This slowdown is attributed to the recursive implementation of these algorithms, which is known to degrade performance on GPUs. However, developing iterative versions of these algorithms on the GPU is challenging, due to the potentially uneven size of the top and bottom trees in the vEB layout. Instead, we define a variant of the vEB layout, which we call the *modified van Emde Boas* layout (mvEB).

In the mvEB layout, the height of the bottom trees are rounded up to the nearest power of two (at the expense of the top subtree's height being shortened by the same change in height as the leaf subtrees). In this way, the bottom subtrees are guaranteed to always be perfectly balanced, i.e.,

all of the top and bottom trees in the following recursive divisions always contain the same number of nodes. This makes developing an iterative version for the perfectly balanced subtrees possible in a single GPU kernel, for each recursive division. In Section 8.3, the mvEB layout is used instead of the recursively implemented vEB layout.

8 Experiments

In this chapter 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.

8.1 Methodology

Our CPU platform consists of two 10-core Intel Xeon E5-2680's with 128GB of main memory and 25MB of L3 cache with 64 byte cache lines. We use GCC 8.3.0 with the `-O3` flag and use OpenMP [33] for parallelization. Our GPU platforms are: (1) a Nvidia Tesla K40 with 2,880 compute cores, 12 GB of GDDR5 type global memory with a theoretical bandwidth of 288 GB/s, and compute capability 3.5; (2) a Nvidia Quadro M4000 with 1,664 compute cores, 8 GB of GDDR5 type global memory with a theoretical bandwidth of 192 GB/s, and compute capability 5.2; and (3) a Nvidia GeForce RTX 2080 Ti with 4,352 compute cores, 11 GB of GDDR6 type global memory with a theoretical bandwidth of 616 GB/s, and compute capability 7.5. We use the CUDA 10.1 compiler with the `-O3` and `-use_fast_math` flags.

Experiments are conducted on arrays of 64-bit integer values ($B = 8$) on our CPU platform and 32-bit integer values ($B = 32$) on our GPU platforms. Permutation experiments are conducted on both perfect trees (powers of 2 minus 1 for BSTs and vEBs; and powers of $(B + 1)$ minus 1 for B-trees) and non-perfect trees (powers of 10 for all trees; and additionally powers of 2 minus 1 for B-trees). All experimental results are averages over 10 trials and queries are randomly sampled from a uniform distribution of $1, 2, \ldots n$, making all the searches 100% successful. Some figures referenced throughout this section can be found in Appendix A and the code used in these experiments can be found at: `https://github.com/algoparc/Tree-Layouts`.

8.2 CPU Results

Figure A.1 (respectively Figure A.2) plots the sequential (respectively parallel) execution time vs. the input size of all of the search tree layouts. The results indicate that our cycle-leader approaches perform best in general, with the vEB cycle-leader algorithm outperforming all other approaches for 20 threads. This is expected since this algorithm has lower I/O complexity than its competitors. Note 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). For non-perfect trees, we see an increase in runtime for all layouts, except for the B-tree and vEB involution permutations. The increase in runtime is significant for the BST involutions (for both $P = 1$ and $P = 20$) and the BST hybrid approach (for $P = 20$). This is due to the fact that the BST involution permutation for perfect trees relies on a single pair of involutions, which is faster than performing either an unshuffle (which is used in the B-tree and vEB involution permutations) or the extended equidistant gather operation for $B = 1$ and $P = 20$ (which is used in the BST cycle-leader permutation).

Figure A.3 plots the speed-up factors of the fastest permutation algorithm for each tree layout versus the number of threads (P) for $N = 2^{29} - 1$. We observe that the B-tree cycle-leader approach does not benefit from additional threads after $P = 9$ with a peak speed-up factor of around 4. Since the B-tree cycle-leader algorithm repeatedly performs the equidistant gather on chunks of elements, to investigate the cause of the speedup performance, we compare it to the simplest analog: swapping the first half of an array with the second half of the same array using chunks of elements. The results are plotted in Figure A.4, where we see that both procedures exhibit similar speed-up factors. Thus, we conclude that there is an inherent bottleneck in permuting chunks of elements, which inhibits the parallel performance of the B-tree cycle-leader approach.

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 the vEB and B-tree layouts, to provide better query performance. Figure A.5 (and Figure A.6) shows the average time to sequentially perform $Q = 10^6$ queries versus the input size. As a baseline, we include results for a binary search performed on the un-permuted sorted array. Notice the spikes in runtime for binary search, which occurs when the array size is close to a power of 2. This is due to cache-line aliasing issues described in [1]. For other array sizes, binary search outperforms BST querying without prefetching after 10 million elements and is competitive with vEB querying until $N = 2^{27} - 1$ elements. As mentioned in Section 7.1, we compare the performance of BST querying with and without explicit prefetching, which is an optimization observed to improve BST query performance significantly in [1]. This observation is corroborated by our results with a speedup of 1.6-2.6 due to the use of explicit prefetching. Nevertheless, the B-tree layout provides the fastest querying once $N > 2^{29} - 1$. In spite of good locality, the vEB layout is on average 48.45% slower than the B-tree layout due to 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 A.7 and Figure A.8 shows the sequential results versus the number of queries for an input size of $N = 2^{29} - 1$ and $N = 1$ billion array elements, respectively. Similarly, Figure 6 and Figure A.9 shows the parallel results for $P = 20$. Sequentially, the B-tree layout provides the highest overall performance, with both fast query and permutation times. In parallel however,

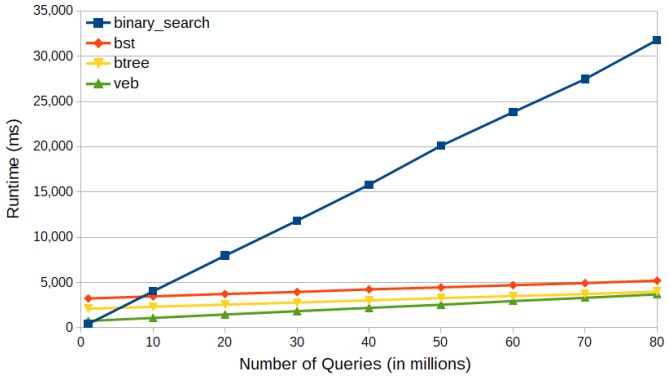

Fig. 6. Combined time of permuting and performing Q queries on an array of $N = 2^{29} - 1$ elements on the CPU using 20 threads.

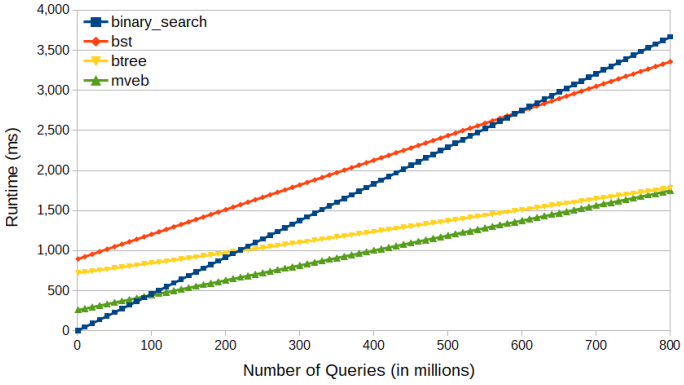

Fig. 7. Combined time to permute and query each layout on the Nvidia RTX 2080 Ti with $N = 2^{30} - 1$ elements.

the lower cost of the vEB permutation offsets its increased query overhead. Due to the slow permutation runtime of the BST layout for non-perfect trees, for $Q \leq N$, the cost of permutation is never offset by the faster query runtime. Table 2 summarizes the number of queries needed for it to be worthwhile (compared to an equal number of binary search queries) to permute and query on each search tree layout.

8.3 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: (1) GPUs have a relatively small memory, making the in-place feature of our permutation algorithms crucial; and (2) GPUs have high memory throughput and many compute cores, making them effective for a large number of independent search queries. We note that when accessing global memory, coalesced accesses are recommended to minimize the number of memory transactions. Thus by using $B = 32$, each thread in a warp will accesses consecutive memory locations resulting in $O(1)$ global memory transactions. For more details on modern GPU architectures we refer interested readers to [34].

We developed GPU implementations of each of our permutation algorithms using standard good practices for writing fast GPU code [34]. Recall that we use our modified van Emde Boas layout (Section 7.3), rather than the van Emde Boas layout. While each of our algorithms provides a high degree of parallelism, synchronization and communication overheads can significantly degrade GPU performance. Due to this reason, we assign each thread to a query and have threads execute independently of each other. For the B-tree layout, in order to access each node of $B = 32$ elements in a coalesced manner, each warp is assigned to a query and warp-level communication primitives are utilized to coordinate the search.

Figure A.10 (respectively Figure A.11 and Figure A.12) plots the average permutation time versus the input size, for the K40 (respectively Quadro M4000 and RTX 2080 Ti). For all three GPU platforms, the general consensus is that the modified van Emde Boas cycle-leader approach is the fastest permutation algorithm. As noted in Section 8.2, this is expected since the vEB algorithm has the lowest I/O complexity. On the K40 and Quadro M4000 GPUs, both the B-tree cycle-leader and BST involution approaches are competitive until the BST involution algorithm shows a sharp increase in runtime for $N > 2^{29} - 1$ elements. Similar to our CPU platform, the BST involutions and BST hybrid permutations show a significant runtime increase for non-perfect trees.

Figure A.13 (respectively Figure A.15 and Figure A.17) shows the average time to perform 1 million queries on each search tree layout and binary search on a sorted array versus the input size, for the K40 (respectively Quadro M4000 and RTX 2080 Ti). It is interesting to see that the query performance varies depending on the GPU platform used. On all GPU platforms, B-tree querying results in the fastest runtime for $N > 2^{28} - 1$ on the K40 and Quadro M400 and $N > 2^{23} - 1$ on the RTX 2080 Ti. On the K40 and Quadro M4000, BST querying outperforms mvEB querying; however on the RTX 2080 Ti, mvEB querying outperforms BST querying for $N \geq 2^{24} - 1$. Furthermore, on the K40 and Quadro M4000 GPUs, a large increase in runtime is observed for $N > 2^{29} - 1$. For this reason, we measure the total runtime of permuting and performing Q queries on one perfect tree and one non-perfect tree for $N \leq 2^{29} - 1$ and similarly for $N > 2^{29} - 1$.

Figure A.19, Figure A.20, and Figure A.21 shows the combined runtime of permuting and querying each search tree layout with $N = 100$ million elements on the K40, Quadro M4000, and RTX 2080 Ti, respectively. Figure A.22, Figure A.23, and Figure A.24 shows the combined runtime of permuting and querying each search tree layout with $N = 2^{29} - 1$ elements on the K40, Quadro M4000, and RTX 2080 Ti, respectively. Figure A.27, Figure A.28, and Figure A.29 shows the combined runtime of permuting and querying each search tree layout with $N = 100$ million elements on the K40, Quadro M4000, and RTX 2080 Ti, respectively. And lastly, Figure A.25, Figure A.26, and Figure 7 plot the combined runtime for $N = 2^{30} - 1$ elements on the K40, Quadro M4000, and RTX 2080 Ti, respectively. Table 8.3 summarizes the number of queries for which it becomes worthwhile to permute and query on the respective search tree layout (compared to an equal number of binary search

TABLE 2
Number of queries needed for it to be beneficial (compared to an equal number of binary search queries) to perform each of the search tree layout permutations. The B-tree layout has the lowest number of queries needed for $P = 1$; while the vEB layout performs better for $P = 20$.

Layout	$N = 2^{29} - 1$ and $P = 1$	$N = 2^{29} - 1$ and $P = 20$	$N = 1$ billion and $P = 1$	$N = 1$ billion and $P = 20$
BST	30 million (5.59% of N)	9 million (1.68% of N)	212 million (21.2% of N)	—
B-tree	8 million (1.49% of N)	6 million (1.12% of N)	52 million (5.2% of N)	170 million (17% of N)
vEB	10 million (1.86% of N)	2 million (0.37% of N)	97 million (9.7% of N)	144 million (14.4% of N)

TABLE 3
Number of queries needed for it to be beneficial (compared to an equal number of binary search queries) to perform each of the search tree layout permutations on each of our GPU platforms with $N = 100$ million (first table), $N = 2^{29} - 1$ (second table), $N = 1$ billion (third table), and $N = 2^{30} - 1$ (fourth table). On the K40 and Quadro M4000, the B-tree layout has the lowest number of queries needed; while on the RTX 2080 Ti, the modified van Emde Boas (mvEB) layout beats both the B-tree and BST layouts.

