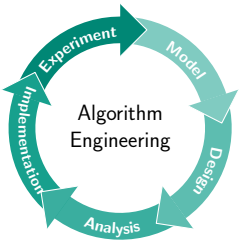


Scalable Parallel Packed Memory Arrays

Master's Thesis of

Moritz Timo Potthoff

at the Department of Informatics
Institute of Theoretical Informatics, Algorithm Engineering

Reviewer:	Prof. Dr. rer. nat. Peter Sanders
Second reviewer:	T.T.-Prof. Dr. rer. nat. Thomas Bläsius
Advisor:	M.Sc. Marvin Williams

January 16, 2023 – July 17, 2023

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself. I have submitted neither parts of nor the complete thesis as an examination elsewhere. I have not used any other than the aids that I have mentioned. I have marked all parts of the thesis that I have included from referenced literature, either in their original wording or paraphrasing their contents. This also applies to figures, sketches, images and similar depictions, as well as sources from the internet.

Karlsruhe, July 17, 2023

. . . . . .

(Moritz Timo Potthoff)

Acknowledgments

First and foremost, I want to thank my advisor Marvin Williams for guiding me through my discovery of Packed Memory Arrays and parallel data structures, for creating a pleasant working atmosphere, for joint puzzling on the challenge of the week, and motivating words when they were needed. No less, my gratitude goes to Prof. Dr. Peter Sanders for introducing me to the world of algorithm engineering in many lectures, for suggesting the topic of this thesis, for valuable inputs during the development of the new data structures, and for providing the infrastructure that enabled the experimental evaluation. Finally, I am grateful for my family and my girlfriend for supporting me during my studies and the work on this thesis.

Abstract

We present two new data structures to efficiently maintain dynamic sorted sequences. Our approach is based on the Packed Memory Array (PMA). PMAs offer cache efficient scan queries by allocating all elements in contiguous memory. During updates, they have to *rebalance* elements within the array, requiring amortized $O(\log^2 N)$ time. Our approach stores elements in *buffer blocks*, and employs rebalancing on the level of block references, rather than individual elements. In this way, our first contribution, the *Buffered Packed Memory Array* (BPMA) improves the cost for sequential, single-element updates to amortized $O(\log N)$ time, bridging the gap to traditional pointer-based data structures.

Our second contribution, the *Batch-Parallel Buffered Packed Memory Array* (BBPMA), offers update operations that insert or delete a *batch* of elements or keys in parallel. It builds on the design of the BPMA but adds effective work-balancing for both inserting the elements and rebalancing the data structure in parallel. In this way, inserting a batch of k elements is possible in amortized span $O(\log N + \log k)$.

Experimental evaluation for the BBPMA shows that it offers batch-parallel insertions that are faster by a factor of 1.2–2.5 for regular inputs and a factor of up to 25 for worst-case inputs, than a comparable batch-parallel PMA without the block indirection. At the same time, we show that our block indirection has small effects on scan performance. In the configuration that offers the fastest insertions, scans in the BBPMA are between 30 % faster and up to 35 % slower than in the traditional PMA, but 2 times faster than in a search tree. With a configuration that focuses on scan performance, the BBPMA dominates the traditional PMA: Scans in the BBPMA are a factor of 1.25–2 faster, while it reaches similar insertion performance.

Deutsche Zusammenfassung

Diese Arbeit entwickelt zwei neue Datenstrukturen, um dynamische sortierte Folgen effizient zu verarbeiten. Sie basieren auf dem Packed Memory Array (PMA), einer Datenstruktur, die besonders effizient Teile der sortierten Folge scannen kann, da sie alle Elemente in einem kontinuierlichen Speicherblock alloziert. Für Einfügungen oder Löschungen müssen sie Elemente im Speicher *rebalancieren*, sodass eine Änderung amortisiert $O(\log^2 N)$ Zeit benötigt. Die Datenstrukturen, die in dieser Arbeit vorgestellt werden, speichern Elemente in *Pufferblöcken* und rebalancieren auf dem Niveau von Pufferblöcken, anstatt einzelner Elemente. Auf diese Weise verbessert die erste Datenstruktur — das *Buffered Packed Memory Array* (BPMA) — die amortisierten Kosten für sequentielle Änderungen einzelner Elemente auf $O(\log N)$. Damit ist sie asymptotisch gleich schnell wie zeigerbasierte Suchbaumdatenstrukturen.

Die zweite Datenstruktur, das *Batch-Parallel Buffered Packed Memory Array* (BBPMA), bietet Änderungsoperationen, die einen ganzen Satz von Elementen oder Schlüsseln auf einmal in die Datenstruktur einfügt oder aus ihr löscht. Sie baut auf dem BPMA auf und erweitert es um effektive Lastverteilung sowohl für das Einfügen neuer Elemente als auch das Rebalancieren der Datenstruktur. Dies ermöglicht die effiziente *parallele* Verarbeitung einer Operation. Auf diese Weise ermöglicht sie es, einen Satz von k Elementen in optimaler paralleler Bearbeitungszeit von $O(\log N + \log k)$ einzufügen.

Die experimentelle Evaluation des BBPMA zeigt, dass die Einfügeoperationen um einen Faktor von 1.2 bis 2.5 für reguläre Eingaben und bis zu 25 für ungünstige Eingaben schneller sind als in einem vergleichbaren PMA ohne Pufferblöcke. Gleichzeitig haben die Pufferblöcke einen geringen Einfluss auf die Scan-Performance. In einer Konfiguration, die die schnellsten Einfügungen ermöglicht, sind Scans im BBPMA zwischen 30 % schneller und 35 % langsamer als in einem PMA ohne Pufferblöcke, aber um einen Faktor 2 schneller als in Suchbäumen. Mit einer Konfiguration, die auf optimale Scan-Performance ausgelegt ist, dominiert das BBPMA traditionelle PMAs: Scans im BBPMA sind einen Faktor von 1.25 bis 2 schneller, während Einfügungen ähnlich schnell verarbeitet werden.

Contents

1. Introduction

Data structures for sorted sequences are the foundation of various applications that need to handle large dynamic data sets [2, 26]. They simultaneously enable fast update and lookup queries on the data set by storing the elements in sorted order [24]. Search trees are the most common representation if elements can only be compared to each other, and no knowledge about the keys can be used [2, 24]. However, *Packed Memory Arrays* (PMAs) [21, 9, 4] are a more efficient data structure in practice for some use cases like representing dynamic graphs [29, 28, 26, 15], or as a building block of (concurrent) cache-oblivious search trees [9, 4, 11, 10, 6, 5].

PMAs store the elements of a sorted sequence in a single, contiguous array. This layout enables memory-efficient scan operations as data can be prefetched efficiently from contiguous memory, allowing fast memory accesses. Opposed to that, scans in pointer-based data structures like search trees require many expensive random accesses [26]. Therefore, PMAs are an ideal candidate for use cases that require fast scans on dynamic data sets. One example are graph algorithms, as they often scan all edges of a node with fast subroutines for each edge [29].

While storing data in a contiguous array benefits the performance of scan queries, it complicates updates, i.e., insertions and deletions. An update in a naive sorted array requires worst-case linear time for moving *all* existing elements, whereas search trees achieve a running time of $O(\log N)$. To improve update times, PMAs intersperse the elements with additional empty cells (*gaps*). Carefully engineered update algorithms *rebalance* elements in the array to ensure that there is always an appropriate number of gaps in each region of the PMA. In this way, PMAs realize an interesting tradeoff between insertions and scan queries: They achieve amortized update times of $O(\log^2 N)$ with only a constant factor of additional gaps. Hence, range queries are still efficient as only few empty cells are scanned unnecessarily. At the same time, the few empty memory cells suffice to achieve updates that are significantly faster than in the naive case. However, updates on PMAs are slower than for search trees both in practice [16, 12] and asymptotically in theory.

In recent years, the increase in performance of a single processor core has slowed down significantly. Instead of faster single-core processors, multi-core processors have become the standard solution for increasing the performance of a computer [2]. One drawback of multi-core processors is that they do not automatically provide better application performance. Instead, the application must be explicitly parallelized so that it can be executed on multiple processor cores in parallel to improve performance. Depending on the application, this process can be difficult or even require a complete redesign of the underlying algorithms or data structures. Overheads for the coordination of work between the cores can quickly reduce the speedup that is achieved by parallel execution.

Therefore, fast and scalable *parallel data structures* are fundamental for the performance of parallel applications — and hence for efficiently using modern multi-core processors and continuing to take advantage of hardware improvements. In the context of parallel sorted sequences, there are two paradigms for parallelization. On the one hand, *concurrent* sorted sequences parallelize the access to the data structure: They allow multiple threads to concurrently perform queries on shared data, and threads see the effects of the updates of other threads. Concurrent data structures can be a simple option to parallelize an application: Instead of a single thread operating on a sequential data structure, multiple threads operate on a shared, concurrent data structure. However, concurrent sorted sequences have significant practical drawbacks. In the worst case, all concurrent updates target the same section of the data structure. Here, concurrent data structures do not scale well, as expensive locking or lock-free synchronization of threads is needed [2, 24]. Good concurrent performance often requires relaxing the semantics [19]. Achieving good speedups over tuned sequential data structures appears to be challenging unless solely read-only operations are executed concurrently [24].

On the other hand, *batch-parallel* sorted sequences process a *batch* of elements in each update and parallelize the execution of an individual batch update. By processing multiple elements in one operation, the work necessary for the update can be distributed efficiently across multiple threads. Moreover, parts of the work (like searching for the insertion position) can be amortized over multiple elements (a section of the batch). In contrast to concurrent data structures, any write accesses to the data structure depend only on the unique current batch update, and not on potential concurrent updates. Hence, sequential correctness is usually easy to achieve and complex parallelization techniques can be used. A recent result from Wheatman et. al. [26] suggests that batch-parallel PMAs can maintain the cache-efficiency of sequential PMAs and therefore provide faster scan queries than batch-parallel trees. However, batch updates on the PMA quickly require lots of work for rebalancing. In practice, the parallel execution becomes memory-bound, significantly restricting scalability [26].

We suggest the *Buffered Packed Memory Array* (BPMA), a novel approach of using a PMA to store references to buffer blocks — with capacity for multiple elements — instead of individual elements. Rebalancing in traditional PMAs is particularly memory-intensive because individual elements are moved. In our approach, the blocks buffer insertions or deletions until they over- or underflow. Only then, buffer blocks need to be split or merged, triggering insertions or deletions on the PMA. Hence, rebalancing work is significantly reduced. We extend the BPMA to the *Batch-Parallel Buffered Packed Memory Array* (BBPMA), a variant with batch-parallel update operations. With one level of indirection, our new data structures are a middle ground between single-element PMAs (without any indirections) and traditional, pointer-based search trees with many levels of indirections.

This thesis is structured as follows: We define sorted sequences and PMAs in Chapter 2 before reviewing related work in Chapter 3. Our new data structures are described and theoretically analyzed in Chapter 4. We give an overview of our implementations in Chapter 5 before presenting extensive experimental evaluations in Chapter 6. Finally, we summarize our findings and discuss future work in Chapter 7.

2. Preliminaries

In this chapter, we define sequential as well as batch-parallel sorted sequences and introduce an abstract definition of a Packed Memory Array (PMA). Explanations and analyses throughout this thesis will build on this abstract PMA.

2.1. Sorted Sequences

Largely following the definition by Sanders et al. [24], we define a basic sequential sorted sequence that provides update operations for single elements. A sorted sequence $\mathcal{S}$ stores a set of elements from a universe $Element$. Each element $e \in Element$ is associated with a key $key(e) \in Key$ from a universe of keys Key. The binary relation $\leq$ on Key induces a total order on the elements. The sorted sequence stores its elements in this order. We use $\perp \notin Key$ as *null key* to indicate that an entry is invalid. The number of elements in the data structure is N and we write $e \leq e'$ as a shortcut for $key(e) \leq key(e')$. The basic operations of a sequential sorted sequence, *insertion*, *deletion*, *search*, and *scan* are defined as follows:

- $\mathcal{S}.insert(e \in Element) : \mathcal{S} \coloneqq \mathcal{S} \cup \left\{e_a \in \{e\} \mid \nexists e_b \in \mathcal{S} : key(e_b) = key(e_a)\right\}$
- $\mathcal{S}.remove(k \in Key) : \mathcal{S} \coloneqq \mathcal{S} \setminus \{e \in \mathcal{S} \mid key(e) = k\}$
- $\mathcal{S}.locate(k \in Key)$: **return** $\min\{e \in \mathcal{S} \mid key(e) \geq k\}$
- $\mathcal{S}.scan(k_{\min} \in Key,\ k_{\max} \in Key,\ f \in Operation)$:
  for $\{e \in \mathcal{S} \mid k_{\min} \leq key(e) \leq k_{\max}\}$ **do** apply f to e

The operation $\mathcal{S}.insert(e \in Element)$ adds a new element e to the set, if no element with the key $key(e)$ was contained in the set before. This ensures that keys are unique for all elements in $\mathcal{S}$. Conversely, the operation $\mathcal{S}.remove(k \in Key)$ removes the unique element with the given key k from the set, if it exists. The search operation $\mathcal{S}.locate(k \in Key)$ returns the element with the smallest key larger than or equal to the given key k. Finally, the scan operation $\mathcal{S}.scan(k_{\min} \in Key,\ k_{\max} \in Key,\ f \in Operation)$ applies an operation f to all elements $e \in \mathcal{S}$ with $k_{\min} \leq key(e) \leq k_{\max}$. We prohibit duplicate keys in $\mathcal{S}$ and require that an existing element remains unchanged if a subsequent insertion operation tries to insert an element with the same key.

2.2. Batch-Parallel Sorted Sequences

Instead of the $insert(e)$ and $remove(k)$ operations which insert or remove a single element, batch-parallel sorted sequences offer batch manipulation operations that insert a batch of

elements $\mathcal{B}$ into the data structure or remove elements corresponding to a batch of keys $\mathcal{K}$ from the data structure. The number of elements or keys in the batch is k. The definitions of the single-element operations canonically extend to the batch operations:

- $\mathcal{S}.insertBatch(\mathcal{B} \subseteq Element) : \mathcal{S} \coloneqq \mathcal{S} \cup \left\{e_a \in \mathcal{B} \mid \nexists e_b \in \mathcal{S} : key(e_b) = key(e_a)\right\}$
- $\mathcal{S}.removeBatch(\mathcal{K} \subseteq Key) : \mathcal{S} \coloneqq \mathcal{S} \setminus \{e \in \mathcal{S} \mid key(e) \in \mathcal{K}\}$

Similar to the single element insertions, batch elements with keys that already exist in the data structure are ignored by $insertBatch(\mathcal{B})$. Practical implementations typically sort the batches for efficient parallel executions. For simplicity, we assume that batches are presorted and do not contain duplicate keys. In sorted batches, duplicates are easy to remove in a linear-time preprocessing step. A batch-parallel operation is executed in parallel by p *processing elements* (PEs). Lookup and scan operations remain unchanged. They can be called concurrently, but the read-only queries have to wait for potential batch-updates to finish.

2.3. Packed Memory Arrays

In this section, we establish an abstract definition of Packed Memory Arrays (PMAs) before we discuss specific realizations in Chapter 3. We start with the data structure layout, then present an abstract explanation of update and search operations and discuss rebalancing operations in detail. Our description closely follows that of Bender et al. [4, 9].

Data Structure Layout The PMA is a cache-friendly data structure that stores elements of a sorted sequence in contiguous memory. Figure 2.1 displays a schematic overview. A PMA for N elements is an array of size $P = \Theta(N)$, where P is a power of two larger than or equal to N. The array stores elements interweaved with *gaps* (empty memory cells). A PMA maintains the *density invariant* that there are always $\Theta(\ell)$ elements stored in any section of length ℓ greater than some small constant [12]. It ensures that there are enough gaps for fast updates, but also enough elements for memory efficiency and fast linear scans. To enforce the invariant, elements are *rebalanced* (i.e., moved within the array) if necessary. The array is implicitly subdivided into *segments*, i.e., sections of the array, of size $\log P$. Some operations on PMAs are defined using tree-terminology on an implicit perfect binary tree over the $P/\log P$ segments. Each leaf on level zero corresponds to one segment. The *region* $r(v)$ of a node v in the tree is the section of the PMA that is covered by segments in the subtree of v. The binary tree is not maintained explicitly.

Operations on the PMA Searches in the PMA are performed using binary search on the array. If a new middle index points to a gap in the PMA, the next valid entry can be found using linear search. A scan query simply locates the start of the range and iterates the PMA linearly until reaching the end of the range. The density invariant ensures memory efficiency. An insertion into a PMA consists of two phases: First, the position for the new element is located. If the position is not empty, other elements are moved out of the way until an empty cell is found. Then, the new element is written to the correct location. Thereafter, a *rebalancing operation* restores the density invariant, starting at the memory

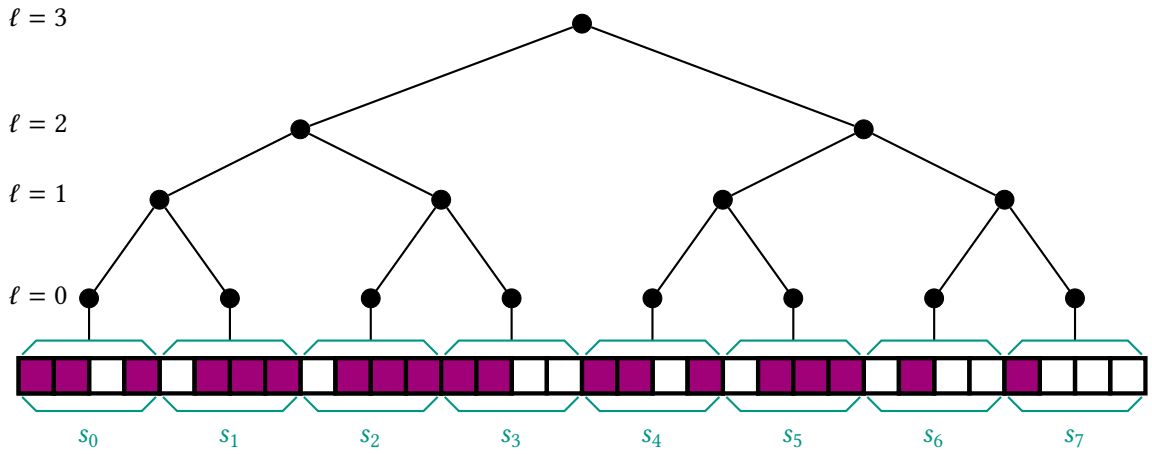


Figure 2.1.: Schematic overview of a Packed Memory Array with $N = 19$, $P = 32$ and segment size 4. A contiguous array (bottom) stores elements (purple) with gaps in between them (white). The array is implicitly classified into segments s_i (green). An implicit binary tree (top) over the segments is used to simplify the definition and analysis of algorithms. Leaves are on level zero.

cell that was empty and now contains an element (the new element or the shifted element that was present before). Rebalancing operations are explained in detail below. Removals proceed analogously: After the element is located, it is replaced by a gap, and a rebalancing operation starting from the new gap restores the density invariant.

Density Thresholds and Rebalancing To maintain the density invariant stated above, PMAs assert that the *density* in PMA regions defined by the implicit binary tree remains within specific thresholds. Let u be a node in the implicit binary tree on level $\ell = \ell(u)$ with region $r = r(u)$. It has capacity $cap(u)$ equal to the length of its region. The *density* $\tau(u)$ is defined as the ratio of used cells in $r(u)$ to $cap(u)$. PMAs bound the density of a node from below and above using a *minimum density function* $0 \leq \tau_{\min}(\ell) < 1$ and a *maximum density function* $0 < \tau_{\max}(\ell) \leq 1$ that depend on the level ℓ of the node. Here, $\tau_{\min}(\ell)$ is monotonically increasing and $\tau_{\max}(\ell)$ is monotonically decreasing in the level of the region. We write $\tau(u)$ and $\tau(r(u))$ as a shortcut for $\tau(\ell(u))$. The density invariant follows as all PMA implementations require these thresholds for some level ℓ where $\tau_{\min}(\ell) > 0$, $\tau_{\max}(\ell) < 1$ and the capacity of nodes on level ℓ is a small constant.

A *rebalancing operation* ensures that all regions respect their density thresholds. Suppose that a region $r(u)$ no longer respects its density thresholds after an insertion or deletion. The rebalancing operation redistributes elements in an enclosing *rebalancing region* $r' \supseteq r(u)$ that respects its thresholds. Effectively, this moves elements from too dense regions into less dense surrounding regions (after insertions) or from sufficiently dense regions into regions that do not have enough elements (after deletions). If the PMA does not respect its density threshold globally, it is typically reallocated in an array that is larger or smaller by a constant factor, and elements are distributed uniformly in it. The rebalancing region is constructed by ascending the implicit binary tree from u to the root. At each node, the number of elements in the corresponding region is counted.

The rebalancing region $r(v)$ is the region of the lowest node v whose region respects its thresholds.[1] Once the rebalancing region is established, elements are redistributed in it. Most implementations use a simple uniform distribution. Lemma 1 shows that this leaves $r(v)$ and all subregions (including $r(u)$) with valid densities. Since we are only interested in the asymptotic behavior, we ignore rounding errors for simplicity.

Lemma 1. *Let r' be a region that respects its density thresholds:*

$$\tau_{min}(r') \leq \tau(r') \leq \tau_{max}(r')$$

By redistributing elements in r' uniformly, all subregions $r_{sub} \subsetneq r'$ respect their density thresholds:

$$\tau_{min}(r_{sub}) \leq \tau(r_{sub}) \leq \tau_{max}(r_{sub})$$

Proof. Let $r_{\text{sub}} \subsetneq r'$ be a subregion of r'. Since $\ell(r_{\text{sub}}) < \ell(r')$, r_{sub} has density thresholds that are less strict than or equal to those of r':

$$\tau_{\min}(r_{\text{sub}}) \leq \tau_{\min}(r') \text{ and } \tau_{\max}(r_{\text{sub}}) \geq \tau_{\max}(r') \tag{2.1}$$

If elements are distributed uniformly in r', r_{sub} has density $\tau(r_{\text{sub}}) = \tau(r')$, conforming to the stricter thresholds of r'. With Inequations 2.1, it follows that r_{sub} respects its density thresholds. □

[1] As the binary tree is not held explicitly in practice, "ascending" the tree corresponds to extending the region to the left or right with exponentially growing lengths, constructing the regions defined by the tree.

3. Related Work

Packed Memory Arrays are well-studied in the literature. We first consider results in the sequential, single-element case to outline the strengths and weaknesses of PMAs. In the parallel case, we briefly consider concurrent approaches, but focus on existing batch-parallel works. For the batch-parallel case, we also review an approach based on search trees that we use in our experimental evaluations.

3.1. Sequential Packed Memory Arrays

The general problem of storing a dynamic set of N elements in sorted order in $\Theta(N)$ space [12] has been studied extensively before, both specifically in the context of PMAs [21, 22, 9, 4, 12, 16], but also independent of those [30, 31, 32, 8, 14, 17]. As far as possible, our descriptions rely on the schematic PMA introduced in Section 2.3 and only explain the specific choices made for its components in the respective work. An overview of running times for selected sequential PMA data structures can be found in Table 3.1.

Early Works Itai et al. [21] introduce an approach similar to modern PMAs (called *Sparse Tables*) and prove that amortized $O(\log^2 N)$ element moves are necessary per insertion, without considering deletions. Their data structure mostly follows our schematic description in Section 2.3, but it does not consider deletions and has no minimum density threshold. The maximum density threshold decreases by a constant term for each level, i.e., logarithmically in the length of the region. Itai and Katriel [22] prove that it is possible to achieve the same asymptotic bounds without restricting rebalancing regions to the regions defined by the implicit binary tree but note that there are no practical benefits to their rebalancing procedure.

Willard [30, 31, 32, 12] first proves a *worst-case* bound of $O(\log^2 N)$ element moves per insertion. We focus on a simplified variant of his data structure by Bender et al. [7]. We start with the abstract PMA from Section 2.3. Regions that originate from sibling nodes in the implicit binary tree are referred to as *sibling regions*. Besides limiting the densities of individual regions, Bender et al. require that the number of elements in sibling regions from level i differs by at most $2 \cdot 2^i$. Whenever this does not hold for such regions, 2^i elements are moved from the dense to the less dense region by shifting elements linearly to the less dense end. This requires $O(2^{i+1} \log N)$ element moves, as elements in 2^{i+1} segments of length $\Theta(\log N)$ need to be moved. To deamortize the time needed for rebalancing, the element moves are performed during the *next* 2^i insertions or deletions that occur within the two regions, processing $O(\log N)$ operations per insertion or deletion. In this way, partial rebalancing operations that originate from different past update operations on two sibling regions can be required during the same subsequent update operation. Bender et al. show that these can be executed together in $O(\log N)$ time. For each insertion or

Name	Ref.	Year	Impl.	Del.	Insertion		Search
					Time	I/Os	I/Os
Sparse Array	[21]	1981	○	○	$O\left(\log^2 N\right)$	$O\left(\log^2 N\right)$	$O\left(\log N\right)$
Deam-ortized	[7, 30]	2002	○	•	$O\left(\log^2 N\right)^*$	$O\left(\log^2 N\right)^*$	$O\left(\log N\right)$
PMA	[9, 4]	2005	○	•	$O\left(\log^2 N\right)$	$O\left(1+\frac{\log^2 N}{B}\right)$	$O\left(\log N\right)$
APMA	[12]	2007	•	○	$O\left(\log^2 N\right)$ $O\left(\log N\right)^{\ddagger}$	$O\left(1+\frac{\log^2 N}{B}\right)$ $O\left(1+\frac{\log N}{B}\right)^{\ddagger}$	$O\left(\log N\right)$
RMA	[16]	2019	•	•	$O\left(\log^2 N\right)$ $O\left(\log N\right)^{\ddagger}$	$O\left(1+\frac{\log^2 N}{B}\right)$ $O\left(1+\frac{\log N}{B}\right)^{\ddagger}$	$O\left(\log N\right)$
SPMA	[27]	2023	•	•	$O\left(\log^2 N\right)$	$O\left(1+\frac{\log^2 N}{B}\right)$	$O\left(\log_B N\right)$

Table 3.1.: Summary of selected sequential PMA data structures in chronological order. *Year* is the publication year of the latest cited reference. *Impl.* shows if an implementation is provided in the original source, and *Del.* shows if deletions were considered in the analysis. Search times are $O(\log N)$ for all structures. Running times marked with an asterisk* are worst case running times, all others are amortized. Running times marked with a dagger‡ only apply to certain input sequences.

deletion, rebalancing is needed for regions in at most $O(\log N)$ levels, yielding the worst case bound of $O(\log^2 N)$.