Layout	K40	Quadro M4000	RTX 2080 Ti
BST	34 million (34% of N)	47 million (47% of N)	83 million (83% of N)
B-tree	13 million (13% of N)	20 million (20% of N)	23 million (23% of N)
mvEB	—	44 million (44% of N)	13 million (13% of N)

Layout	K40	Quadro M4000	RTX 2080 Ti
BST	62 million (11.55% of N)	65 million (12.11% of N)	293 million (54.58% of N)
B-tree	45 million (8.38% of N)	55 million (10.24% of N)	119 million (22.17% of N)
mvEB	127 million (23.66% of N)	106 million (19.74% of N)	50 million (9.31% of N)

Layout	K40	Quadro M4000	RTX 2080 Ti
BST	71 million (7.1% of N)	53 million (5.3% of N)	936 million (93.6% of N)
B-tree	14 million (1.4% of N)	14 million (1.4% of N)	194 million (19.4% of N)
mvEB	20 million (2% of N)	32 million (3.2% of N)	103 million (10.3% of N)

Layout	K40	Quadro M4000	RTX 2080 Ti
BST	62 million (5.77% of N)	62 million (5.77% of N)	596 million (55.51% of N)
B-tree	14 million (1.3% of N)	15 million (1.4% of N)	217 million (20.21% of N)
mvEB	19 million (1.77% of N)	32 million (2.98% of N)	96 million (8.94% of N)

queries). For non-perfect trees ($N = 100$ million and $N = 1$ billion), we see a significant increase in the percentage of queries needed for the BST layout, mainly due to the high cost of permutation. For $N = 100$ million on the K40, the mvEB layout is never worth the cost of permuting because binary search is faster than querying the mvEB layout until $N \geq 2^{27} - 1$. Overall, for all input sizes, the B-tree layout results in the lowest number of queries needed on the K40 and Quadro M4000 GPUs, as it has both the fastest permutation and query runtimes. However, on the RTX 2080 Ti, the mvEB is the best performing layout, due to it having the fastest permutation runtime and second fastest query runtime. We note that for a large number of queries on the RTX 2080 Ti, we expect the better query performance of the B-tree layout to eventually overcome the faster permutation runtime of the mvEB layout.

9 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 utilize the cache hierarchy effectively. We measure the performance of our algorithms on both CPU and GPU platforms, and key results are as follows.

On the CPU, permuting a sorted array of $N = 2^{29} - 1$ 64-bit elements into the BST, B-tree, and vEB layouts, and then searching for Q 64-bit queries, all in parallel, outperforms a parallel binary search on the original sorted array when $Q \geq 9$ million (1.68% of N), $Q \geq 6$ million (1.12% of N), and $Q \geq 2$ million (0.37% of N), respectively. On the latest GPU hardware available to us, the Nvidia RTX 2080 Ti GPU, the same experiments on an array of $N = 2^{30} - 1$ 32-bit elements outperform binary search when $Q \geq 596$ million (55.51% of N), $Q \geq 217$ million (20.21% of N), and $Q \geq 96$ million (8.94% of N), for the BST, B-tree, and mvEB layouts, respectively.

This work underscores the importance of I/O-efficiency when designing parallel algorithms for modern manycore hardware. The development of efficient memory layouts, beyond searching, provides fertile ground for future research.

ACKNOWLEDGMENTS

We would like to thank Michael Bender for his helpful suggestions.

REFERENCES

[1] P.-V. Khuong and P. Morin, "Array layouts for comparison-based searching," *J. Exp. Algorithmics*, vol. 22, pp. 1.3:1–1.3:39, 2017.
[2] A. Bradford, *The Investment Industry for IT Practitioners*, 2008.
[3] W. Inmon, D. Strauss, and G. Neushloss, *DW 2.0: The Architecture for the Next Generation of Data Warehousing*, 2010.
[4] F. 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*, vol. 113, pp. 37 – 54, 2018.
[5] G. 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*, 2002, pp. 39–48.
[6] R. E. Ladner, R. Fortna, and B.-H. Nguyen, "A comparison of cache aware and cache oblivious static search trees using program instrumentation," in *Experimental Algorithmics*, 2002, pp. 78–92.
[7] J. JaJa, *Introduction to Parallel Algorithms*, 1992.
[8] R. P. Brent, "The parallel evaluation of general arithmetic expressions," *Journal of the ACM*, vol. 21, no. 2, pp. 201–206, 1974.
[9] L. Arge, M. T. Goodrich, M. Nelson, and N. Sitchinava, "Fundamental parallel algorithms for private-cache chip multiprocessors," in *Proc. of 20th ACM SPAA*, 2008, pp. 197–206.
[10] A. Aggarwal and J. S. Vitter, "The input/output complexity of sorting and related problems," *Commun. ACM*, vol. 31, no. 9, pp. 1116–1127, 1988.
[11] Y.-J. Chiang, "Experiments on the practical I/O efficiency of geometric algorithms: Distribution sweep vs. plane sweep," in *Algorithms and Data Structures*, 1995, pp. 346–357.
[12] D. Ajwani and N. Sitchinava, "Empirical evaluation of the parallel distribution sweeping framework on multicore architectures," in *European Symposium on Algorithms*, 2013, pp. 25–36.
[13] R. Bayer and E. McCreight, "Organization and maintenance of large ordered indices," in *Proc. of ACM SIGFIDET Workshop on Data Description, Access and Control*, 1970, pp. 107–141.
[14] H. Prokop, "Cache-oblivious algorithms," Master's thesis, MIT, 1999.
[15] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in *40th FOCS*, 1999, pp. 285–297.
[16] F. E. Fich, J. I. Munro, and P. V. Poblete, "Permuting in place," *SIAM Journal on Computing*, vol. 24, no. 2, pp. 266–278, 1995.
[17] Q. Yang, J. Ellis, K. Mamakani, and F. Ruskey, "In-place permuting and perfect shuffling using involutions," *Information Processing Letters*, vol. 113, no. 10, pp. 386 – 391, 2013.
[18] P. Diaconis, R. Graham, and W. Kantor, "The mathematics of perfect shuffles," *Advances in Applied Mathematics*, vol. 4, no. 2, pp. 175 – 196, 1983.
[19] C. Ronse, "A generalization of the perfect shuffle," *Discrete Mathematics*, vol. 47, pp. 293 – 306, 1983.
[20] H. S. Stone, "Parallel processing with the perfect shuffle," *IEEE Transactions on Computers*, vol. C-20, no. 2, pp. 153–161, 1971.
[21] D. D'Angeli and A. Donno, "Shuffling matrices, kronecker product and discrete fourier transform," *Discrete Appl. Math*, vol. 233, pp. 1 – 18, 2017.
[22] M. Davio, "Kronecker products and shuffle algebra," *IEEE Transactions on Computers*, vol. C-30, no. 2, pp. 116–125, 1981.
[23] S. F. Sultana and D. Shubhangi, "Video encryption algorithm and key management using perfect shuffle," *International Journal of Engineering Research and Applications*, vol. 07, pp. 01–05, 2017.
[24] J. Ellis and M. Markov, "In-situ, stable merging by way of the perfect shuffle," *The Computer Journal*, vol. 43, no. 1, pp. 40–53, 2000.
[25] M. Dalkilic, E. Haytaoglu, and G. Tokatli, "A simple shuffle-based stable in-place merge algorithm," *Proc. Computer Science*, vol. 3, pp. 1049 – 1054, 2011.
[26] J. Ellis and U. Stege, "A provably, linear time, in-place and stable merge algorithm via the perfect shuffle," *CoRR*, 2015.
[27] J. Ellis, T. Krahn, and H. Fan, "Computing the cycles in the perfect shuffle permutation," *Information Processing Letters*, vol. 75, no. 5, pp. 217 – 224, 2000.
[28] J. Ellis, H. Fan, and J. Shallit, "The cycles of the multiway perfect shuffle permutation." *Discrete Mathematics & Theoretical Computer Science*, vol. 5, no. 1, pp. 169–180, 2002.
[29] P. Jain, "A simple in-place algorithm for in-shuffle," *CoRR*, 2008.
[30] D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, 1998.
[31] J. S. Vitter, "Algorithms and data structures for external memory," *Found. Trends Theor. Comput. Sci.*, vol. 2, no. 4, pp. 305–474, 2008.
[32] K. Berney, H. Casanova, A. Higuchi, B. Karsin, and N. Sitchinava, "Beyond binary search: Parallel in-place construction of implicit search tree layouts," in *International Parallel and Distributed Processing Symposium*, 2018, pp. 1070–1079.
[33] OpenMP Architecture Review Board, "OpenMP application program interface version 3.0," 2008. [Online]. Available: http://www.openmp.org/mp-documents/spec30.pdf
[34] NVIDIA, "CUDA programming guide 10.1," 2019. [Online]. Available: http://docs.nvidia.com/cuda

Kyle Berney Kyle Berney is a Ph.D. candidate in the Department of Information and Computer Sciences at the University of Hawaii at Manoa. His research interests are in parallel algorithms, cache-efficient algorithms, and GPGPU computing.

Henri Casanova Dr. Henri Casanova is a Professor in the Department of Information and Computer Sciences at the University of Hawaii at Manoa. His research is in the broad area of high performance computing, and in particular the scheduling and the simulation of parallel and distributed applications. He obtained his Ph.D. from the University of Tennessee, Knoxville in 1998.

Ben Karsin Dr. Ben Karsin is an engineer at Nvidia Corp. He obtained his Ph.D. in the Department of Information and Computer Sciences at the University of Hawaii at Manoa in 2018.

Nodari Sitchinava Dr. Nodari Sitchinava is an Associate Professor in the Department of Information and Computer Sciences at the University of Hawaii at Manoa. His research interests are in parallel and cache-efficient algorithms. He obtained his Ph.D. in Computer Science from the University of California, Irvine in 2009.

APPENDIX A
FIGURES FOR EXPERIMENTAL RESULTS

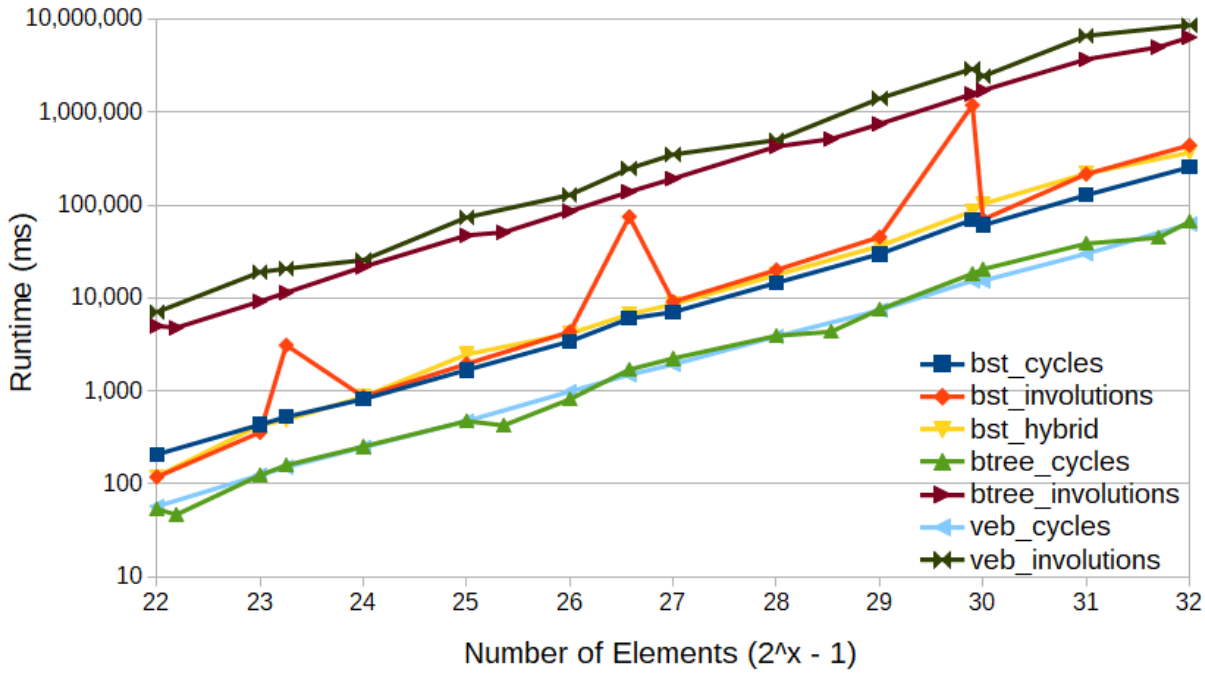

Fig. A.1. Average time to permute a sorted array using each permutation algorithm on the CPU using 1 thread. The graph is displayed on a log-log scale.