A generalization of the data maintenance problem for PMAs is *Online List Labeling* [8, 14, 17]. Here, N items of a dynamic set have to be stored in sorted order in an array of size P. The objective is to minimize the number of element moves required for an insertion or deletion. Clearly, PMAs rely on the special case $P = (1 + \Theta(1))\, N$. An upper bound of $O(\log^2 N)$ can be derived from the previously mentioned works [21, 30, 31, 32]. The same bound was shown to be a lower bound for deterministic algorithms [14] and any algorithms that only redistribute elements of a continuous subarray uniformly [17]. A recent theoretical result [8] proposes a randomized algorithm with expected cost $O(\log^{3/2} N)$, constituting a significant improvement over the previous state of the art.

Cache-Oblivious Packed Memory Array Bender et al. [9, 4] introduce the first cache-oblivious variant of the previous sparse array and introduce the term *Packed Memory Array*. Their data structure closely follows that of Itai et al. [21] but is also cache efficient as it uses only amortized $O(1 + (\log^2 N)/B)$ memory transfers in insertions and deletions, for the cache line size B. Opposed to Itai et al., they additionally support deletions while maintaining fast updates and scans by adding lower-bound density thresholds for regions. After each update, the rebalancing region is constructed by scanning to the left and right of the updated cell, reconstructing the regions of the leaf-to-root path in the implicit binary tree. Elements are then distributed uniformly in the rebalancing region. This way, only $O(1 + \ell/B)$ memory transfers are necessary for rebalancing a region of length ℓ.

Improvements to Packed Memory Arrays Several improvements to PMAs have been proposed. Bender and Hu [12] present the *Adaptive Packed Memory Array* (APMA) that achieves the original bounds from Bender et al. [9, 4] for all input distributions, but performs asymptotically better on some distributions. Specifically, APMAs improve the rebalancing cost to amortized $O(\log N)$ operations and amortized $O(1+\log N/B)$ memory transfers for sequences of random inserts, repeated inserts after the same element, as well as *bulk inserts*, where up to n elements are inserted at the same location at once. Opposed to traditional PMAs, APMAs do not necessarily distribute elements uniformly when rebalancing a region. Instead, they distribute them *unevenly* to achieve lower density in those segments of the PMA where recent insertions took place, anticipating further insertions there in the next operations. To control the uneven distributions, APMAs store a set of *markers*, i.e., predecessors of elements recently inserted into the PMA. For each marker, the *predictor* data structure maintains a pointer to the segment to which the element was inserted. Furthermore, the predictor provides an *insertion number*, a lower bound to the number of insertions after the marker within the last $O(\log^2 N)$ insertions. The predictor is designed to ignore noisy insertions, i.e., few insertions that do not follow the current update pattern. When redistributing the elements of a rebalancing region, elements are distributed unevenly to segments, and segments with a large insertion number receive less elements. The densities are carefully chosen to ensure that the usual amortized $O(\log^2 N)$ number of element moves is maintained in all cases and that the work needed for redistributions is asymptotically the same as for uniform redistributions. Deletions are not considered in the theoretical analyses and experiments. Experimental evaluations show that APMAs are up to 7 times faster than traditional PMAs when all elements are inserted at the same position, the worst case for traditional PMAs. A speedup of 3 is achieved if inserts at the start of the data structure and random inserts are mixed equally. For random inserts, uneven distributions do not seem to offer a practical advantage and APMAs are slower than traditional PMAs due to the additional overhead.

In *Rewired Memory Arrays* (RMAs), De Leo and Boncz [16] provide several improvements to (A)PMAs. They reconstruct the adaptive rebalancing scheme from Bender and Hu [12] to make it more robust. To improve scan performance, they store all elements of a segment at one end of the segment, while all empty cells are stored at the other end. As elements are no longer randomly interspersed with gaps, there are less branch mispredictions during scans. To maximize the length of continuous dense sections, they alternate between shifting elements to the left and right of a segment, so that two segments always form

one larger block of elements. To improve search performance, they add a static (fixed size) index with keys that can be used for faster navigation to the segments. Finally, they use *memory rewiring* [25] to rebalance elements in regions larger than a memory page with only one write per element instead of two. Traditionally, rebalancing first moves the elements of the rebalancing region to auxiliary storage without gaps, and then writes them back uniformly into the rebalancing region in the PMA. Instead, RMAs directly write elements uniformly to unused memory pages and replace the pages in the rebalancing region with the new pages by changing the virtual page addresses. The authors report a speedup of about 20 % by memory rewiring. Their adaptive rebalancing has an overhead of about 20 % for random insertions but achieves a speedup factor of around 5 for sequential inserts at the same position.

While PMAs are more cache-efficient than B-Trees [3] for scanning, they are less cache-efficient for searches. Each step of a binary search in a PMA reads an element from an arbitrary position, so that a binary search requires $\Theta(\log N)$ memory transfers. In contrast, B-Trees read $\Theta(B)$ splitters at once in an inner node, so that only $\Theta(\log_B N)$ memory transfers are required. Since search is also a component of update and scan queries, it influences their running times as well. Wheatman et al. develop *Search-optimized Packed Memory Arrays* (SPMAs) [27] to improve binary searches. Compared to traditional PMAs, SPMAs use a small auxiliary data structure that stores the first element, called *head*, of each segment. A binary search for the target segment in this compact data structure is more cache-friendly than in the full PMA. Cache-efficiency can be further improved by permuting the head elements so that the order in the auxiliary data structure resembles the order in which the elements would be accessed during a binary search. In this way, searches in PMAs can match the theoretical memory efficiency bound for B-Trees. In practice, searches in SPMAs are up to 3.6 times faster than in PMAs for an application benchmark for data stores. They achieve a speedup of around 2 compared to B+-Trees.

3.2. Parallel Packed Memory Arrays

To understand how PMAs can be parallelized, we first review works on concurrent PMAs. Then, we turn towards previous works on PMAs with batch-parallel operations, the focus of this thesis. Table 3.2 presents a comparison of the batch-parallel PMA and a batch-parallel search tree (see Section 3.3).

Concurrent Operations De Leo and Boncz [15] use a PMA with concurrent access to store dynamic graphs. Their data structure maintains one lock per segment. Before inserting or deleting an element and rebalancing a region, a thread has to acquire the lock of the respective segment. This is challenging if a rebalancing region extends further than the segment that was originally locked by the updating thread. Here, it is necessary to acquire additional locks, as the rebalancing region can include segments that are concurrently written by other operations. To avoid deadlocks, such rebalancing operations are executed by a centralized *rebalancing service*. It computes the rebalancing region, acquires all necessary locks, and partitions the rebalancing operation into subtasks. The subtasks are issued to a job queue from where they are fetched by worker threads. The rebalancing service releases the locks once all subtasks are executed and wakes up other threads that

wait for the lock of an updated segment. These threads have to verify that they still target the respective segment, as its content might have changed. Experimental evaluations show that this scheme is slow if concurrent updates target the same region of the PMA, as multiple writers compete for the same locks. To achieve better throughput in this scenario, a writer waiting for a lock passes its update operation to the thread that currently holds the lock. The update operation will then be executed asynchronously by the thread holding the lock. Again, a special case treatment is necessary if the queue of a thread contains elements that no longer belong to the segment that the thread updates. No theoretical analysis is presented for the data structure.

Wheatman and Xu [28] also present a PMA for dynamic graphs with concurrent updates but handle large rebalancing regions differently. Rather than centralizing the rebalancing in such cases, rebalancing is coordinated by the thread that performed the original update operation. This thread acquires all necessary locks for the rebalancing region using a carefully engineered scheme to acquire multiple locks in parallel while avoiding deadlocks. The element moves necessary for large rebalancing regions are then performed by multiple threads, using *inter-operation* parallelism: A concurrently called update might be executed by multiple threads. In this way, all operations have polylogarithmic span. To facilitate locking, all elements are stored to the left of a segment.

Batch-Parallel Operations Durand et al. [18] present a PMA that processes updates in mixed batches of insertions and deletions but executes these batch operations *sequentially*. Their solution is tailored to workloads from particle simulations. In each time step, the location of certain particles is updated by deleting the old entry for a particle and inserting a new one at the new location. Deletions and insertions are executed as one batch operation. First, all deletions are executed without rebalancing by replacing the respective elements with gaps. Second, insertions are performed by recursively splitting the PMA and the sorted list of insertion elements in the middle. After each split, the number of old and new elements in each half of the split is determined. If either of the two halves does not respect its density bound, both are rebalanced together, ending this branch of the recursion. If both halves respect their density bounds, they are each split again. A global resize is necessary if both initial halves do not respect their density bounds with the new number of elements. No theoretical analysis is provided.

Wheatman et al. [26] propose the *batch-parallel Compressed Packed Memory Array* (CPMA), the first PMA variant with batch-parallel updates. The batch insertion proceeds in three stages. First, the sorted batch is *merged* into the PMA. Second, a *set of* rebalancing regions is computed in the *counting phase*. Finally, the rebalancing regions are rebalanced.

To merge the batch into the PMA, the PMA segment for the middle element of the batch is located. The range of batch elements that need to be merged into the segment is found using exponential search on the batch. The remaining left and right halves of the PMA and the batch are processed recursively. For each segment reached by the insertion, a job is dispatched to insert the section of the batch into a segment. If all batch elements fit into the segment, they are inserted directly. Otherwise, all elements from the segment and those from the batch are stored in auxiliary storage and the segment maintains a pointer to them. If one segment receives a large section of the batch, this process is done using

Name	Ref.	Year	Del.	Insertion		Scan
				Work	Span	I/Os
Parallel Search Tree	[2]	2016	∘	$O\left(k \log \frac{N}{k} + \log N\right)$	$O\left(\log N\right)$	$O\left(\log_B N + \ell\right)$
CPMA	[26]	2023	•	$O\left(k \log^2 N\right)^{\dagger}$	$O\left(\log^2 N\right)$	$O\left(\log_B N + \frac{\ell}{B}\right)$

Table 3.2.: Comparison of two data structures for sorted sequences with batch-parallel updates. *Year* is the publication year of the cited reference. Both variants are implemented in the cited source. *Del.* shows if deletions were considered in the analysis. Running times marked with a dagger† are amortized, all others are worst case. Scan performance is for a scan of ℓ elements.

parallel merge. A reference to any segment that is altered during this phase is stored in a thread-safe set.

CPMAs use a binary tree over the segments to find rebalancing regions. Opposed to single-element operations, multiple segments of the PMA may have been changed in one batch update. To find all necessary rebalancing regions at once, the counting phase ascends the tree level-wise, processing levels sequentially and the necessary nodes of each level in parallel. The counting phase starts with the nodes on level zero that correspond to segments of the tree that were changed before. Here, it counts the number of elements in the corresponding segments. Whenever a number of elements is computed for any node, it is cached so that it can be reused later. If a leaf does not respect its density thresholds, its parent is stored to be counted in the next level. On each level above level zero, a node is counted by summing its children, reusing the cached number of elements for the children. If the number of elements is not cached for a child, the counting phase descends into the subtree rooted at the child. The traversal ends at any nodes that have a cached number of elements, or at leaves. Leaves are counted for a first time and the value is cached. When backtracking to the node that needs to be counted, the respective number of elements is cached for each node along the paths. In this way, it is guaranteed that each segment is only counted once in the counting phase and no segment is counted unnecessarily. If a node on a level above level zero respects its density bounds, its PMA region is stored as a rebalancing region. Otherwise, its parent is stored to be counted in the next level. Finally, the rebalancing regions from the previous step are rebalanced. First, the elements from the segment and its auxiliary storage are copied densely to a buffer. Then, they are redistributed uniformly into the target segments. Parallel copy is employed to parallelize the process.