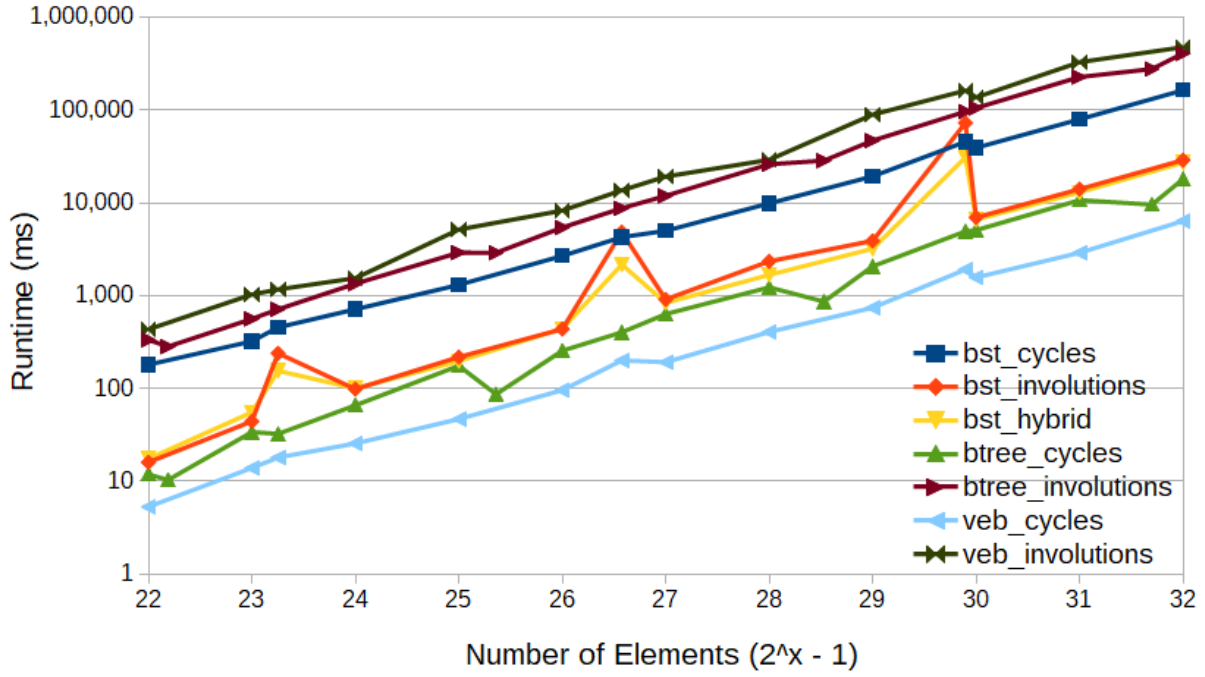

Fig. A.2. Average time to permute a sorted array using each permutation algorithm on the CPU using 20 threads. The graph is displayed on a log-log scale.

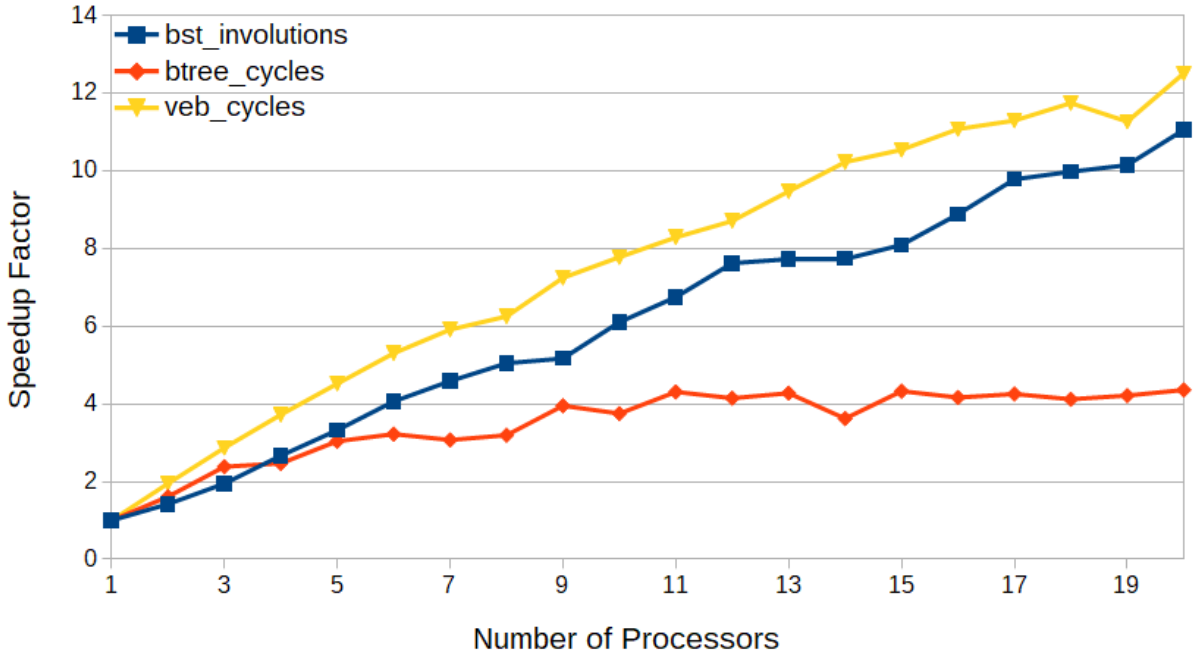

Fig. A.3. Speed-up factor of the permutation algorithms on the CPU for $N = 2^{29} - 1$.

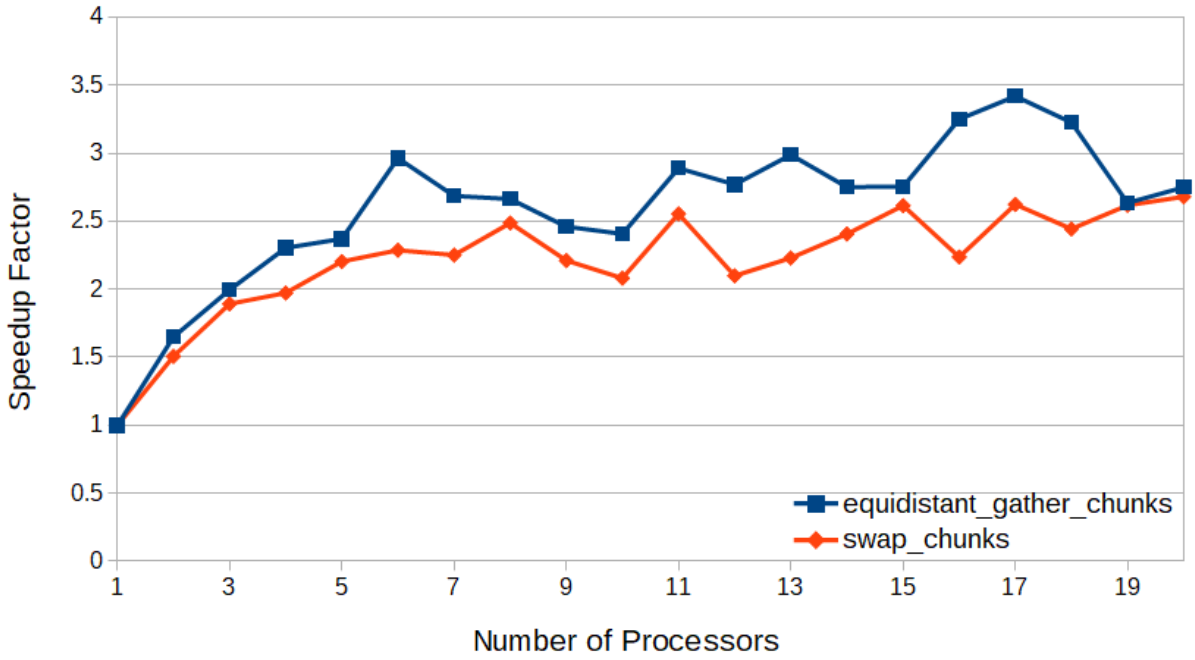

Fig. A.4. Speed-up factor of the extended equidistant gather on chunks of elements and swapping the first half of an array with the second half on the CPU.

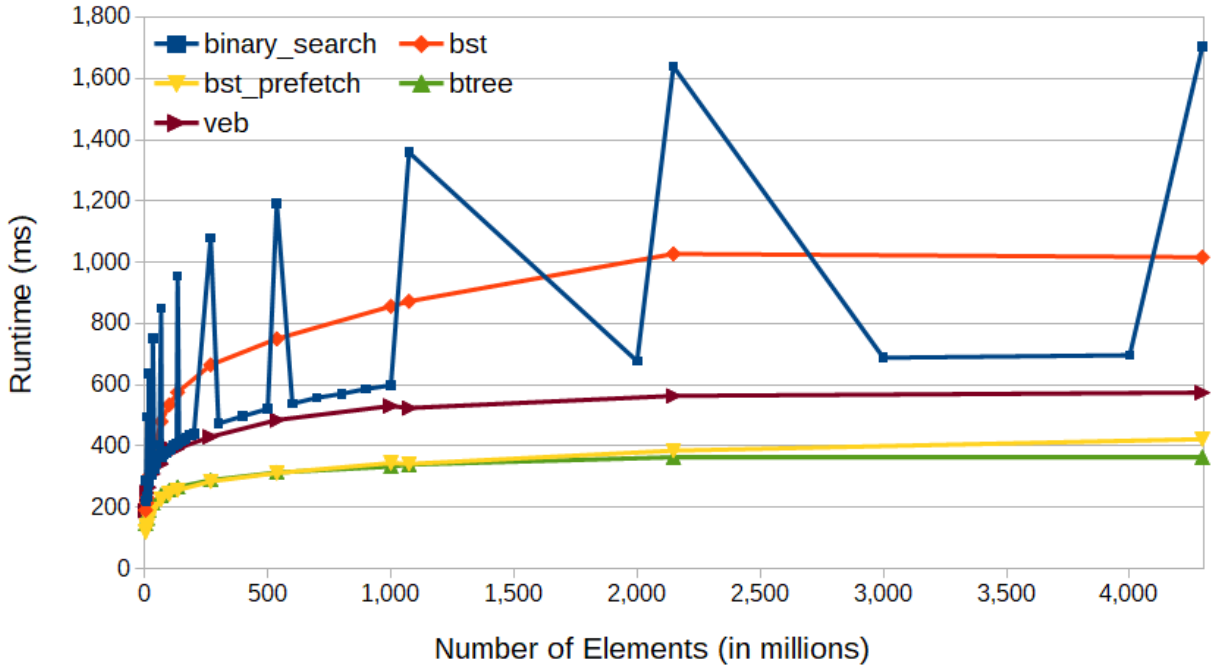

Fig. A.5. Average time to perform 1 million queries on each search tree layout on the CPU for varying array size. The spikes in runtime for binary search occurs when the array size is close to a power of 2 due to cache-aliasing issues. The graph is displayed on a linear scale to emphasize the logarithmic shape of the querying.

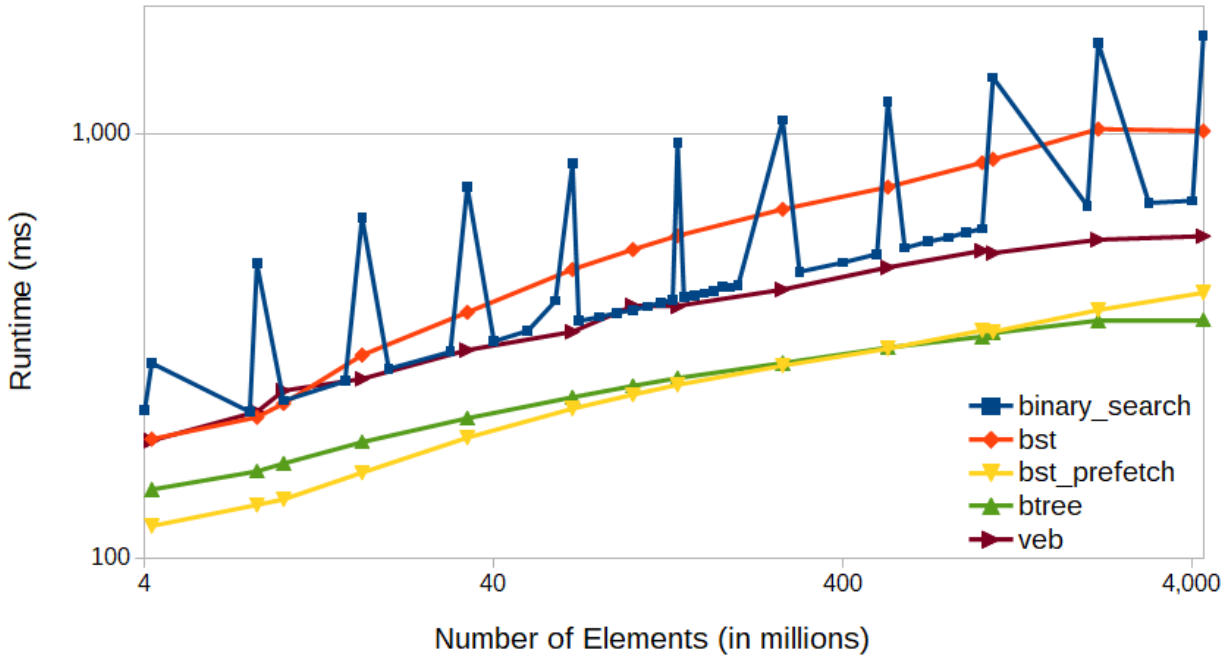

Fig. A.6. Average time to perform 1 million queries on each search tree layout on the CPU for varying array size. The spikes in runtime for binary search occurs when the array size is close to a power of 2 due to cache-aliasing issues. The graph is displayed on a log-log scale to see the comparison of results between different array layouts.

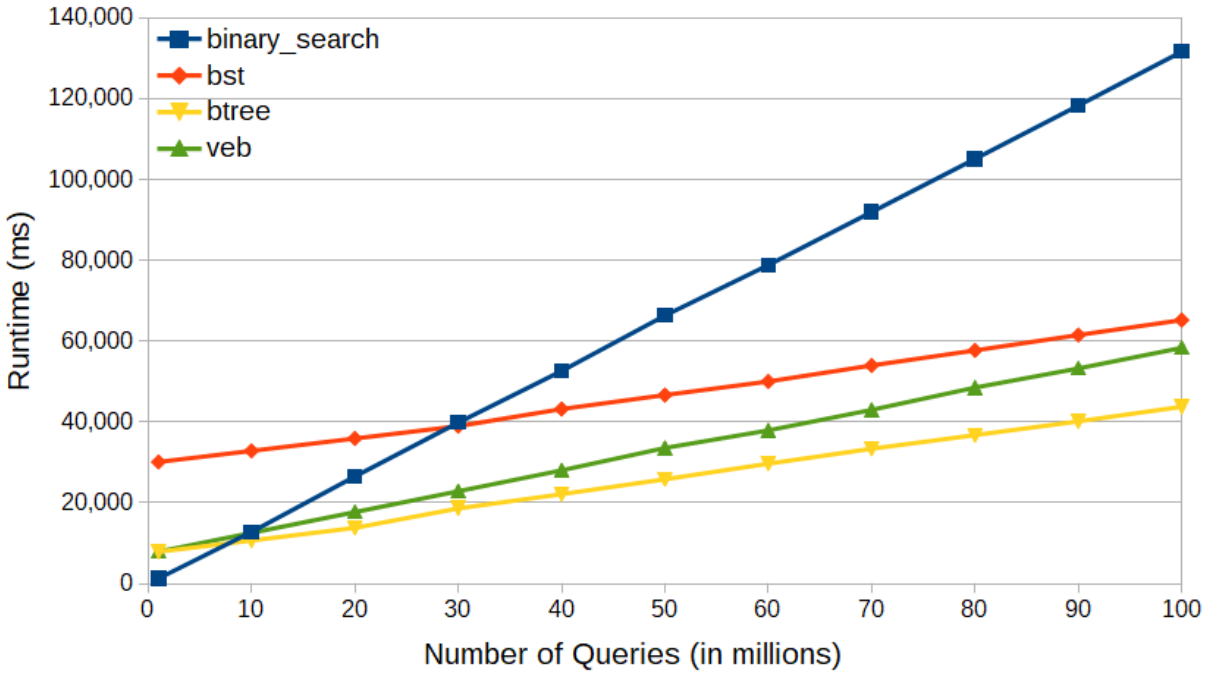

Fig. A.7. Combined time of permuting and performing Q queries on an array of $N = 2^{29} - 1$ elements on the CPU using 1 thread.

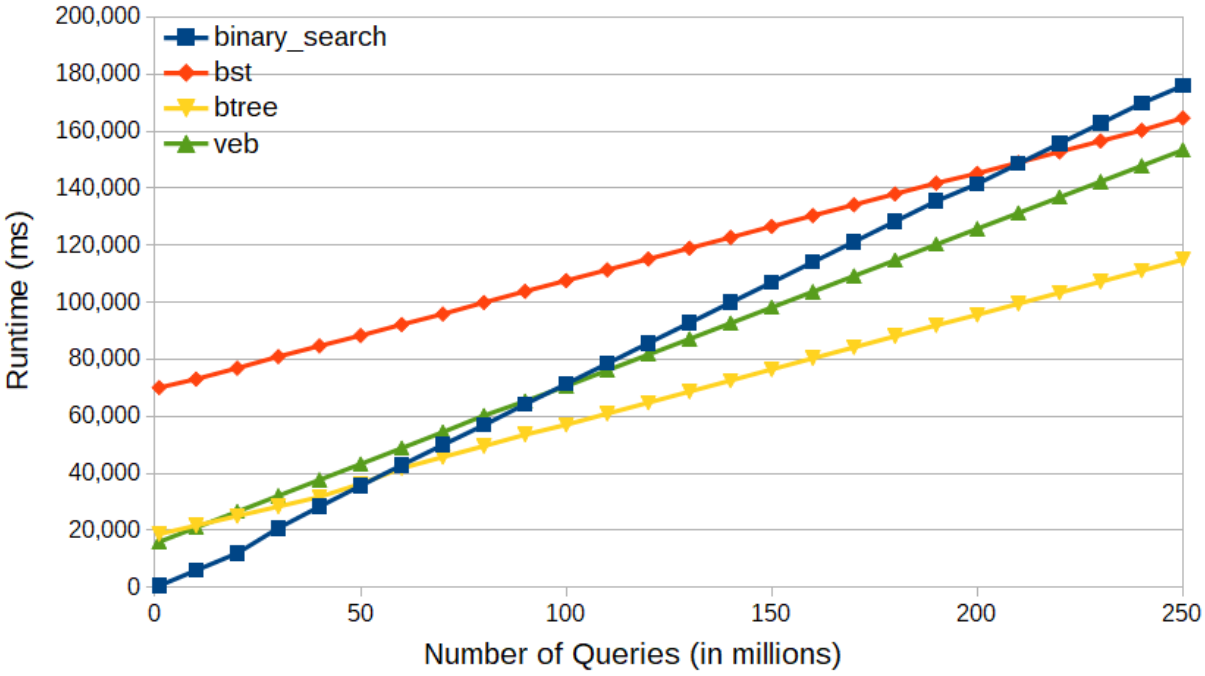

Fig. A.8. Combined time of permuting and performing Q queries on an array of 1 billion elements on the CPU using 1 thread.

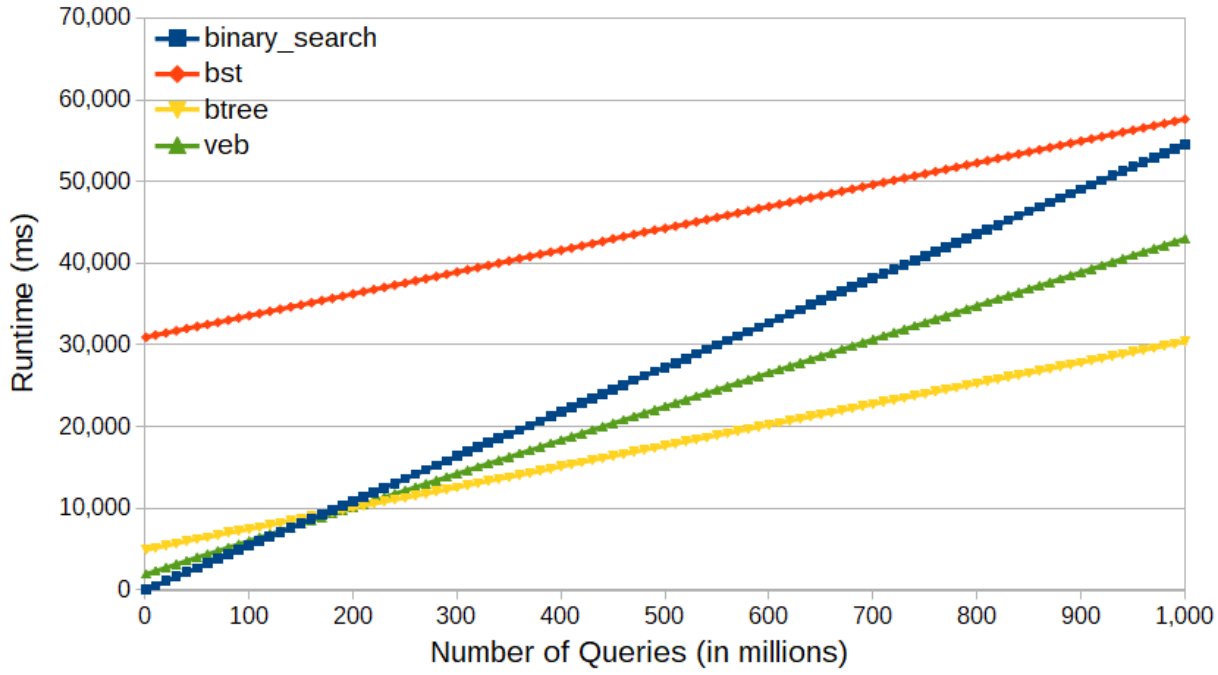

Fig. A.9. Combined time of permuting and performing Q queries on an array of 1 billion elements on the CPU using 20 threads.

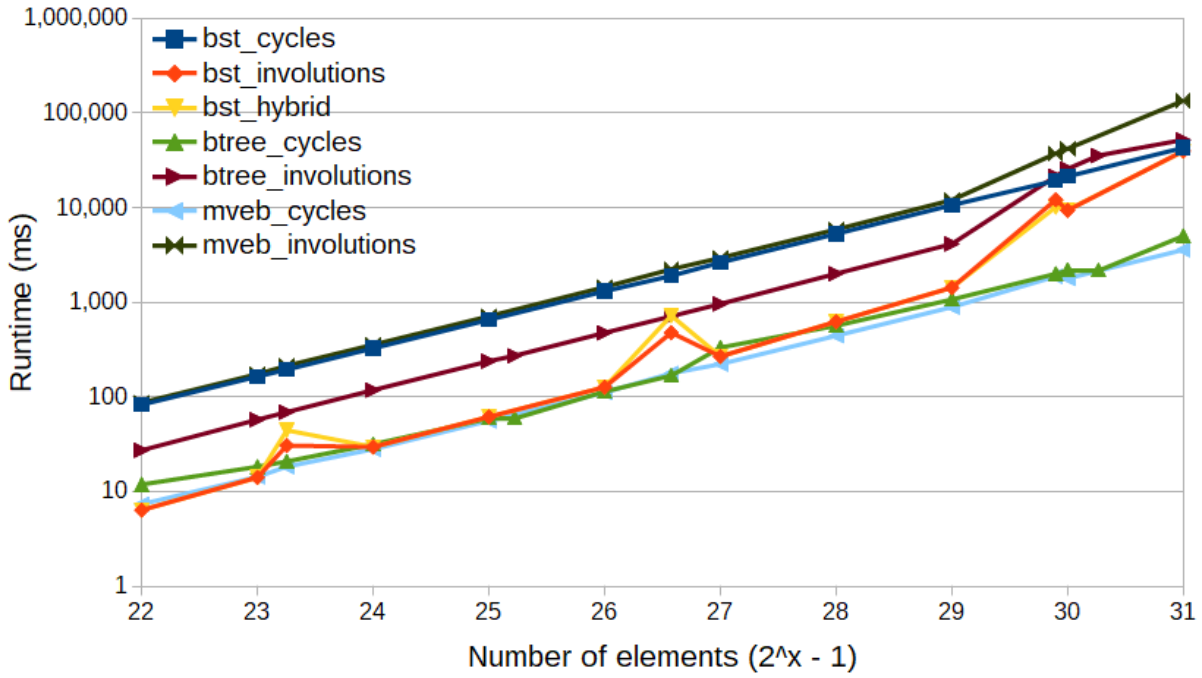

Fig. A.10. Average time to permute a sorted array using each permutation algorithm on the Nvidia K40. The graph is displayed on a log-log scale.

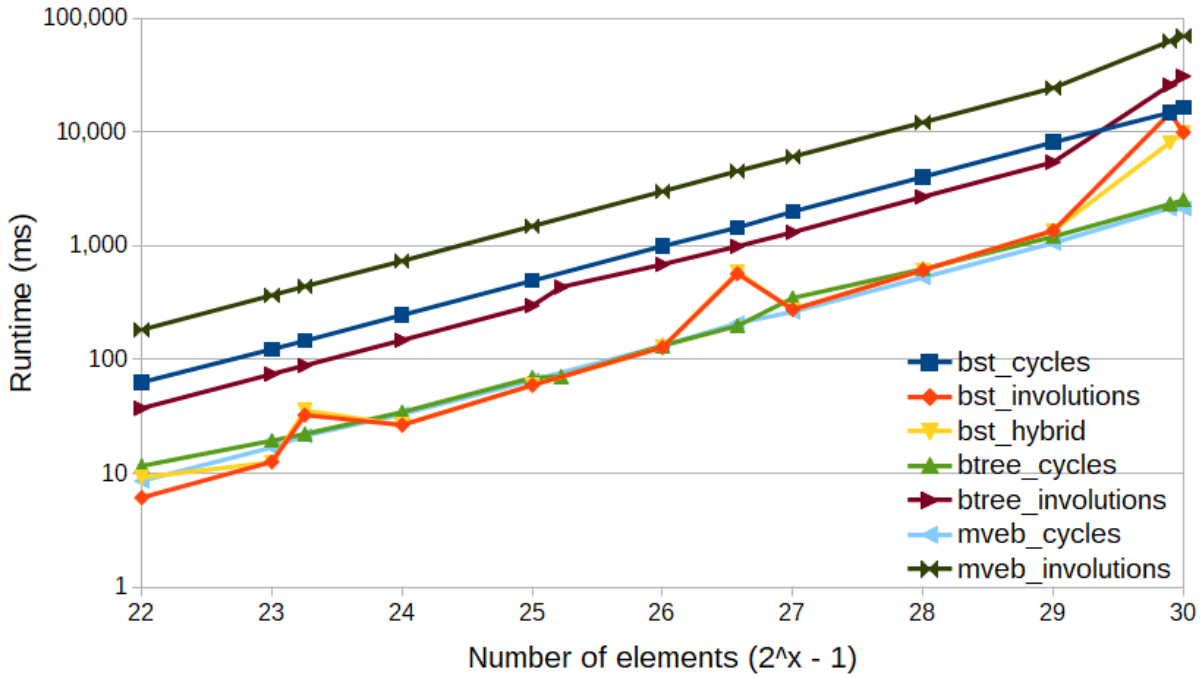

Fig. A.11. Average time to permute a sorted array using each permutation algorithm on the Nvidia Quadro M4000. The graph is displayed on a log-log scale.