CPMAs support insertions of sorted batches of size k in $O(k \log^2 N)$ amortized work and $O(\log^2 N)$ worst-case span. Experimental evaluation shows that scalability is limited in the variant described so far, as the algorithm is memory-bound in practice. It achieves a relative speedup of 19 on 128 cores. Therefore, the CPMA adds compression by *delta-encoding* elements in the segments. Each segment only stores its first element explicitly. From there, only differences (deltas) between subsequent elements are stored using the

minimal number of bits possible. Wheatman et al. use the CPMA as a key-store, storing only 40-bit integers without associated values. In this setting, delta encoding halves the space usage in comparison to a standard PMA, and the CPMA achieves a relative speedup of 43 on 128 cores. As the space reduction through compression depends on how small the deltas are, it is unclear how effective delta encoding can be for general use cases like storing keys associated with mapped values. While the key could be encoded efficiently, delta encoding offers no benefit for arbitrary (unordered) values. At the same time, values need to be stored inline in the PMA so that range queries accessing them are fast.

3.3. Parallel Search Trees

Search trees are a widely used representation for sorted sequences with various different implementations in the sequential case [24]. We briefly introduce sequential (a, b)-trees as they are the foundation of the batch-parallel search tree by Akhremtsev and Sanders [2] that we use for comparison to our data structures.

The (a, b)*-tree* [1, 20, 24] is a sequential search tree. Elements are stored in leaves, which are also connected in a doubly-linked list. In the search tree, each inner node except the root has an outdegree d between a and b. It stores $d - 1$ *splitters*, i.e., keys that are used for navigation between the children. All leaves have the same depth, so that the total tree depth is logarithmic in the number of elements. Search queries are answered by descending the tree from the root, selecting the next child using binary search in the splitters of each node. If $b \leq 2a$, a search requires time $O(\log b + \log N)$. Insertions create a leaf for the new element and add it as a child to the correct inner node. If the degree of an inner node is greater than b, it is split and the new inner node is added to its parent recursively.

Akhremtsev and Sanders [2] present a batch-parallel search tree based on the (a, b)-tree. We will refer to this data structure as the *Parallel Search Tree* (PST). A sorted batch of elements is inserted by parallelly splitting the tree and the batch into p subtrees and sections of the batch such that each section can be inserted into one subtree. Then, each processor can independently insert (or merge) its batch section into (or with) the subtree. Finally, the subtrees are joined in parallel. To achieve good load balance, the split positions are carefully chosen depending on both current tree and batch. Using intricate parallel split and join methods, a batch update for tree size N and batch size k requires parallel time

$$O\left(\frac{k}{p}\log\frac{N}{k} + \log p + \log N\right).$$

A comparison of this data structure and the CPMA is given in Table 3.2.

4. Buffered Packed Memory Arrays

In this section, we introduce three new Packed Memory Array data structures. First, we present the Buffered Packed Memory Array (BPMA), a sequential PMA with single-element operations and the block indirection. We continue with a description of the Batch-Parallel Buffered Packed Memory Array (BBPMA), a variant with batch-parallel operations. Finally, we add the Global Rebalancing Batch-Parallel Buffered Packed Memory Array (GBPMA), a simplified variant of BBPMA to investigate a simpler way of rebalancing.

4.1. Sequential Buffered Packed Memory Array

While traditional PMAs offer faster scans than pointer-based data structures, a major disadvantage of PMAs is that they require amortized $O(\log^2 N)$ time per single-element insertion or deletion, whereas search trees need time $O(\log N)$. The update cost is dominated by expensive rebalancing operations that move elements within the array. Existing PMAs can only match the update cost of pointer-based data structures for certain input patterns [12], but not in general. The *Buffered Packed Memory Array* (BPMA) is a new data structure based on PMAs that achieves update cost of only $O(\log N)$ using a single level of indirections, matching traditional pointer-based data structures.

4.1.1. Data Structure

Figure 4.1 shows the layout of the BPMA. Let N be the number of elements in the BPMA. It stores the elements in M *blocks* b_j for $j = 0, \ldots, M-1$. Blocks are maintained using the *reference PMA*, a PMA $\mathcal{R}$ with an array size of $P = \Theta(M)$. Each of the entries $\mathcal{R}[i]$ for $i = 0, \ldots, P-1$ is either a valid reference entry for a block b or empty. A *reference entry* for the block b is a three-tuple

$$ref(b) := (head(b), size(b), pointer(b)).$$

The *head* $head(b)$ is the key of the first element in b and $size(b)$ stores the current number of elements in b. The pointer $pointer(b)$ references the memory location for the elements in b. An empty entry is written as

$$(\bot, 0, \texttt{null}),$$

where $\bot$ is the null key. Valid block references in the reference PMA are sorted by the heads of their blocks. As usual, the reference PMA is allocated in contiguous memory. We write $\mathcal{R}[i].head$, $\mathcal{R}[i].size$ and $\mathcal{R}[i].pointer$ for the respective part of a reference entry at index i in the reference PMA.

Each *block* has capacity $C = \Theta(\log N)$ that is a multiple of 4. Elements in a block are stored densely (i.e., without gaps) and in sorted order. In this way, the order of block references in the reference PMA induces a global ordering of all elements in the BPMA. Elements in block b are indexed using 0-based indices and empty cells are marked as $\perp$:

$$b = \langle b[0], \ldots, b[size(b) - 1], \perp, \ldots, \perp \rangle.$$

The BPMA adopts the behavior of the PMA of Bender et al. [4, 9] for its reference PMA. Details are defined in Section 4.1.3. Furthermore, a BPMA maintains the *block size invariant*. It asserts that a block always contains at least $C/4$ elements, except if $N < C/4$ and $M = 1$. This limits the number of blocks and — simultaneously — the number of entries in the reference PMA to

$$M = \Theta\left(\frac{N}{\log N}\right).$$

Consequently, the maximum array size of the reference PMA is

$$P = \Theta(M) = \Theta\left(\frac{N}{\log N}\right).$$

Note that previous works often use the notion "block" for our PMA "segments". In our case, a block is allocated independently of the reference PMA and contains multiple elements. A nonempty entry of the reference PMA is a reference to a block. A segment is a sequence of adjacent (empty or nonempty) entries in the reference PMA.

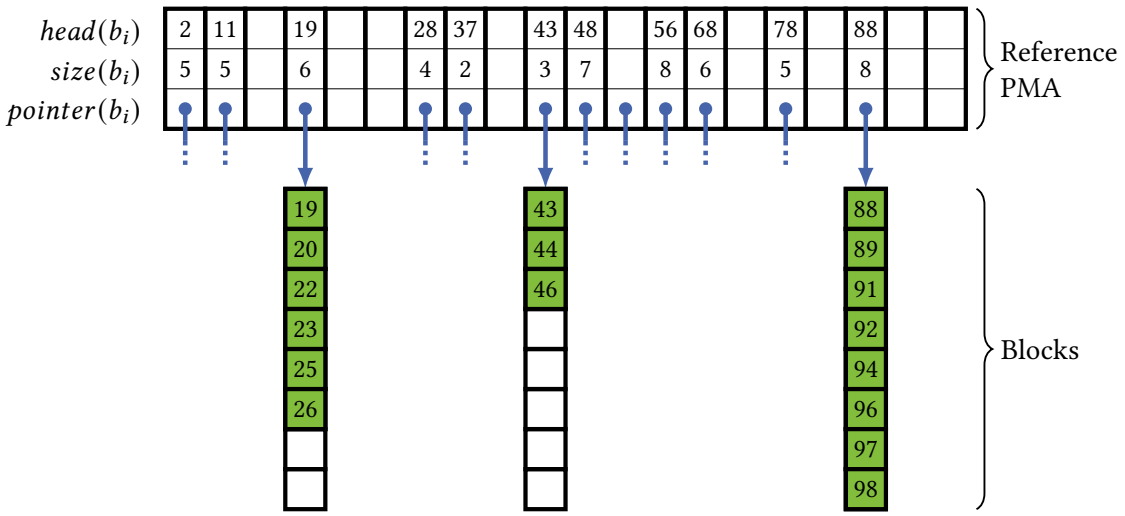


Figure 4.1.: A Buffered Packed Memory Array. The reference PMA (top) stores pointers (blue) to blocks along with the size and key of the first element (head) of the block. Block references are sorted with respect to the heads. The blocks (bottom) store elements (green) in sorted order. Each block has capacity $\Theta(\log N)$ and is at least one quarter full. For clarity, we only show three blocks and leave invalid block references empty. As an example, we store integers in the BPMA.

4.1.2. Search and Scan Queries

A search query $locate(k)$ in the BPMA first finds the last block b with head $head(b) \leq k$ in $\mathcal{R}$. This part of the search query only uses the reference PMA as the head of each block is stored within $\mathcal{R}$. If an element with key k exists, it must be in block b. Subsequently, the element is located within the block using binary search. A search query returns two pointers: One to the reference of the block in the reference PMA and one to the correct entry within the block.

To perform a scan query $scan(k_{\min}, k_{\max}, f)$, the start of the range is found using a search query for $k_{\min}$ on the BPMA. The search query initializes the two pointers used for iteration: One pointer to the current entry in the reference PMA and one pointer to the current element in the respective block. Since elements are stored densely in blocks, iterating a block is trivial. Whenever the end of a block is reached, the next valid block is found through a linear search on the reference PMA. The scan ends when an element e with $key(e) > k_{\max}$ is found or when the end of the BPMA is reached.

4.1.3. Update Operations and Rebalancing

Algorithm 1 gives a pseudocode representation of an insertion query $insert(e)$ on the BPMA. The BPMA first uses a search on the reference PMA to locate the last block b with $head(b) \leq key(e)$. For a new global minimum e, the first block is found instead. If b is full, the upper half of the elements is split off into a new block b', and $ref(b')$ is inserted into $\mathcal{R}$ behind $ref(b)$. After a block split, b is set to the block that e needs to be inserted to. After the split, or if no split is necessary, e is inserted into b. The insertion position in b is found using binary search and greater elements are shifted back by one position. The size and head of the block are updated in the PMA. If N is too large for the current block capacity C, the data structure is reconstructed with blocks that are larger or smaller by a constant factor.

A remove operation $remove(k)$ proceeds similar to insertions. Pseudocode is given in Algorithm 2. After the element e with $key(e) = k$ is located in block b, it is removed from b. If $size(b) \geq C/4$, it suffices to update $size(b)$ and $head(b)$ in the reference PMA. If instead $size(b) < C/4$, the deletion triggered an underflow, conflicting with the block size invariant. To maintain a valid block size, the BPMA initially tries to steal the first or last element from b', the successor or predecessor block of b and insert it to the end or start of b [23]. If either of these blocks has $size(b') > C/4$, an element can be stolen and the reference PMA entry for b' is updated. Otherwise, some neighbor block b' has $size(b') = C/4$. In this case, b can be merged into b' at the start or end, so that $size(b') = C/2 - 1$ and b' is still valid. After a merge, the reference PMA entry for b' is updated and the entry for b is removed from the reference PMA, potentially triggering a rebalancing operation on the reference PMA. The only case that leaves b with $size(b) < C/4$ is if no neighbors are found. Then, b is the only block so that $N < C/4$ and $M = 1$. Here, the block size invariant is trivially respected.

Rebalancing operations on the reference PMA are performed as outlined in Section 2.3. We redefine the density thresholds by Bender et al. [4, 9] in the notation introduced in Section 2.3. Note that we use levels (where leaves are on level zero) instead of depths

Algorithm 1: BPMA single-element insertion

Input: BPMA $\mathcal{R}$ with N elements, new element e with key $key(e)$.
Output: BPMA $\mathcal{R}'$ that contains all elements of $\mathcal{R}$ as well as e, in sorted order.

```
1  (b, i) ← R.getBlock(key(e))            // block b, index i to reference in R
2  if size(b) = C then
3      (b, b') ← b.split()                                  // lower and upper half
4      R[i].size ← size(b) j ← R.insertAt(b', i)   // insert ref(b') as successor
         of ref(b)
5      if key(e) ≥ head(b') then                                  // insert e to b'
6          (b, i) ← (b', j)
7  b.insert(e)
8  R[i].size ← R[i].size + 1
9  R[i].head ← b[0]
10 if N + 1 is too large for C then
11     Reallocate the BPMA with larger blocks
12 return R
```

(where the root node has depth zero). For height h of the implicit binary tree of the reference PMA, densities are controlled through arbitrary constants

$$0 < \tau_{\min}^0 < \tau_{\min}^h < \tau_{\max}^h < \tau_{\max}^0 = 1.$$

Then, the minimum and maximum density functions for nodes on level ℓ are defined as

$$\tau_{\min}(\ell) := \tau_{\min}^h + \ell \cdot \frac{\tau_{\min}^h - \tau_{\min}^0}{h}$$

and

$$\tau_{\max}(\ell) := \tau_{\max}^0 - \ell \cdot \frac{\tau_{\max}^0 - \tau_{\max}^h}{h}.$$

Note that $\tau_{\min}(\ell)$ is strictly monotonic increasing and $\tau_{\max}(\ell)$ is strictly monotonic decreasing in ℓ. On low levels, more variation in densities is tolerated — local densities can vary considerably. On higher levels, densities are controlled more strictly to keep the global density in a small range. During rebalancing, references in the reference PMA are distributed uniformly within a rebalancing region.