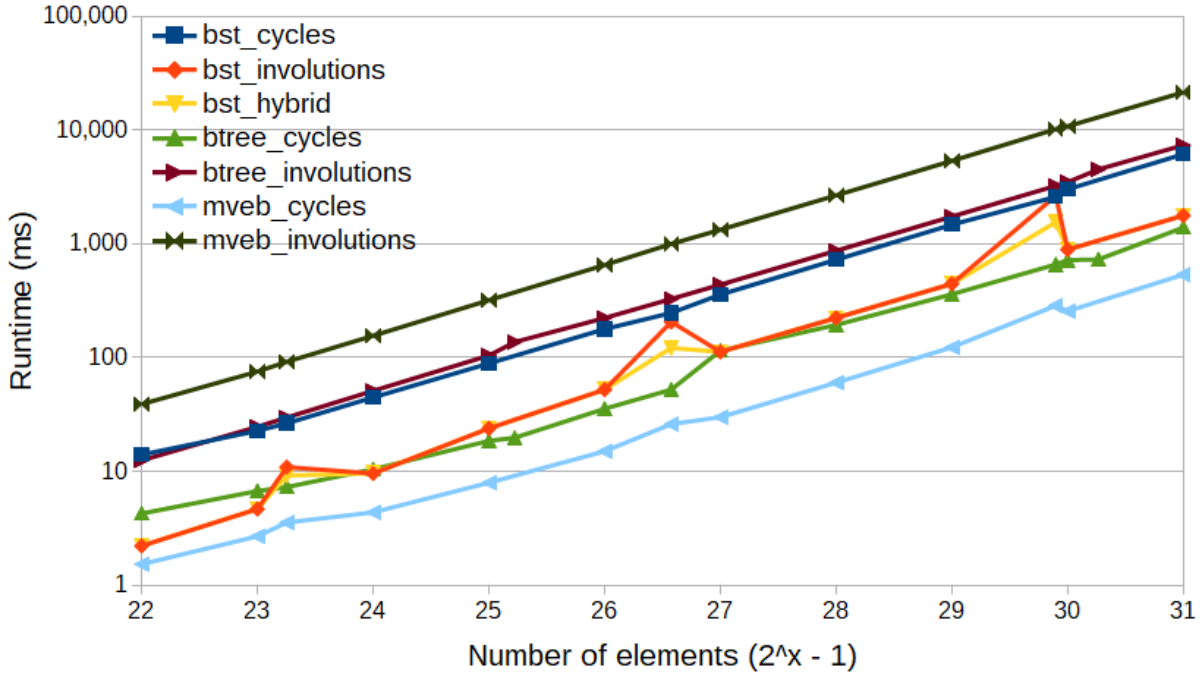

Fig. A.12. Average time to permute a sorted array using each permutation algorithm on the Nvidia GeForce RTX 2080 Ti. The graph is displayed on a log-log scale.

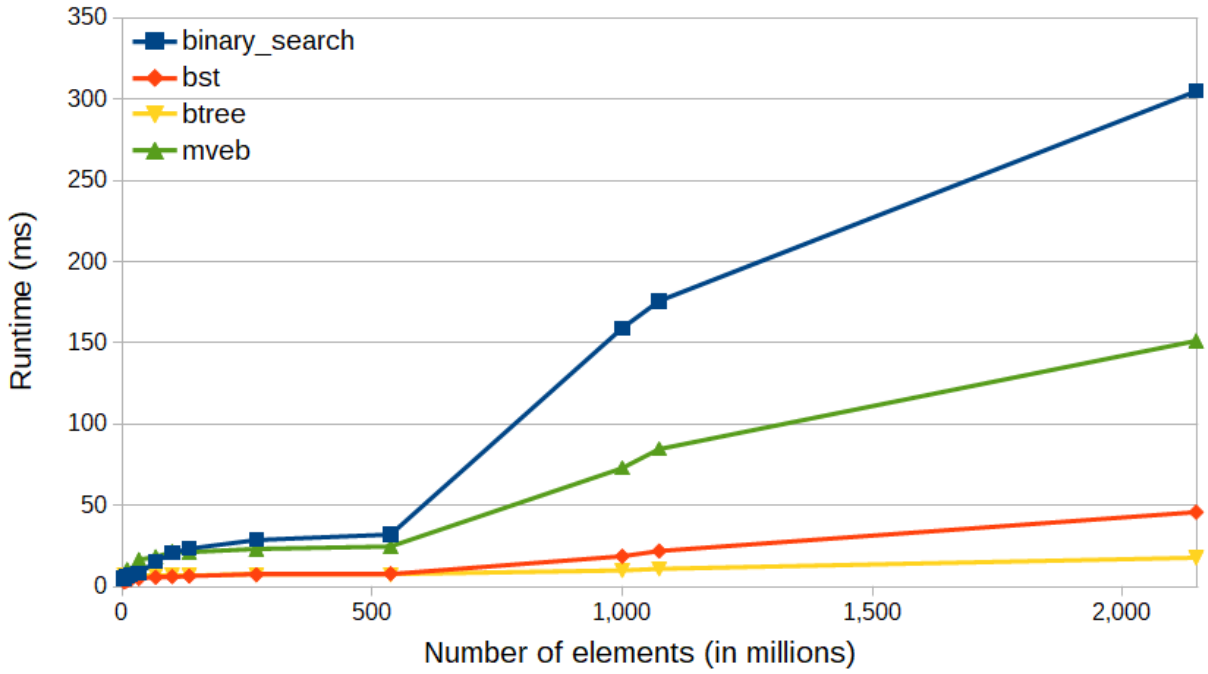

Fig. A.13. Average time to perform 1 million queries on each search tree layout and binary search on a sorted array on the Nvidia K40. The graph is displayed on a linear scale to emphasize the logarithmic shape of the querying.

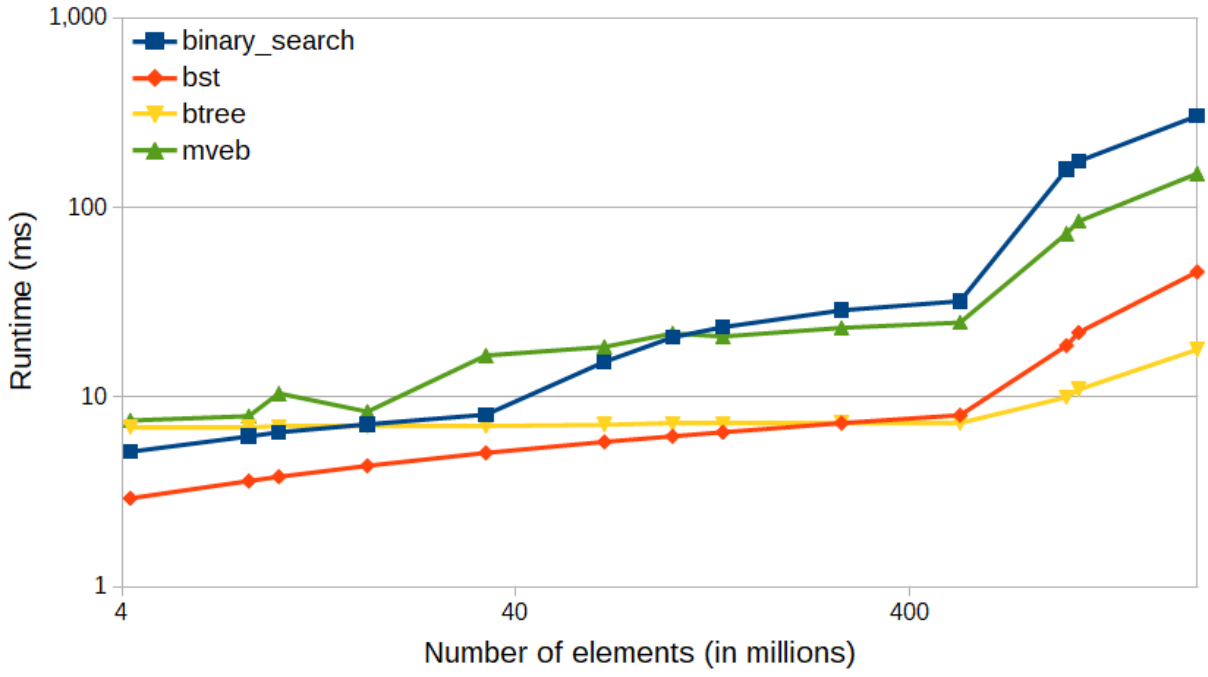

Fig. A.14. Average time to perform 1 million queries on each search tree layout and binary search on a sorted array on the Nvidia K40. The graph is displayed on a log-log scale to see the comparison of results between different array layouts.

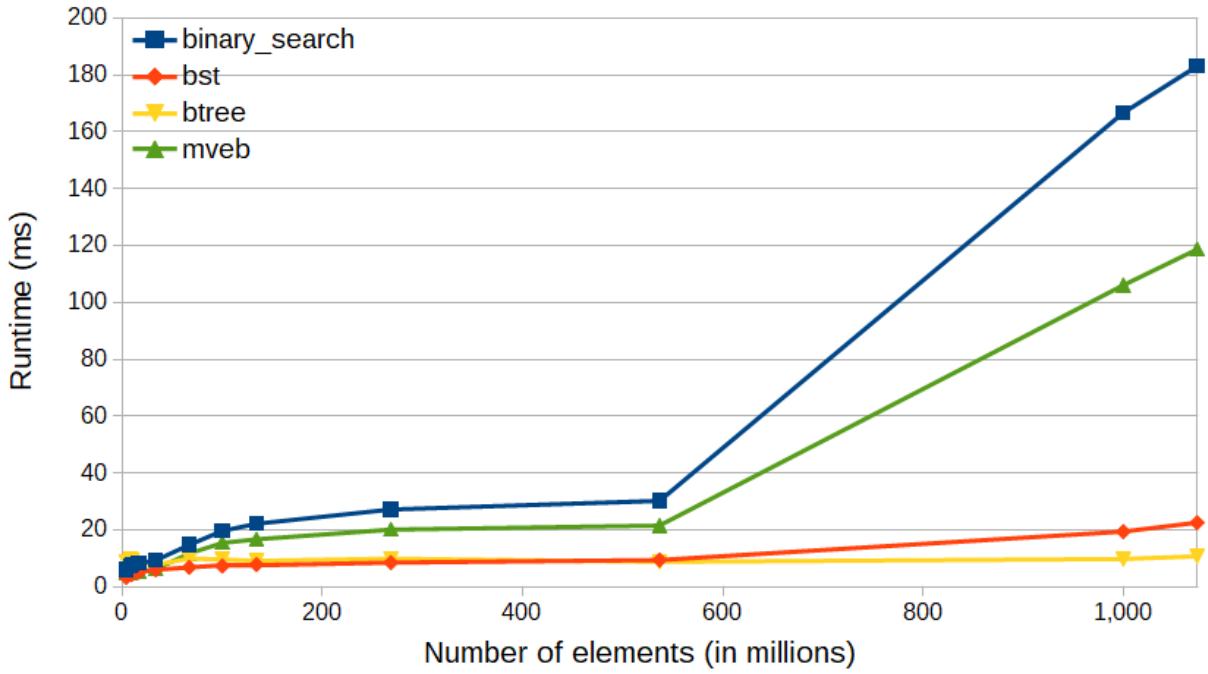

Fig. A.15. Average time to perform 1 million queries on each search tree layout and binary search on a sorted array on the Nvidia Quadro M4000. The graph is displayed on a linear scale to emphasize the logarithmic shape of the querying.

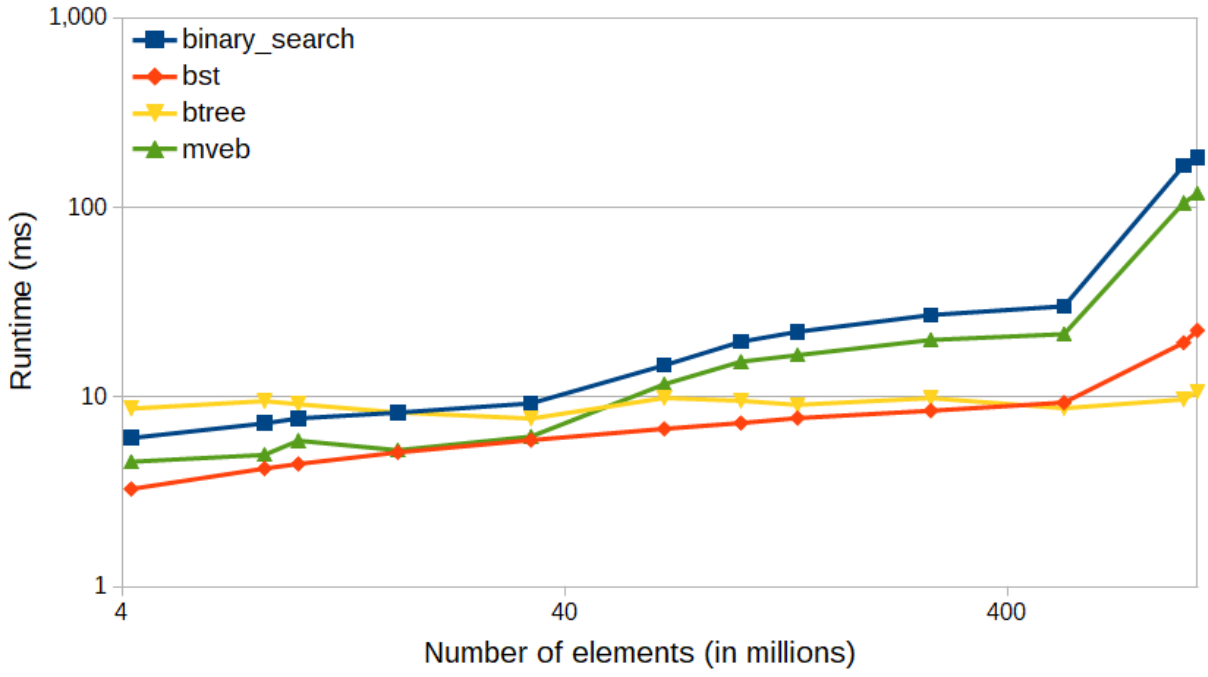

Fig. A.16. Average time to perform 1 million queries on each search tree layout and binary search on a sorted array on the Nvidia Quadro M4000. The graph is displayed on a log-log scale to see the comparison of results between different array layouts.

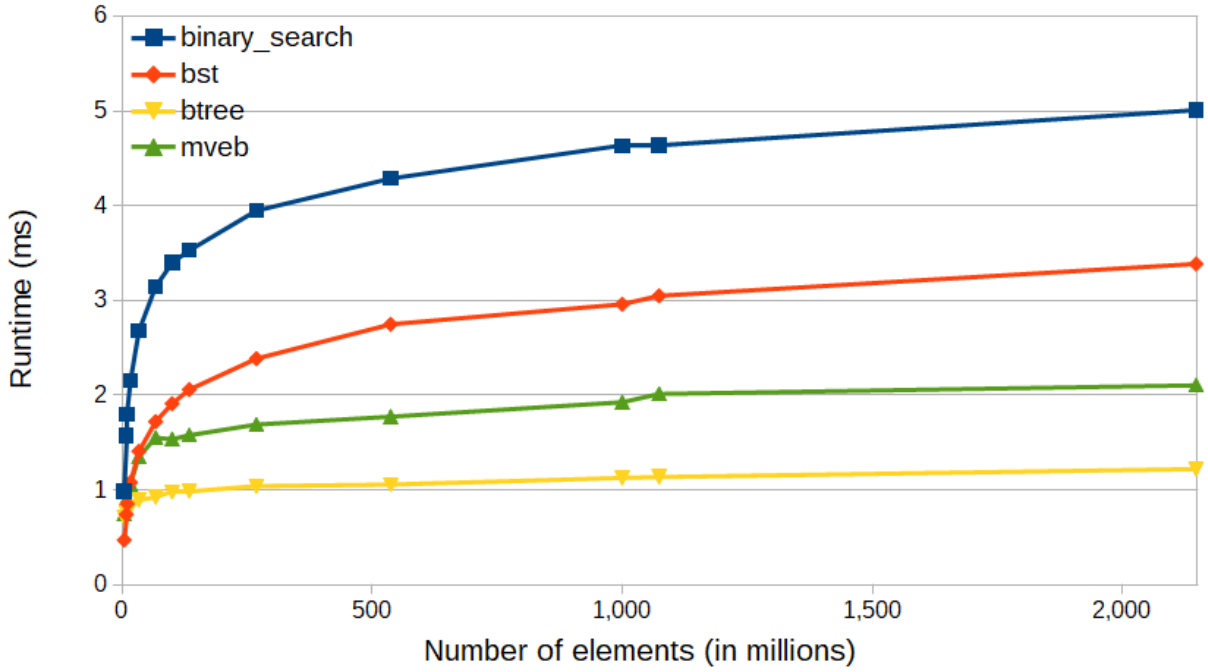

Fig. A.17. Average time to perform 1 million queries on each search tree layout and binary search on a sorted array on the Nvidia GeForce RTX 2080 Ti. The graph is displayed on a linear scale to emphasize the logarithmic shape of the querying.

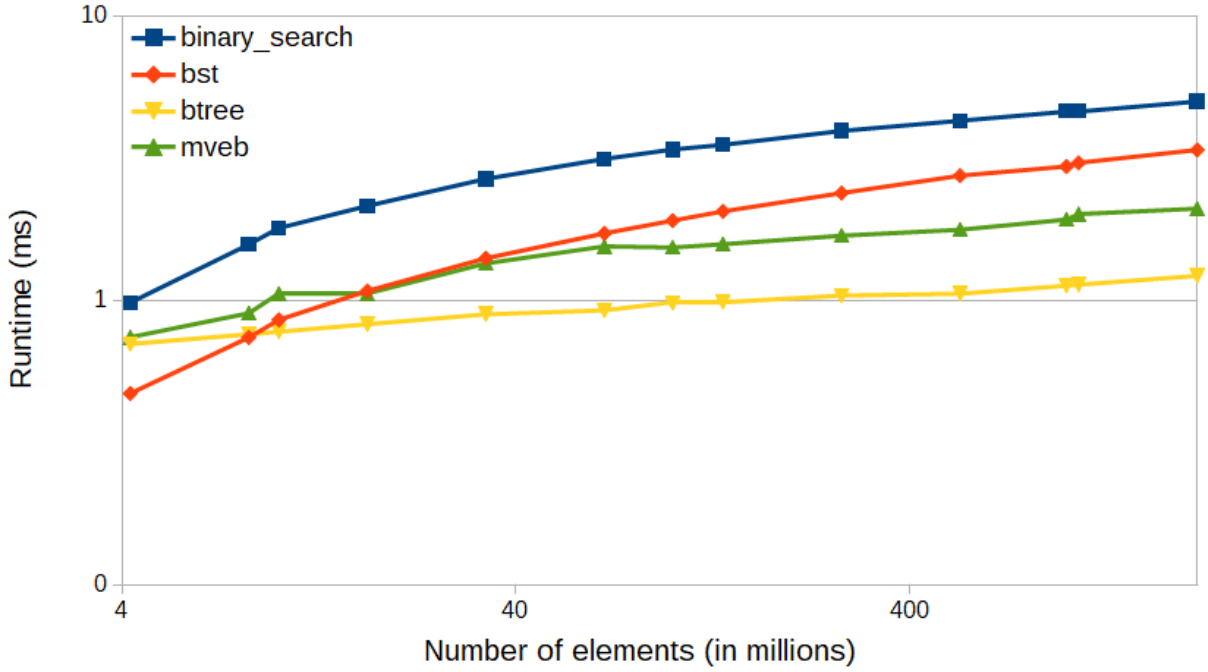

Fig. A.18. Average time to perform 1 million queries on each search tree layout and binary search on a sorted array on the Nvidia GeForce RTX 2080 Ti. The graph is displayed on a log-log scale to see the comparison of results between different array layouts.

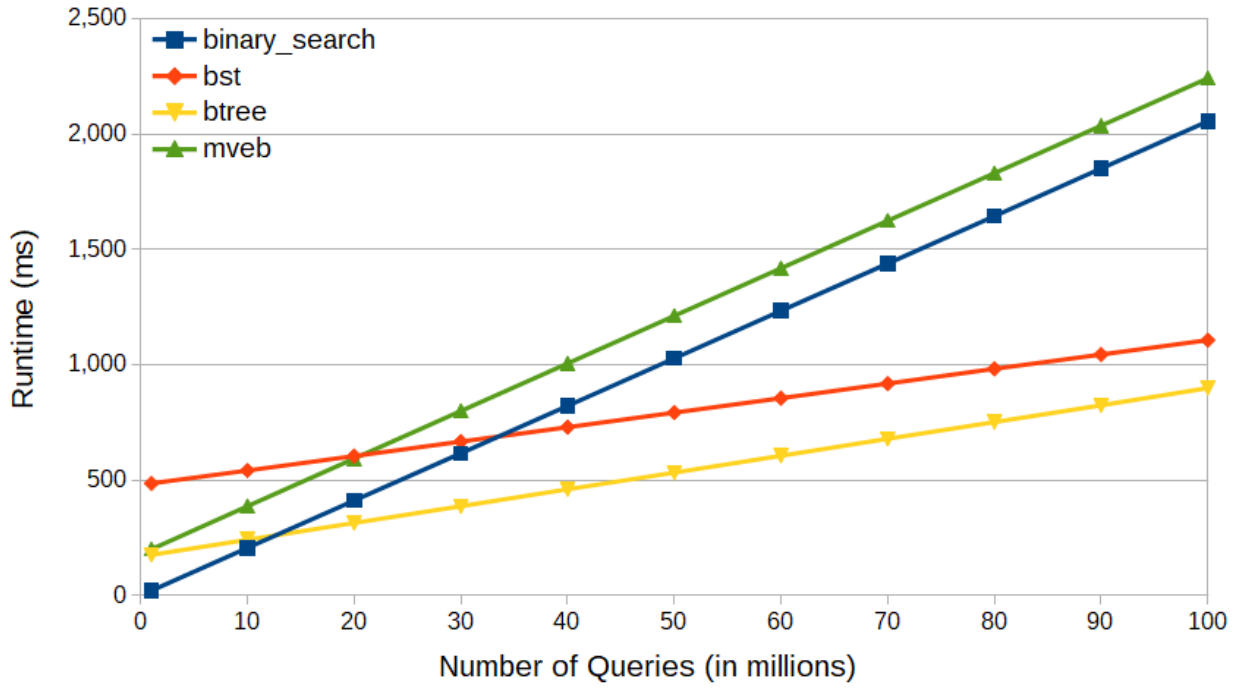

Fig. A.19. Combined time to permute and query each layout on the Nvidia K40 with $N = 100$ million elements.

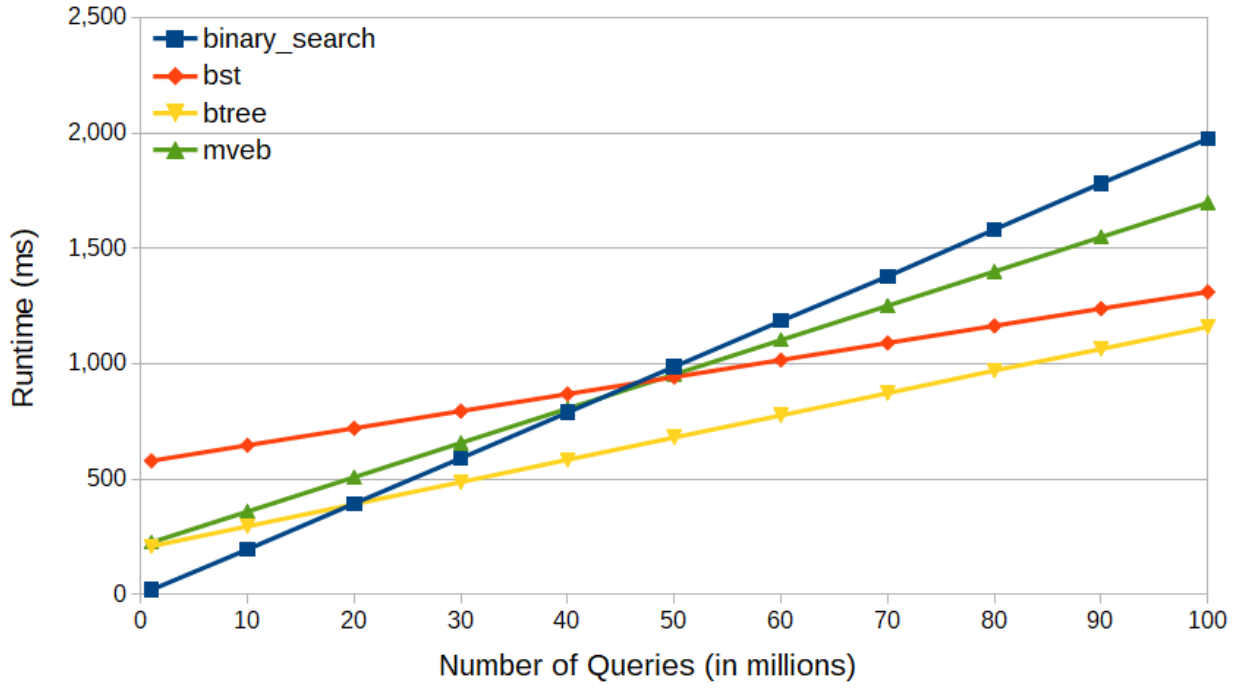

Fig. A.20. Combined time to permute and query each layout on the Nvidia Quadro M4000 with $N = 100$ million elements.