4.1.4. Analysis

To assess the influence of the block indirection in the BPMA in comparison to traditional PMAs, we analyze the space consumption of our data structure as well as the running times for update, search and scan queries. For the update queries, we perform an amortized analysis of the running times for single-element insertions and deletions.

Algorithm 2: BPMA single-element removal

Input: BPMA $\mathcal{R}$ with N elements, key k.
Output: BPMA $\mathcal{R}'$ that contains all elements of $\mathcal{R}$ except e with $key(e) = k$.

```
1  (b, i) ← R.getBlock(k)                    // block b, index i to reference in R
2  b.remove(k)
3  if size(b) ≥ C/4 then                                  // only update reference
4      R[i].head ← b[0]
5      R[i].size ← R[i].size − 1
6  else                                                       // fix block size
7      (b_ℓ, i_ℓ) ← R.predecessor(i)
8      (b_r, i_r) ← R.successor(i)
9      if b_ℓ ≠ ⊥ and size(b_ℓ) > C/4 then              // steal from predecessor
10         e_ℓ ← b_ℓ.stealEnd()
11         R[i_ℓ].size ← R[i_ℓ].size − 1
12         b.insertFront(e_ℓ)
13         R[i].head ← b[0]
14     else if b_r ≠ ⊥ and size(b_r) > C/4 then          // steal from successor
15         e_r ← b_r.stealFront()
16         R[i_r].head ← b_r[0]
17         R[i_r].size ← R[i_r].size − 1
18         b.insertEnd(e_r)                                   // no update needed
19     else if b_ℓ ≠ ⊥ then                              // merge with predecessor
20         b_ℓ.mergeGreater(b)
21         R[i_ℓ].size ← size(b_ℓ)
22         R.remove(i)
23     else if b_r ≠ ⊥ then                                // merge with successor
24         b_r.mergeSmaller(b)
25         R[i_r].head ← b_r[0]
26         R[i_r].size ← size(b_r)
27         R.remove(i)
28     else
           // N < C/4 and M = 1.  Invariant fulfilled.
29 if N − 1 is too small for C then
30     Reallocate the BPMA with smaller blocks
31 return R
```

Space Consumption We define the space consumption of a single element as 1. A traditional PMA with N elements requires space $\Theta(N)$ as the PMA invariants ensure that it only uses a constant factor of empty cells, and because the auxiliary memory that is required during rebalancing operations is in $\Theta(N)$. We show that the BPMA maintains the asymptotic space consumption of traditional PMAs, requiring only a constant factor more space than the optimum space consumption of N.

Lemma 2. *A BPMA containing N elements can be stored in space $\Theta(N)$.*

Proof. The BPMA stores elements in blocks instead of storing them directly in the PMA. Each block is allocated with a static capacity of C. The block size invariant guarantees that it contains at least $C/4$ elements. Therefore, the total space requirement for all blocks is $\Theta(N)$. Due to the block size invariant, there are at most $\Theta(N/\log N)$ blocks. The reference PMA has to store an equal number of block references, requiring space $\Theta(N/\log N)$ which is dominated by the space required for the elements. Analogously, the additional space required for rebalancing the reference PMA is a lower order term. □

Search and Scan Queries A search query on the BPMA requires a search on the reference PMA for the correct block and a search within the block. We show that search queries require the same time as in traditional PMAs.

Lemma 3. *A search query on a BPMA containing N elements requires time $O(\log N)$.*

Proof. Finding the correct block reference on a reference PMA that contains M block references requires time $O(\log M) = O(\log(N/\log N)) = O(\log N)$.[1] Binary search within the correct block requires running time $O(\log C) = O(\log \log N)$. □

Lemma 4. *A scan query that scans ℓ elements of a BPMA containing N elements requires time $O(\log N + \ell)$.*

Proof. A scan query first uses a search query to find the start of its range. From there, it iterates the elements of the PMA. Elements are densely stored in blocks so that $O(1)$ operations are needed per element within a block. When the end of a block is reached, worst-case $O(\log(N/\log N)) = O(\log N)$ operations are needed to find the next valid block reference using linear search on the segment. As each block contains at least $\Theta(\log N)$ elements, the search cost per scanned element is $O(1)$. □

Amortized Analysis of Update Operations Both insertions and deletions first navigate the reference PMA and work on blocks locally before potentially rebalancing the reference PMA. We consider the time for navigation and local work first before concentrating on an analysis of the amortized time needed for rebalancing the reference PMA.

[1] Note that this running time is only achieved by additionally storing the index or key of the first valid entry of each segment of the PMA. Then, the binary search can find the first key of each segment that is reached in $O(1)$. Otherwise, scanning the segment would require $O(\log M)$ time per step of the binary search, yielding a search time of $O(\log^2 M)$. Updating the additional information does not alter the other running times, as any update (insertion, deletion, or rebalancing) already traverses the entire segment in the worst case. Implementations that using segments of constant length do not require the additional information as segments can be scanned in time $O(1)$.

Lemma 5. *Without rebalancing, a single-element insertion or deletion into a BPMA with N elements can be executed in worst-case time $O(\log N)$.*

Proof. The block that an element must be inserted to can be found with a search query in time $O(\log N)$. Inserting or deleting an element in a block, splitting a full block, stealing from a neighboring block as well as merging with it are each possible in time $O(C) = O(\log N)$. A new reference entry for a split block can be added by moving $O(\log N)$ existing entries in the segment. Neighboring blocks for stealing or merging can be found in the same time by iterating the reference PMA. □

The remaining running time arises from rebalancing the reference PMA. We analyze the influence of BPMA update operations on rebalancing operations in the reference PMA. Our analysis broadly uses the argument of Bender et al. [4, 9].

Theorem 1. *Rebalancing on the reference PMA after single-element insertions and deletions in a BPMA with N elements is possible in amortized time $O(\log N)$ per update operation on the BPMA.*

Proof. We use the token method for amortized analysis [24]. To simplify the proof that sufficient tokens are available at each point, we implicitly subdivide the global account into one account for each combination of a block in the reference PMA and a level of the implicit binary tree. Each insertion deposits $\Theta(h) = \Theta(\log(N/\log N)) = \Theta(\log N)$ tokens in total, $c = \Theta(1)$ per account for each level of the block that its element was inserted to. Deletions deposit tokens analogously, but into the accounts of the block that ultimately lost an element. Typically, this is the block b the element was removed from. However, if an underflow in b was corrected by stealing from another block b', the tokens are deposited to b'. A rebalancing operation on a node u on level ℓ requires time $O(cap(u))$. Hence, it withdraws $cap(u)$ tokens from the respective accounts for level ℓ in all blocks that are currently in $r(u)$. When a block is split, the balances of its accounts are equally distributed to the accounts in the two new blocks. If two blocks are merged, their accounts are aggregated level-wise.

It needs to be shown that c can be chosen in such a way that there are sufficient tokens for each necessary rebalancing operation. Let an insertion into or deletion from the BPMA trigger a rebalancing operation for a node u at level ℓ of the implicit binary tree over $\mathcal{R}$. Then, $r(u)$ is within threshold, while the region of one of its children v (at level $\ell - 1$) is not. After the last rebalancing operation that affected $r(v)$, $r(v)$ had a density conforming to the (stronger) thresholds of u on level ℓ instead of its level $\ell - 1$ (compare the proof of Lemma 1):

$$\tau_{\min}(\ell) \leq \tau(v) \leq \tau_{\max}(\ell)$$

To show that there are enough tokens, we consider the number of block insertions or removals in the region $r(v)$ of the reference PMA (caused by block splits and block merges). We only look at operations that happened since the last rebalancing operation that affected $r(v)$. First, consider the case that $r(v)$ is too dense, which is triggered by an

insertion. If v has density $\tau(v) > \tau_{\max}(v) = \tau_{\max}(\ell - 1)$, there must have been at least

$$\begin{aligned}(\tau_{\max}(\ell - 1) - \tau_{\max}(\ell))\, cap(v) &= \frac{\tau^0_{\max} - \tau^h_{\max}}{h}\, cap(v) \\ &= \Theta\left(\frac{1}{\log P}\right) cap(v)\end{aligned} \tag{4.1}$$

block reference insertions. Each insertion into the reference PMA requires a block b to have been split into b and b'. At some previous point, b contained at most $C/2$ elements. For the block to have required a split, at least another $C/2$ elements must have been inserted into b. These insertions also added $c \cdot C/2$ tokens into the account of b for level ℓ. Since the last rebalancing operation affecting v, u was not rebalanced as rebalancing it would have affected $r(v)$. Therefore, the tokens were not withdrawn from the accounts for level ℓ. In total, there were $\Theta(1/\log P) \cdot cap(v)$ block splits (Equation (4.1)), each requiring $C/2$ element insertions. In total, this leaves

$$\Theta(C) \cdot \Theta\left(\frac{1}{\log P}\right) \cdot cap(v) = \Theta(cap(v)) \tag{4.2}$$

tokens available in the collective accounts for level ℓ of all blocks in $r(u)$.

The removal case is analogous: If v has density $\tau(v) < \tau_{\min}(v) = \tau_{\min}(\ell - 1)$, there must have been at least

$$\begin{aligned}(\tau_{\min}(\ell) - \tau_{\min}(\ell - 1))\, cap(v) &= \frac{\tau^h_{\min} - \tau^0_{\min}}{h}\, cap(v) \\ &= \Theta\left(\frac{1}{\log P}\right) cap(v)\end{aligned}$$

removals in the reference PMA, i.e., block merges of b_1 and b_2. Because they participate in a block merge operation, $size(b_1) \leq C/4$ and $size(b_2) \leq C/4$. Since new blocks are only created by splitting existing blocks, both blocks once used to have $C/2$ elements. This is even true for the initial block of the BPMA (which starts without any elements), as it must have been split at some point so that there is another block with which it can be merged. At least $C/4$ elements must have been removed from each block to reach a size smaller than or equal to $C/4$. For each of the removals, c tokens were deposited into the account for level ℓ of b_1 and b_2. In sum, there are $\Theta(C)$ tokens available per block merge. With Equation (4.2), it follows that there are $\Theta(cap(u))$ tokens in the collective accounts of blocks in $r(u)$.

Therefore, $c = \Theta(1)$ can be chosen so that the total number of tokens is sufficient for the withdrawals by the rebalancing operation. As the number of tokens withdrawn for a rebalancing operation is proportional to its cost, and the number of tokens spent per insertion or deletion is $O(\log N)$, the amortized cost of an insertion or deletion is $O(\log N)$.

Note that block references can move in the reference PMA due to rebalancing operations or reference insertions. However, the argument made above is independent of block movements as we track tokens per block. The accounts of a block always have a number

of tokens that is proportional to the previous number of insertions and deletions in the block. At any point, the tokens are available to the region that a block resides in. Above, we count the number of update operations by arguing how many block splits or merges must have occurred since the last rebalancing operation affecting the region. Therefore, no blocks can be moved into or out of $r(u)$ by a rebalancing operation. If a region for a leaf u has density 1 and a block b must be split, existing block references must be shifted to make space for the new block reference. In this case, a block reference for a block b' can be moved out of $r(u)$, so that its tokens are no longer available to u. Our proof already considers this: Instead of b', there is now an additional block that was split from b. To the same degree that there are less tokens in the new block than in b', there are also more free slots that can be used for future insertions. Block shifts therefore do not change the amortized complexity of update operations. □

The result of Theorem 1 above is consistent with the result that is achieved when considering the reference PMA as a black box. The proof indicates that amortized $O(\log N)$ single-element updates on the BPMA are necessary to trigger a block reference insertion or removal in the reference PMA. Rebalancing operations on the reference PMA take amortized $O(\log^2 N)$ time. This cost can be charged to the $O(\log N)$ single-element updates, so that $O(\log N)$ amortized operations are necessary for rebalancing per single-element update on the BPMA.

4.2. Batch-Parallel Buffered Packed Memory Array

The *Batch-Parallel Buffered Packed Memory Array* (BBPMA) is a variant of the BPMA that supports batch-parallel insertions. The efficient rebalancing operations of BPMAs are particularly promising in the batch-parallel case as the existing solution suffers from insufficient memory bandwidth to support fast parallel rebalancing in the PMA [26].