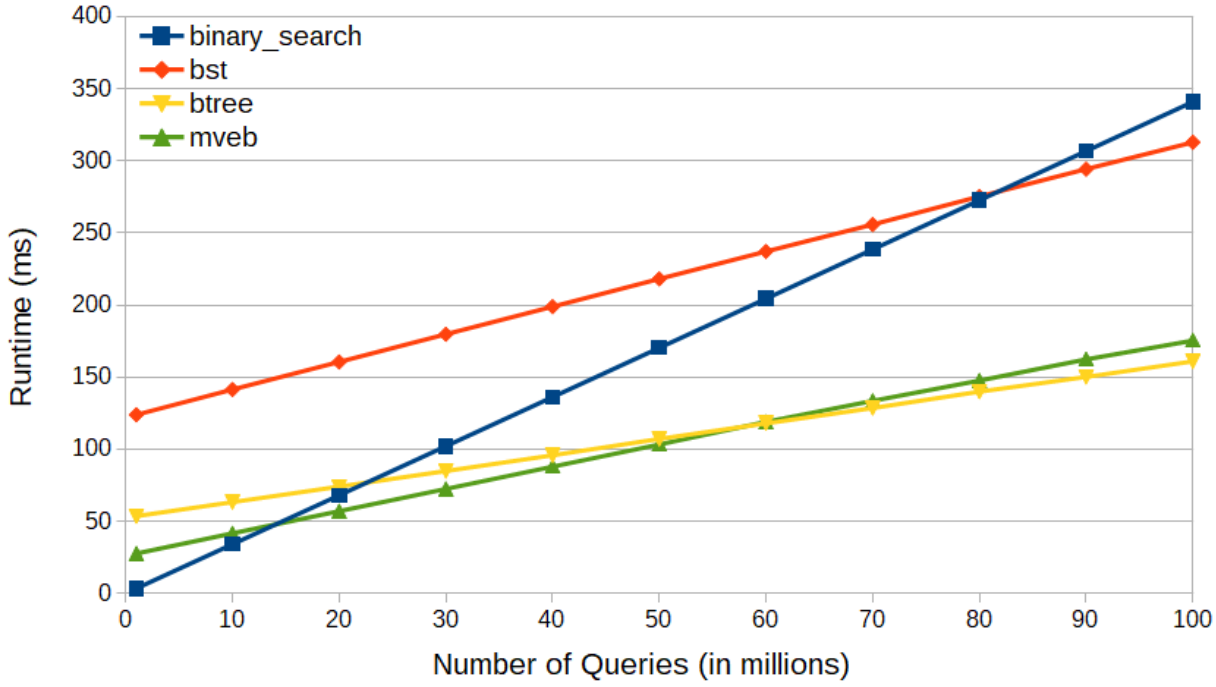

Fig. A.21. Combined time to permute and query each layout on the Nvidia RTX 2080 Ti with $N = 100$ million elements.

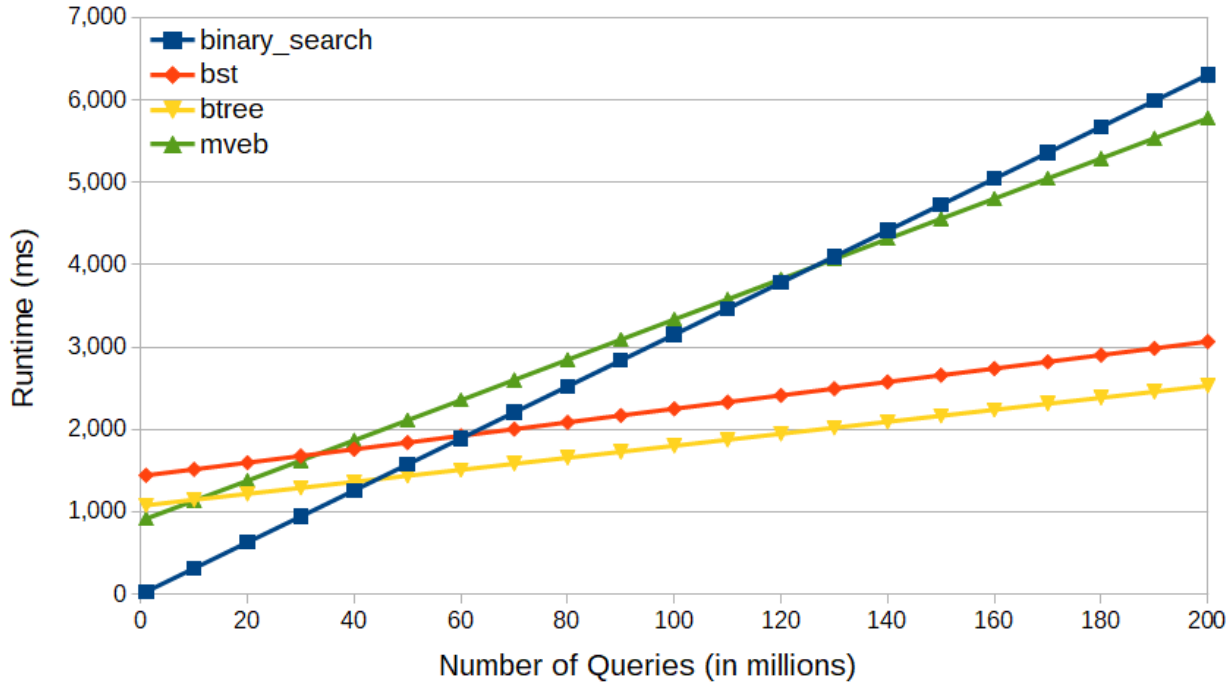

Fig. A.22. Combined time to permute and query each layout on the Nvidia K40 with $N = 2^{29} - 1$ elements.

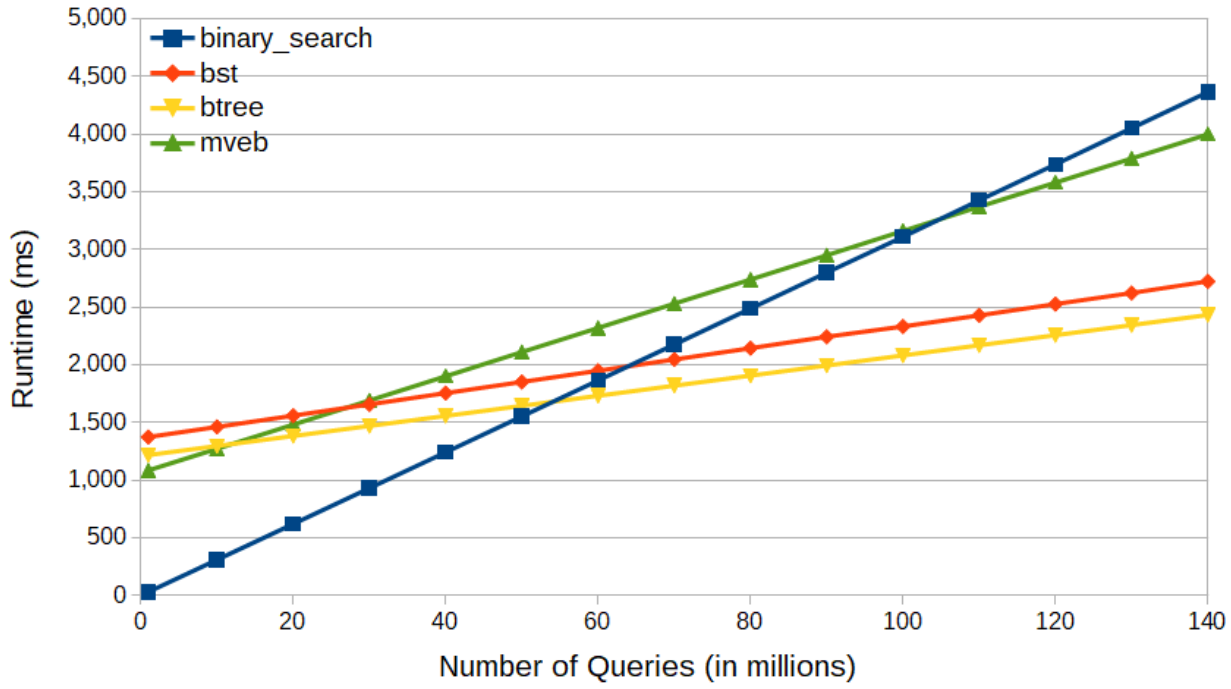

Fig. A.23. Combined time to permute and query each layout on the Nvidia Quadro M4000 with $N = 2^{29} - 1$ elements.

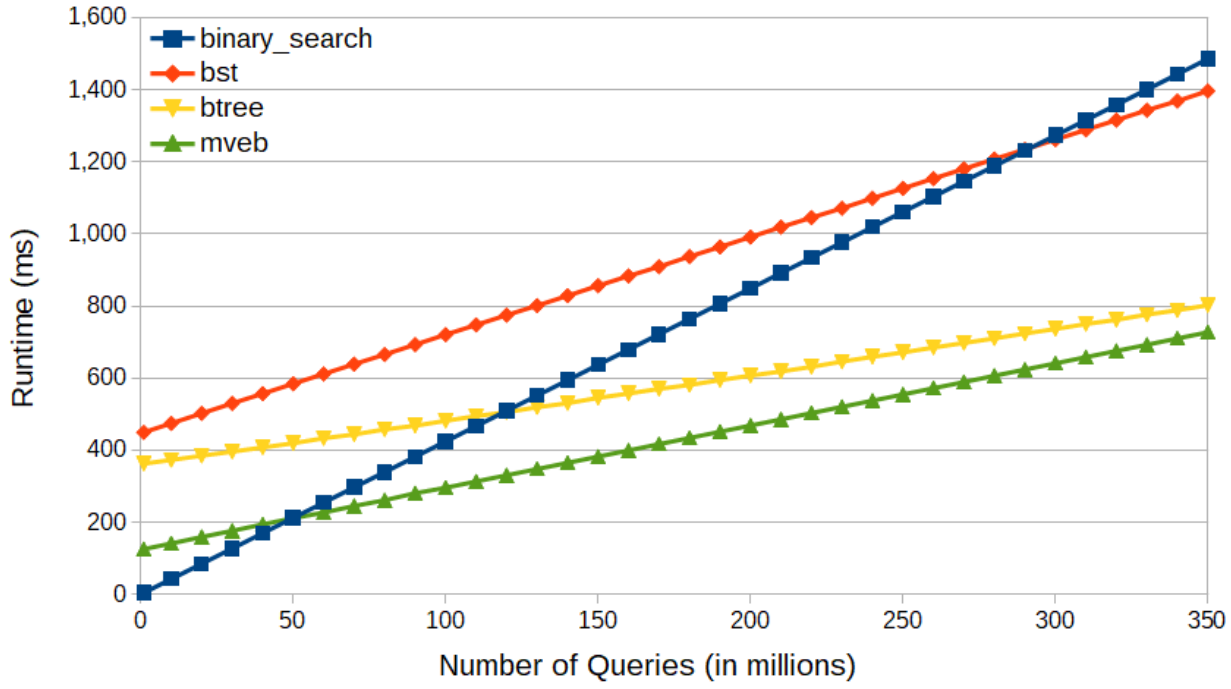

Fig. A.24. Combined time to permute and query each layout on the Nvidia RTX 2080 Ti with $N = 2^{29} - 1$ elements.