4.2.1. Data Structure

The data structure layout is based on the layout of the BPMA (see Section 4.1.1 and Figure 4.1). However, batch-parallel update operations create two new problems that need to be handled: First, a single batch-parallel update operation can affect multiple regions of the reference PMA so that they require rebalancing. To compute the rebalancing regions, densities must be computed in parallel in potentially overlapping regions. A naive parallelization of the sequential counting mechanism would count regions repeatedly, impairing efficiency. To avoid repeated computations of the same partial results, there needs to be a mechanism to store previously generated results. Second, good work balance must be ensured even in cases where few, large rebalancing regions have to be rebalanced. To this end, we need to be able to split large rebalancing regions into independent sections of a desired amount of work so that large rebalancing regions can be rebalanced in parallel.

Both of these requirements can be fulfilled by maintaining the (previously implicit) binary tree *explicitly* as a *rebalancing tree* $\mathcal{T} = (\mathcal{V}, \mathcal{E})$. Its nodes store the number of block references in their regions of the reference PMA. In this way, a result that was generated by one PE can be reused by other PEs, addressing the first problem. Moreover, the tree can be used to efficiently find subregions with a specific number of block references to split large rebalancing tasks, solving the second problem. We therefore define the BBPMA as a two-tuple

$$(\mathcal{R}, \mathcal{T})$$

consisting of the reference PMA $\mathcal{R}$ and the rebalancing tree $\mathcal{T} = (\mathcal{V}, \mathcal{E})$. Due to the PMA invariants and the block size invariant, $\mathcal{R}$ has size

$$P = \Theta(M) = \Theta\left(\frac{N}{\log N}\right).$$

As before, we use the rebalancing tree to define potential rebalancing regions. Leaves in the rebalancing tree correspond to segments in the reference PMA. Opposed to traditional PMAs and the BPMA, we use segments of *constant* size $S = \Theta(1)$ for the BBPMA. While this does not increase the asymptotic space requirements, it enables more efficient rebalancing, see Section 4.2.4. There are $L = O(P)$ leaves and the tree has height $h = O(\log P)$. We will define more precisely how the information in $\mathcal{T}$ is updated and when it is used in Section 4.2.2.

Search and scan queries are identical to the BPMA case. The read-only queries can be run concurrently but have to wait for potential batch-updates to finish. This can be implemented using a readers-writer-lock.

4.2.2. Batch-Parallel Insertion

A batch-parallel insertion inserts a batch $\mathcal{B}$ of elements into the data structure, ignoring elements with keys that are already present. Opposed to the single-element variant, we do not treat the reference PMA as a black box but define how it is rebalanced for each batch-parallel insertion. Broadly, the operation proceeds in three phases. First, the *insertion phase* splits both batch and the reference PMA into p sections so that each PE inserts one batch section into the corresponding section of the reference PMA. Second, the *update phase* updates the rebalancing tree and generates a set of global rebalancing regions. Third, the *rebalancing phase* distributes the work in the rebalancing regions to p tasks which are executed in parallel by the PEs. Pseudocode with an overview can be found in Algorithm 3.

Algorithm 3: BBPMA BATCH-PARALLEL INSERTION (OVERVIEW)

Input: BBPMA $(\mathcal{R}, \mathcal{T})$ with N elements, *sorted* batch of new elements $\mathcal{B}$ of size k, number of PEs p.
Data: Map $\mathcal{M}$ of rebalancing tree nodes to be updated, rebalancing Regions $\mathcal{V}'$, collection of auxiliary blocks $\mathcal{A}$.
Output: BBPMA $(\mathcal{R}', \mathcal{T}')$ that contains all elements of $(\mathcal{R}, \mathcal{T})$ as well as new elements from $\mathcal{B}$, in sorted order.

1 **do in parallel** `// 1: Insertion Phase`
2 Partition $\mathcal{R}$ and $\mathcal{B}$ into p corresponding parts $(\mathcal{R}_0, \mathcal{B}_0), \ldots, (\mathcal{R}_{p-1}, \mathcal{B}_{p-1})$
3 Insert $\mathcal{B}_i$ into $\mathcal{R}_i$
4 Store new blocks in $\mathcal{A}$
5 Store altered leaves to *rebalance* in $\mathcal{M}$

6 **do in parallel** `// 2: Update Phase`
7 Update $\mathcal{T}$ using $\mathcal{M}$, get rebalancing regions $\mathcal{V}' \subseteq \mathcal{V}$
8 Clear $\mathcal{M}$

9 **do in parallel** `// 3: Rebalancing Phase`
10 Distribute rebalancing work in $\mathcal{V}'$ to p rebalancing tasks $T_0, \ldots, T_{p-1}$
11 Execute the rebalancing task T_i, merging sections of $\mathcal{R}$ and $\mathcal{A}$
12 Store altered leaves to *recount* in $\mathcal{M}$
13 Clear $\mathcal{A}$ and $\mathcal{V}'$

Data Dependencies To improve the presentation of the algorithm, we first give an overview over the main data dependencies in our batch insertion algorithm, which are also reflected in Algorithm 3. The first data dependency is between the insertion phase and the rebalancing phase of the same batch insertion operation. Opposed to the single-element case, inserting a section of a batch into a section of the reference PMA can generate multiple new blocks at once before the next rebalancing operation is run. Therefore, it is possible that a region of the reference PMA does not have sufficient space for new block references. Consider a block b that is referenced from index i in $\mathcal{R}$ and let b be split into

two blocks b_1 and b_2. We refer to b_1 and b_2 as *auxiliary blocks* and their block references $ref(b_1)$ and $ref(b_2)$ are stored in auxiliary storage $\mathcal{A}$. To ensure that auxiliary blocks are found and represented at the correct position in the reference PMA, $\mathcal{R}$ maintains a pointer from cell i to the auxiliary blocks that originated from b along with the number of auxiliary blocks for b. Blocks are written to $\mathcal{A}$ in the insertion phase and read from $\mathcal{A}$ in the rebalancing phase, where they are merged into the reference PMA at the relative position of the pointers that reference them. Figure 4.2 gives an example. For simplicity, we do not show explicit pointers but draw auxiliary blocks below the cell from which they originate, in sorted order from top to bottom.

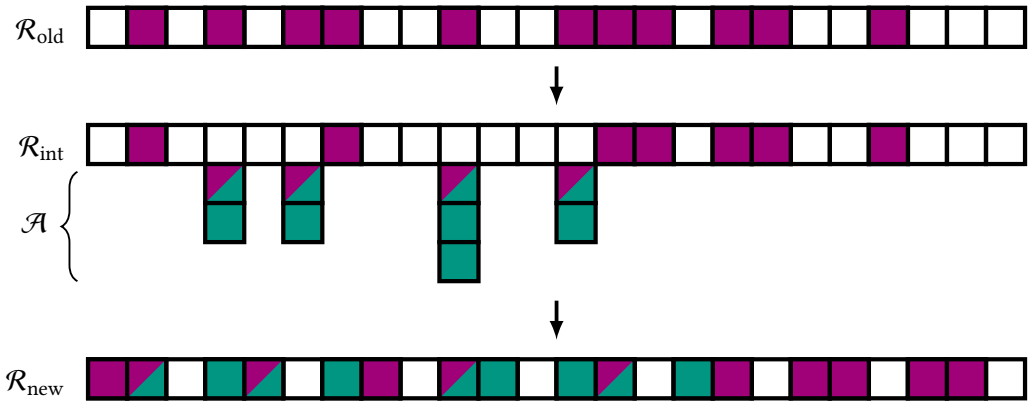


Figure 4.2.: Schematic representation of a batch-insertion operation on the reference PMA $\mathcal{R}_{\text{old}}$. Each square is one cell of a reference PMA. Existing block references are purple. The initial status is depicted as $\mathcal{R}_{\text{old}}$, while $\mathcal{R}_{\text{int}}$ shows the intermediate status after the insertion phase. We represent auxiliary blocks of $\mathcal{A}$ below their origin cell in $\mathcal{R}$ (green). Cells with both colors are blocks that existed in $\mathcal{R}_{\text{old}}$, received new elements and were moved to $\mathcal{A}$. At the bottom, $\mathcal{R}_{\text{new}}$ shows the status after rebalancing. The rebalancing phase merges blocks from $\mathcal{A}$ back into $\mathcal{R}$ in the desired order. It finds a region that respects its density thresholds with the new blocks (here: the entire reference PMA) and distributes block references uniformly in it.

The second data dependency concerns the update of the rebalancing tree $\mathcal{T}$. It is only used to generate rebalancing regions and distribute work in them. Hence, it is sufficient if $\mathcal{T}$ is correct after the insertion phase and before the rebalancing phase. The update phase not only ensures correctness of $\mathcal{T}$, but also integrates the generation of rebalancing regions into the update procedure. The update phase in a batch insertion operation must consider changes made by the current insertion phase as well as those made by the rebalancing phase of the *previous* batch insertion. Both phases track the leaves of $\mathcal{T}$ that correspond to changed segments in $\mathcal{R}$. However, changes from the two phases must not be treated equally: If a segment s was updated in a past rebalancing phase, its number of blocks might have changed. Hence, its leaf must be *recounted* in the update phase. The rebalancing phase already ensures that all densities are valid, and no further blocks must be merged. Only if s was changed during the current insertion phase, new blocks might have been written to $\mathcal{A}$ that need to be merged and the densities can be invalid. Here, the leaf must

be *rebalanced* to ensure that a rebalancing region is created for it. To coordinate which leaves need to be recounted or rebalanced, both insertion and rebalancing phase write leaves to a concurrent map $\mathcal{M}$. For each updated leaf, an entry in $\mathcal{M}$ tracks whether it needs rebalancing or just recounting. To track the combined changes of two subsequent batch insertions, $\mathcal{M}$ is persistently stored in the BBPMA. It is only cleared after it was read during an update phase, but not at the end of a batch insertion.

The last data dependency is between the update and rebalancing phases: The update phase creates a set of rebalancing regions that is described by their root nodes $\mathcal{V}' \subseteq \mathcal{V}$ in the rebalancing tree. It is read and cleared in the following rebalancing phase.

After the overview over the data dependencies, we describe the three phases of a batch insertion in detail. The description of the insertion and rebalancing phases are each split into a work distribution phase and an execution phase.

Insertion Phase: Distribution The insertion phase starts by distributing the work for insertions. To distribute the work, batch and reference PMA are partitioned into p regions each, such that each region of the batch can be inserted into the blocks in the corresponding region of the reference PMA. In that way, p PEs can process the actual insertions in parallel without synchronization as they write to disjoint parts of the data structure. To compute the partition, p batch elements with equal distances in the sorted batch are selected as *splitters*. PE i uses the i-th splitter s and locates the block b into which s needs to be inserted using a binary search. Then, each PE uses exponential search on the batch to find the first batch element in front of s that needs to be inserted into b. These searches can be run in parallel. In this way, each PE has defined the starting position of its regions in the batch and the reference PMA. The ends of the regions are derived from the starting positions of the successor PE, with appropriate sentinels for the last PE.

If consecutive splitters map to the same block, the ranges generated so far are not necessarily disjoint in the reference PMA. Therefore, a post-processing step ensures that there is a unique PE which performs all insertions for each block. This can lead to uneven work balancing if large sections of the batch map to a single block. To ensure good work balance on all input distributions, more flexibility is needed. If multiple PEs originally decided to work on the same block b, and there are sufficiently many batch elements in $\mathcal{B}'$ to be inserted, we use *inverse merge* execution of the insertions. In the standard execution described above, a small section of a batch is merged into the (comparably larger) block by a single PE. An inverse merge execution inverts this principle if a large section $\mathcal{B}'$ of the batch has to be inserted into a (comparably smaller) block b. Here, the existing *block elements* are merged into new blocks that are generated for the batch elements. The elements in $\mathcal{B}'$ are distributed equally between the PEs that were assigned to b. The PEs will create new blocks for their elements of $\mathcal{B}'$ and merge the appropriate elements of b into them so that b is no longer needed. To maintain the block size invariant, each PE that participates in an inverse merge execution must receive enough batch elements so that it can write at least one valid block. If there are not enough batch elements, the number of PEs that participate in the inverse merge execution is reduced. To accommodate new block references, each PE is allocated a section of $\mathcal{A}$ depending on the maximum number of block references that it can write.

Insertion Phase: Execution We begin the description of insertion execution with standard insertion execution and describe the inverse merge case below. Consider a PE i that got a section of the reference PMA $\mathcal{R}_i$ and a corresponding section of the batch $\mathcal{B}_i$. For standard insertion execution, the PE sequentially iterates over the blocks in $\mathcal{R}_i$ and adds all necessary elements from $\mathcal{B}_i$ to each block. It maintains pointers to the current insertion block b_{ins}, the next block b_{next} after b_{ins} in $\mathcal{R}_i$, and the next batch element e that needs to be inserted. At the start of the operation and whenever all batch elements were added to the previous insertion block, exponential search on $\mathcal{R}_i$ finds the new insertion block b_{ins}, i.e., the first block such that $key(e) \geq head(b_{\text{ins}})$. Then, the number of batch elements n that need to be inserted into b_{ins} is computed. If $size(b_{\text{ins}}) + n \leq C$, the elements are directly merged into the block.