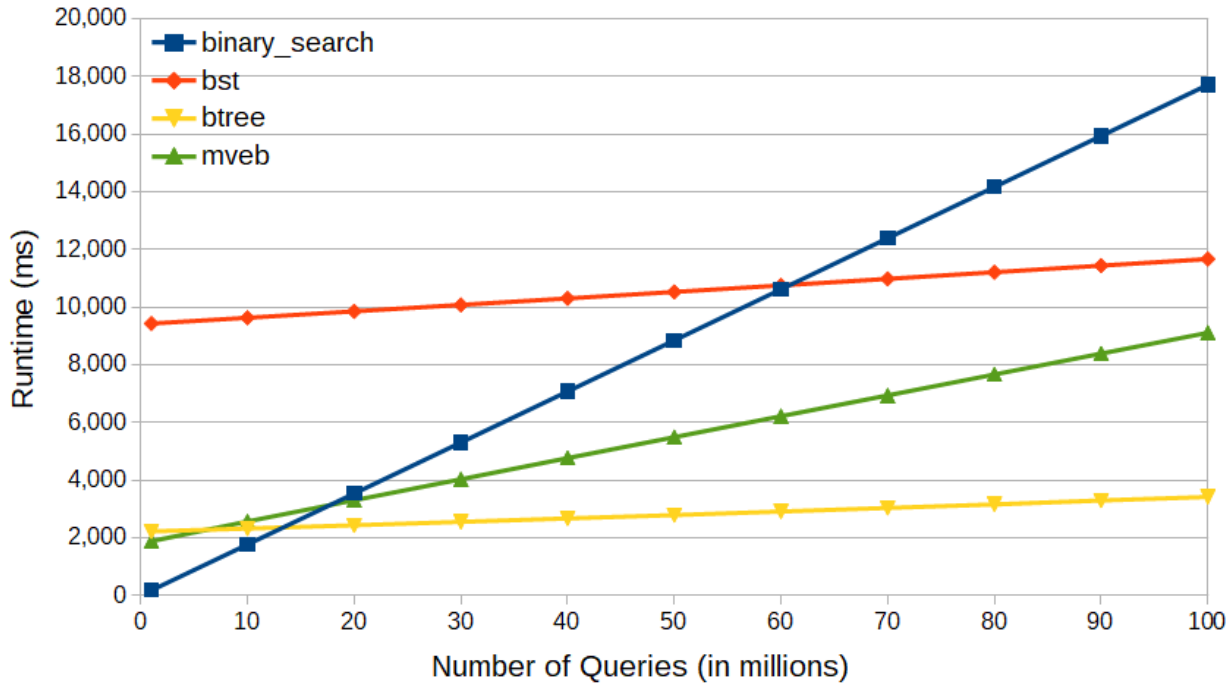

Fig. A.25. Combined time to permute and query each layout on the Nvidia K40 with $N = 2^{30} - 1$ elements.

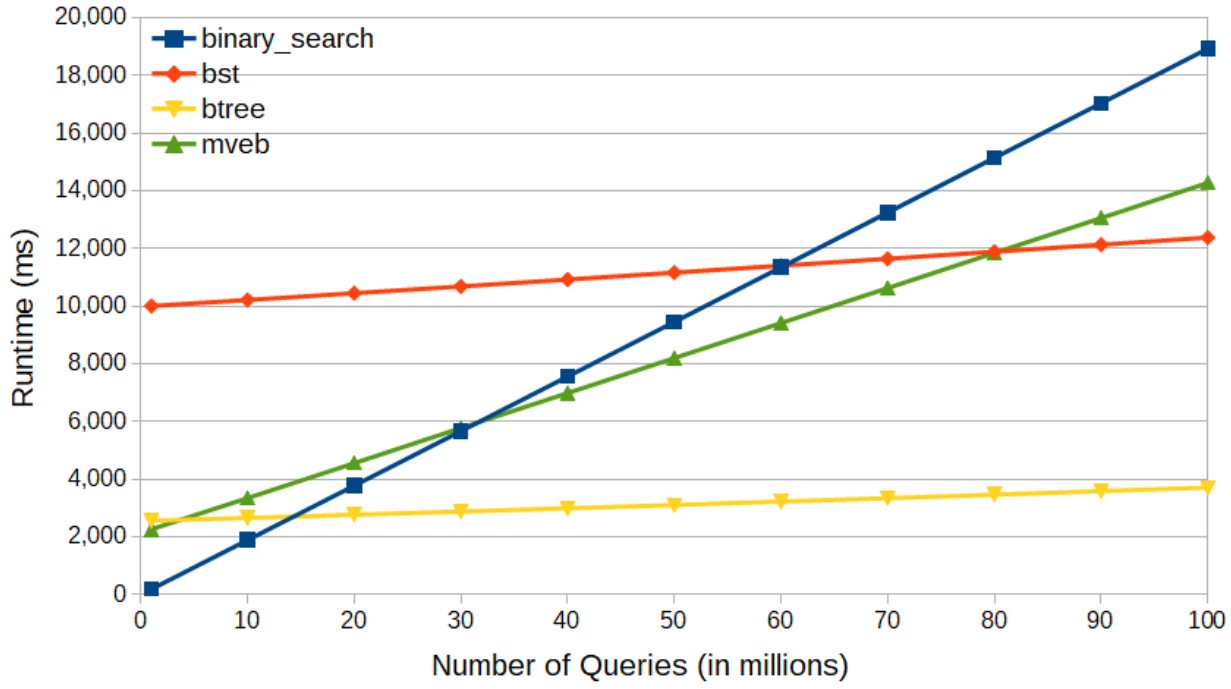

Fig. A.26. Combined time to permute and query each layout on the Nvidia Quadro M4000 with $N = 2^{30} - 1$ elements.

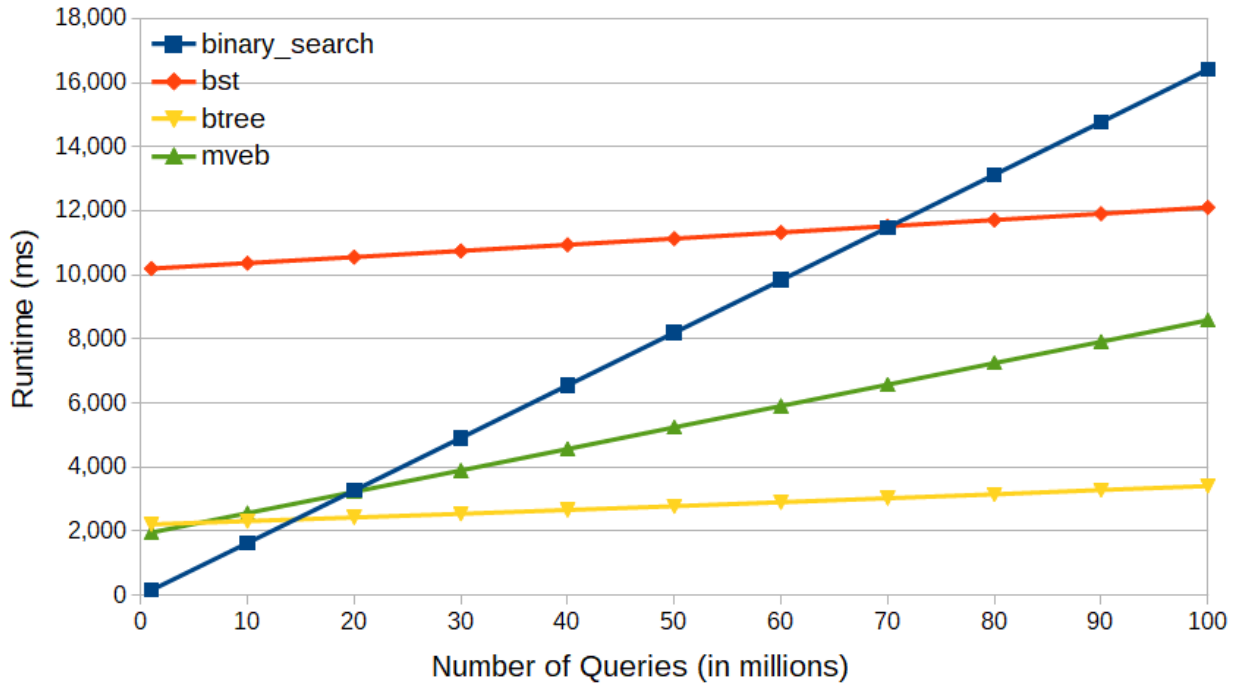

Fig. A.27. Combined time to permute and query each layout on the Nvidia K40 with $N = 1$ billion elements.

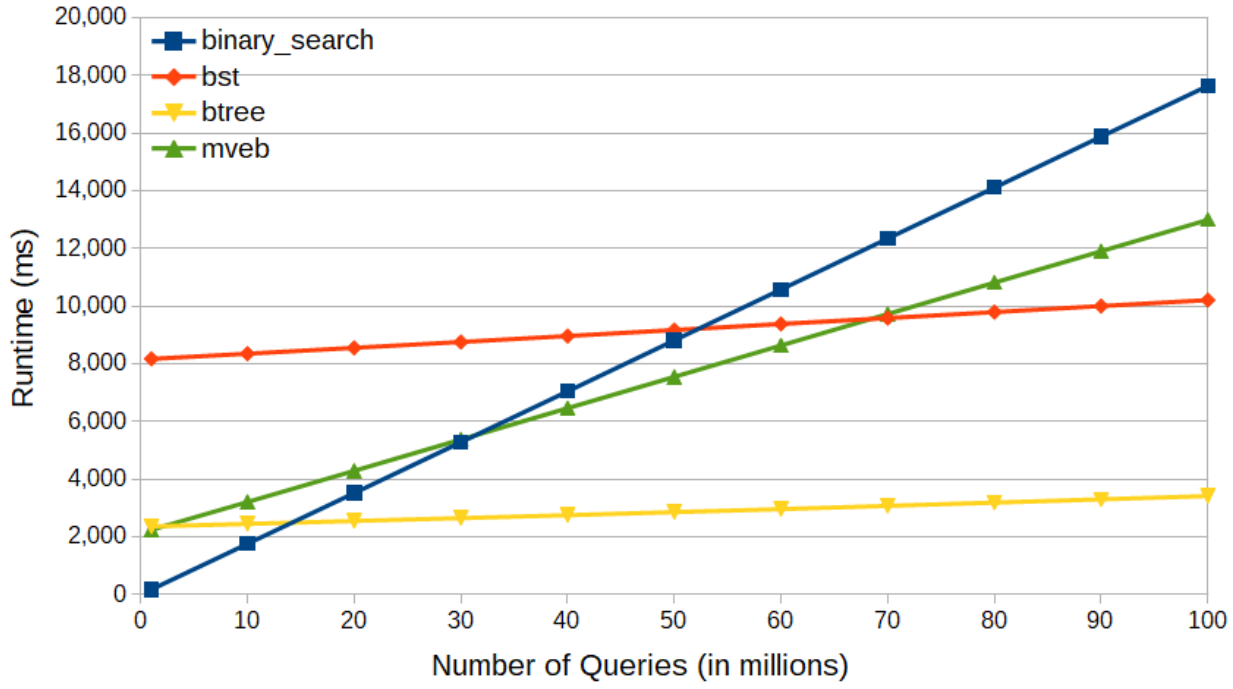

Fig. A.28. Combined time to permute and query each layout on the Nvidia Quadro M4000 with $N = 1$ billion elements.

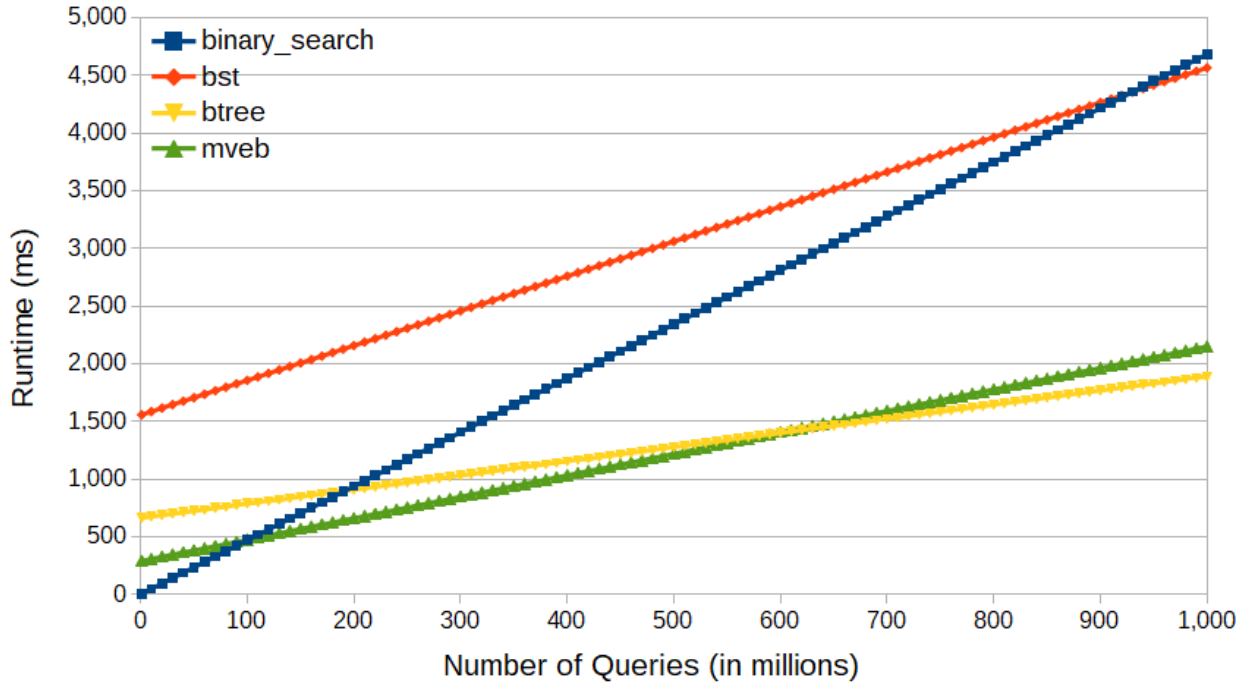

Fig. A.29. Combined time to permute and query each layout on the Nvidia RTX 2080 Ti with $N = 1$ billion elements.