Otherwise, b_{ins} must be split to accommodate all elements. Pseudocode for this case can be found in Algorithm 4. We will refer to the original b_{ins} as *origin block*. The reference for b_{ins} is removed from $\mathcal{R}$ as it will be written to $\mathcal{A}$. We iterate over the n elements that need to be inserted into b_{ins}. At each new element e, the algorithm first checks if e needs to be inserted into a block that was previously split off instead of into b_{ins}. To this end, the algorithm maintains a pointer b_{split} that stores a block that was split off, which is initially unused. If e has to be inserted into b_{split}, b_{ins} is no longer needed. It is written to $\mathcal{A}$ as next child of the origin block and b_{ins} is overwritten by b_{split}. If b_{ins} is full, the greater $C/2$ elements of b_{ins} are split off into a new block b_{new}. If b_{split} is not used yet, b_{new} is stored in b_{split} for the next iteration.

The alternative case is also possible: The insertion block has been split once, leaving greater elements in b_{split}. Then, it is filled up and split again, leaving three blocks b_{ins}, b_{new} and b_{split} with elements in this order. An important observation is that in this situation, no more elements have to be inserted into b_{ins}. Figure 4.3 illustrates our argument with an example. After the insertion block was split the first time, b_{ins} has exactly $C/2$ elements. For it to be split again, another $C/2$ insertions from $\mathcal{B}$ are necessary. Therefore, b_{ins} contains these elements when it is split again. Because the batch is sorted, the next batch element e is greater than all the $C/2$ elements that were inserted. After the second split, b_{ins} contains the half of the elements with the smaller keys. There are two possible cases. In the worst case, these are all $C/2$ elements from the batch that were previously inserted. As e is greater than all those, it must either be inserted at the end of b_{ins} or into b_{new}. In the second case, at least one of the $C/2$ elements that were inserted from the batch is in b_{new} after the split. Then, e must be inserted into b_{new}. In both cases, it is possible to insert e into b_{new}, so that b_{ins} is no longer needed. The current b_{ins} can be written to $\mathcal{A}$ as next child of the origin block and replaced by b_{new}. In consequence, the three block pointers are sufficient for all necessary references. After any block split, it is checked again if e has to be inserted into b_{ins} or b_{split}. Then, the element can be inserted into the respective block.

Once all elements are added, b_{ins} and potentially b_{split} are written to $\mathcal{A}$ as next children of the origin block in that order. In total, this removes the origin block from $\mathcal{R}$ and adds a sequence of auxiliary blocks in $\mathcal{A}$. See Figure 4.2 for an example.

In the case of inverse merge execution, each PE receives a section of the batch $\mathcal{B}_i$ and a single block b_{merge} that it shares with other PEs. It first computes the section of b_{merge} that it needs to merge with its batch section. Starting with an empty insertion block b_{ins}, it iterates through $\mathcal{B}_i$ and its section of b_{merge}. At each iteration, it selects the next element e

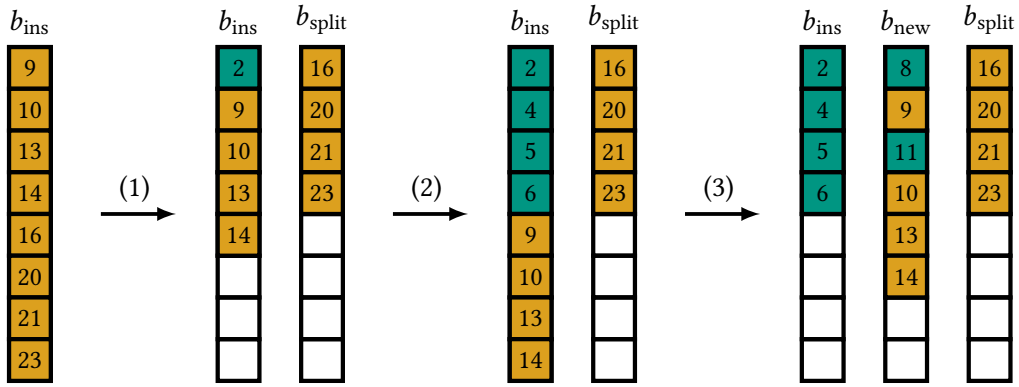


Figure 4.3.: Example for the insertion of a batch section $\langle 2, 4, 5, 6, 8, 11\rangle$ (green) into a block b_{ins} (elements in orange) with repeated splits. At operation (1), the element 2 is added into b_{ins}, triggering its first split. After operation (2), the first $C/2$ batch elements were added to b_{ins}. In operation (3), the insertion of the next batch element 8 triggers the second split. Now, any further batch elements (8, 11) can be inserted into b_{new} or b_{split} instead of b_{ins} and b_{ins} is no longer needed for insertions.

from $\mathcal{B}_i$ and b_{merge}. In the case of equal keys, elements from b_{merge} are favored. Then, e is appended to b_{ins}. If b_{ins} is full, it cannot always simply be replaced by a new empty block. This is only possible if it is guaranteed that the remaining elements suffice to maintain the block size invariant for the new block. Therefore, the minimum number of elements n_{min} that is left to be inserted is computed, considering potential duplicates. If $n_{\text{min}} \geq C/4$, b_{ins} can be written to $\mathcal{A}$ and be replaced by a new, empty block. Otherwise, b_{ins} is split, the left half is written to $\mathcal{A}$ and the remaining elements are added to the right half. If more than n_{min} elements are left to insert, the block might need to be split again. Each PE adds their new blocks to $\mathcal{A}$ as children of b_{merge}. There, they are ordered first by the PE adding them, then by the order in that they were added by their PE to ensure a correct global ordering. After all insertions are executed in all PEs, b_{merge} is removed from $\mathcal{R}$ as its elements are now represented in blocks in $\mathcal{A}$.

Whenever the insertion phase removes a block reference from $\mathcal{R}$ or adds new blocks to $\mathcal{A}$ as children of a cell in $\mathcal{R}$, the corresponding segment s must be updated in the following update phase. Let v be the leaf of $\mathcal{T}$ that corresponds to s. If there is no entry for v in $\mathcal{M}$, the insertion phase adds an entry that classifies it as a *rebalancing node*. If there is an entry for v that classifies it as a recounting node, the insertion phase adjusts the entry to classify v as a rebalancing node instead.

Update Phase The update phase updates the rebalancing tree so that the information stored in it is correct. This serves two purposes: The rebalancing regions are computed in the update phase using the rebalancing tree $\mathcal{T}$ with updated information. Furthermore, the following rebalancing phase uses $\mathcal{T}$ to parallelize the rebalancing process. The update phase passes the tree twice: The upwards pass updates the information in the tree, while the downwards pass generates the set of necessary rebalancing regions. Both passes process levels sequentially in the respective order, while they process the relevant nodes

Algorithm 4: BBPMA batch-parallel insertion: block split case

Input: Insertion block b_{ins}, number of batch elements n to be inserted, batch section $\mathcal{B}$ that has to be inserted with $|\mathcal{B}| = n$, map $\mathcal{M}$ of rebalancing tree nodes to be updated, auxiliary blocks $\mathcal{A}$.

```
1  remove b_ins from R                                              // moves to A
2  add leaf for b_ins to M for rebalancing
3  b_split ← ⊥; i ← 0
4  while i < n do
5      if b_split ≠ ⊥ and B[i] ≥ head(b_split) then                 // switch blocks
6          A.write(b_ins)
7          b_ins ← b_split; b_split ← ⊥
8      if size(b_ins) = C then                                      // block must be split
9          (b_ins, b_new) ← b_ins.split()                           // lower and upper half
10         if b_split = ⊥ then
11             b_split ← b_new
12         else
13             A.write(b_ins)          // at least C/2 insertions to b_ins ∪ b_new,
14             b_ins ← b_new           // B[i] greater than all those, can go to b_new
15     else
16         b_ins.insert(B[i])
17         i ← i + 1
18 A.write(b_ins)
19 if b_split ≠ ⊥ then A.write(b_split)
```

on each level in parallel. They maintain the nodes that need to be processed in the current level in a map $\mathcal{M}$ and add nodes that need to be processed on the next level to a map $\mathcal{M}'$. The roles of $\mathcal{M}$ and $\mathcal{M}'$ are swapped after each level.

The upwards pass maintains the invariant that all information stored in levels that were already processed is correct. On each level, all nodes in $\mathcal{M}$ are processed in parallel by the PEs, either as recounting or rebalancing node. We describe how an individual node is processed sequentially by the PE to which it was assigned. Pseudocode can be found in Algorithm 5. We start at level zero, considering a leaf u. First, the total number of block references in its segment including auxiliary blocks is counted. The remaining work depends on the type of u. For a recounting leaf u, the value of the leaf in $\mathcal{T}$ is updated to the new number of elements. If the value changes, the parent $parent(u)$ is added to $\mathcal{M}'$ as a recounting node so that the updated count is propagated up the tree. Processing a rebalancing leaf u is more complicated: If the corresponding segment respects its density threshold, u is *marked* as potential rebalancing node in $\mathcal{T}$. Then, the entry for u in $\mathcal{T}$ is updated. If the value changes, $parent(u)$ is added to $\mathcal{M}'$ as a *recounting node*. As the rebalancing region was already created, it suffices to propagate the updated number of elements up the tree. If the density threshold for $r(u)$ is not respected, u is not marked,

the entry in $\mathcal{T}$ is updated and $parent(u)$ is added to $\mathcal{M}'$ as a rebalancing node, even if the number of blocks did not change. This ensures that a rebalancing region will be generated on a higher level. Similar to the insertion phase, any node v that is added to $\mathcal{M}'$ as a rebalancing node overwrites a potential previous entry that classifies v as a recounting node. A new update classifying v as recounting node is ignored if an existing entry for v classifies it as a rebalancing node. For inner nodes, the number of blocks is counted by simply adding the values for the two children. These are correct due to the invariant stated above. The upwards pass finishes at the root node or if no more nodes need to be updated.

Algorithm 5: BBPMA batch-parallel insertion: node update (upwards pass)

Input: BBPMA ($\mathcal{R}$, $\mathcal{T}$), single node u to be updated as recounting or rebalancing node. Map $\mathcal{M}'$ for nodes to be updated on the next level.

```
1  if ℓ(u) = 0 then                                   // compute number of blocks
2      n ← R.countBlocks(r(u)) + R.countAuxiliaryBlocks(r(u))
3  else
4      n ← T[leftChild(u)] + T[rightChild(u)]
5  if u is recounting node then
6      if T[u] ≠ n then
7          Add parent(u) to M' as recounting node
8  else                                                // u is rebalancing node
9      if n/cap(u) ≤ τ_max(u) then
10         Mark u as potential rebalancing node
11         if T[u] ≠ n then
12             Add parent(u) to M' as recounting node
13     else                                            // still needs rebalancing
14         Add parent(u) to M' as rebalancing node
15 T[u] ← n                                            // update node value
```

The objective of the downwards pass is to find all nodes that were previously marked as potential rebalancing nodes and that are not in the subtree of any other node that is marked. Their regions form the set of required rebalancing regions without any duplicate work. The downwards pass only considers nodes that were processed during the upwards pass. To this end, each node remembers from which child nodes it was reached in the upwards pass (none, left, right, or both). In the downwards pass, each node in $\mathcal{M}$ is classified as a *generating node* or *clearing node*. A generating node might still generate a rebalancing region in its subtree, while a clearing node is in the subtree of a node that created a rebalancing region. It is only used to clear the auxiliary information stored in $\mathcal{T}$ (marks for potential rebalancing nodes and children from which a node was reached). The downwards pass begins at the highest level where nodes were processed in the upwards pass. Here, it considers all nodes that were processed on that level (using the previous $\mathcal{M}$) as generating nodes. Again, we describe the sequential processing of a node u. For a

clearing node u, the respective children u was reached from are added to $\mathcal{M}'$ as clearing nodes, and the auxiliary information of u is cleared. For a generating node u which is not marked as potential rebalancing region, the respective children from which u was reached are added to $\mathcal{M}'$ as generating nodes and its auxiliary information is cleared. If u is a generating node marked as potential rebalancing region, u is added to $\mathcal{V}'$. This ensures that $r(u)$ becomes a rebalancing region. In this case, the respective children of u are added to $\mathcal{M}'$ as *clearing* nodes. No more rebalancing regions are needed from the subtree of u, but the auxiliary information in it needs to be cleared. This ensures that all required rebalancing regions are found, while the downwards pass processes only the nodes that were already used in the upwards pass. If the root node does not respect its density threshold, a rebalancing region for the entire reference PMA is created that triggers $\mathcal{R}$ to be reallocated in a larger array. In this case, the downwards pass is not needed.

Rebalancing Phase: Distribution The rebalancing regions generated in the update phase define the necessary work to rebalance $\mathcal{R}$. To rebalance a region $r(u)$, the respective blocks from $\mathcal{R}$ and $\mathcal{A}$ are merged in the correct order and uniformly distributed in $r(u)$. An example with multiple rebalancing regions can be found in Figure 4.4. If no rebalancing regions were generated, all auxiliary data structures are reset and the batch insertion operation is complete.

It is crucial to distribute the work in the rebalancing regions for good work balance. Notably, it is not sufficient to just allocate entire rebalancing regions to the PEs and rebalance each region with one PE. In the worst case, there is only a single large rebalancing region so that this approach would not be scalable. We first define the distribution we want to achieve and then explain how it can be computed. Consider a set of rebalancing regions defined by the set of their root nodes $\mathcal{V}'$. Each rebalancing region $r(u)$ respects its density thresholds so that the sequence of blocks from $\mathcal{R}$ and $\mathcal{A}$ can be uniformly distributed in $r(u)$. We define the *target numbering* of all blocks of all rebalancing regions of $\mathcal{V}'$ as follows: We process rebalancing regions from left to right in $\mathcal{R}$. Within each rebalancing region, we use an in-order numbering of blocks from $\mathcal{R}$ and $\mathcal{A}$, compare Figure 4.4. The objective of the distribution is that each PE processes an equally sized, continuous section of the blocks with respect to the target numbering. Figure 4.4 shows the distribution using colored blocks. In terms of the target numbering, the desired distribution is trivially defined: Each PE reads an equally wide range of target numbers and writes them to uniformly distributed positions defined by the target numbers. However, to execute the rebalancing work, we need to access the respective blocks. In particular, it is not sufficient to only distribute indices in $\mathcal{R}$ as all the work might be in auxiliary blocks that originated from a single cell in $\mathcal{R}$. Therefore, we introduce an *extended block index* that uniquely identifies each block reference from $\mathcal{R}$ *and* $\mathcal{A}$. An extended block index is a tuple $(i,\ j)$ where i is an index into $\mathcal{R}$ and j is an index into the sequence of auxiliary blocks of $\mathcal{R}[i]$. We use $j = -1$ to denote the block reference at $\mathcal{R}[i]$. Values $j \geq 0$ denote the respective entry in the sequence of auxiliary blocks of $\mathcal{R}[i]$ with zero-based indices. Examples are given in Figure 4.4.

The remaining challenge is to map a given target number t to an extended block index $(i,\ j)$. We start by computing an inclusive prefix sum over the number of block references that need to be rebalanced in each rebalancing region. This can be done using a parallel

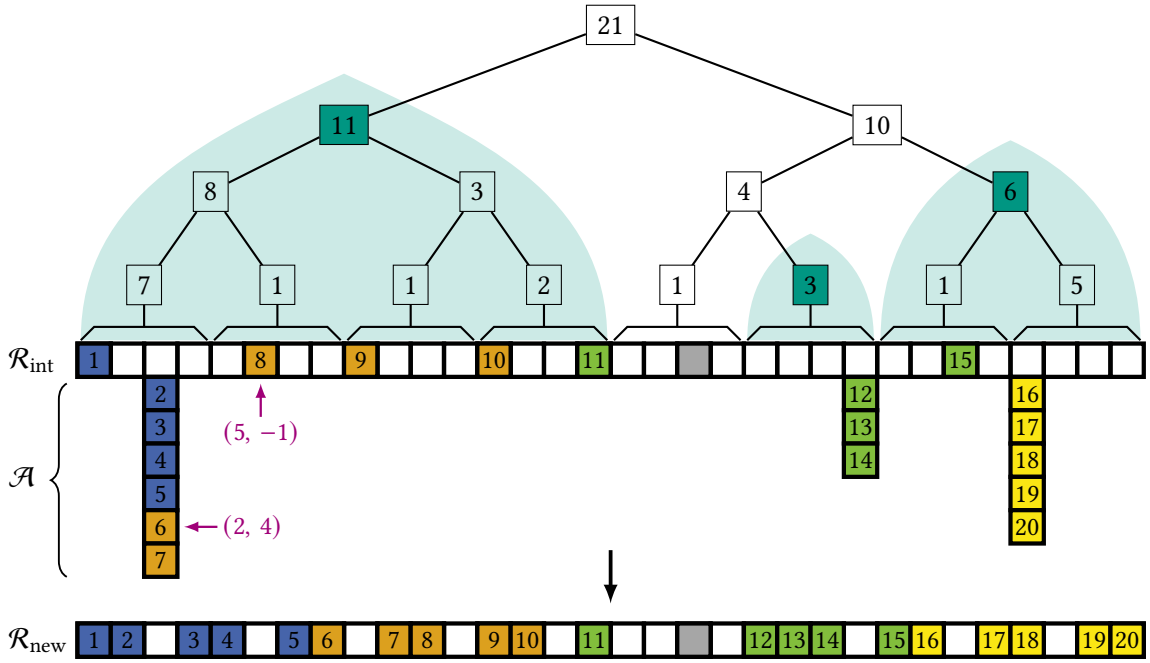


Figure 4.4.: Schematic representation of the rebalancing phase. The top shows the status of the BBPMA after the update phase. New blocks in $\mathcal{A}$ are drawn below their parent cells. There are three rebalancing regions (dark green) with a total of 20 block references that need to be rebalanced. Block references are numbered in the target ordering. We give two examples for extended block indices (purple). The rebalancing regions are split between four PEs (blue, orange, light green, yellow) for good work balance. One block reference does not need to be moved (gray). The bottom shows the reference PMA after the rebalancing regions were executed. Blocks from $\mathcal{A}_{\text{new}}$ are merged into $\mathcal{R}$ and block references are distributed uniformly in their respective rebalancing regions.

prefix sum. Then, a binary search on the prefix sum for the first rebalancing region $r(u)$ with prefix sum greater than the desired target number t yields the rebalancing region that contains the block index i. Using the prefix sum, t is transformed to a *local* target number within $r(u)$. Then, the correct extended index is found by descending the subtree rooted at u to the leaves. A pseudocode representation of this procedure can be found in Algorithm 6. At each inner node v, the algorithm decides between the left and right child. If the number of block references in the left child is greater than or equal to the current value of t, it descends to the left child. Otherwise, the right child is chosen and t is reduced by the number of block references that were skipped in the left child. Once a leaf v is reached, its segment s is scanned from left to right. When scanning an entry at index k, t is reduced by 1 for a valid block reference in $\mathcal{R}[k]$ and by the number of block references in $\mathcal{A}$ that correspond to $\mathcal{R}[k]$. When t reaches a value below or equal to zero at an index k, the block index is $i = k$ and the respective index for the auxiliary block j can be computed from t.

Algorithm 6: BBPMA batch-parallel insertion: rebalancing range split

Input: Rebalancing region denoted by node u, *local* target number t of the block reference that shall be found in $r(u)$.
Output: Extended index (i, j) of the block reference with local target number t within $r(u)$.

```
1  while ℓ(u) > 0 do                                          // descend subtree
2      if t > T[leftChild(u)] then                  // go right, skip left blocks
3          u ← rightChild(u)
4          t ← t − T[leftChild(u)]
5      else                                                        // go left
6          u ← leftChild(u)
7  foreach index i in r(u) do                    // u is a leaf, search locally
8      if t = 0 then                                      // found at normal block
9          return (i, −1)
10     if R[i] is valid block reference then
11         t ← t − 1
12     if t < R.numberOfAuxiliaryBlocks(i) then   // found at auxiliary block
13         return (i, t)
14     t ← t − R.numberOfAuxiliaryBlocks(i)
```

We can now assemble the whole rebalancing distribution process. Each PE receives a target number range of blocks $[a, b]$ that it has to rebalance. Using binary searches on the prefix sum, it computes a corresponding range of rebalancing regions $r_0, \ldots, r_i$. The first and last rebalancing regions can be shared between multiple PEs, each rebalancing a section of it. Therefore, the PE maps its block range $[a, b]$ to the local numbering within the respective rebalancing region using the prefix sum. Then, it descends the tree as described above to receive a range of extended indices that it works on within the region. For the remaining rebalancing regions, no computations are necessary as the PE rebalances the entire region in $\mathcal{R}$. Example 1 defines the mapping corresponding to Figure 4.4.

Example 1. *We define the distribution described in Figure 4.4 in terms of target numbering and extended indices. As there are 20 block references to rebalance, each of the four PEs rebalances five blocks. This yields target number ranges* $[1, 5]$, $[6, 10]$, $[11, 15]$, *and* $[16, 20]$ *for the respective PE. These regions are mapped as follows: The first and second PE rebalance only the first region. The subregions in terms of extended indices are* $(0, -1)$ *to* $(2, 3)$ *for the first PE and* $(2, 4)$ *to* $(12, -1)$ *for the second PE (inclusive ranges). The third PE works on all three rebalancing regions. For the first region, it only rebalances the block with extended index* $(15, -1)$. *It rebalances the second region entirely and only considers block* $(26, -1)$ *for the third region. Finally, the fourth PE only works on the third region and rebalances the extended block indices* $(28, 0)$ *to* $(28, 4)$.

Rebalancing Phase: Execution Executing the rebalancing work requires two steps: In the *read step*, blocks are read from $\mathcal{R}$ and $\mathcal{A}$ and merged densely to some auxiliary storage for all rebalancing regions in parallel. Then, all PEs are synchronized. If necessary, the reference PMA is reallocated. In the *write step*, the PEs write the blocks from their auxiliary storage back into $\mathcal{R}$, distributing them uniformly in their rebalancing regions.

We consider a PE with a set of rebalancing regions $r_0, \ldots, r_i$ from the distribution phase. For the first and last rebalancing task, it has a region of extended indices that it needs to consider while it processes the remaining rebalancing regions entirely. The PE iterates over its range of rebalancing regions. For a rebalancing region $r(u)$ that it needs to process entirely, both steps are straightforward: In the read step, the PE iterates over the respective entries in $\mathcal{R}$ and their children in $\mathcal{A}$ in the order of the target numbering and writes all block references densely to the auxiliary memory, removing them from $\mathcal{R}$. In the write step, it iterates over the dense section of block references in auxiliary memory and writes the block references back to the range $r(u)$ in $\mathcal{R}$ uniformly.

Processing a section of the first or last rebalancing region $r(v)$ requires some additional work. In the read step, the PE iterates over the *read section*, the section of $r(v)$ defined by the extended indices as computed in the distribution. Again, it writes block references densely to some auxiliary memory. After the synchronization, the PE computes the *write section*, the section of $r(v)$ in $\mathcal{R}$ to which it needs to write the block references uniformly. To this end, it maps the local target number range that it processes in $r(v)$ to uniformly distributed indices in $r(v)$. This ensures that each PE writes to independent subregions of $r(v)$. The global synchronization between read and write step ensures that no blocks are overwritten before they are copied to auxiliary memory. This is crucial as the write section is not necessarily equal to the read section of a PE (compare Figure 4.4). As all PEs work on disjoint sections of $r(v)$ in both phases, no further synchronization is needed.

During the write phase, the PE adds the leaves corresponding to all changed segments of $\mathcal{R}$ to $\mathcal{M}$ as recounting nodes. After all blocks are read by all PEs, pointers from $\mathcal{R}$ to $\mathcal{A}$ are reset and the rebalancing tasks are cleared.

4.2.3. Batch-Parallel Removal

Batch-parallel removals proceed largely analogous to batch-parallel insertions. Therefore, we focus on key differences to the insertion case. Again, the removal is split in three phases for *deletion* of the respective elements, *update* of $\mathcal{T}$ and *rebalancing* of $\mathcal{R}$.

Deletion Phase In the deletion phase, distribution starts similar to the insertion case by partitioning the batch of keys $\mathcal{K}$ and $\mathcal{R}$ into p sections each, such that only one PE works on every block. While the total number of deletions that is possible in a block is bounded by C, the number of keys that map to a block is not. In the worst case, each key must be searched, so that a single block can require up to $O(k \log C)$ operations. In such cases, an *inverse deletion* execution is used to parallelize the work. The inverse deletion for a deletion block b_{del} splits the corresponding batch into one section for each participating PE. A PE then tests whether any keys of its section are in the corresponding subrange of b_{del}. In this case, the PE overwrites the elements in b_{del}. Potential concurrent writes to the same entry of b_{del} are not an issue as each writing thread overwrites the entry with the

null key. After all PEs are done, b_{del} is sequentially compacted by shifting the remaining elements forwards.

During the execution of standard deletions, blocks are treated similarly to the sequential, single-element deletion case. Specifically, the PE first deletes all elements from the blocks in its region. Then, it iterates over the blocks again to reinstate the block size invariant. Whenever a block b does not have enough elements, the PE either steals elements from the previous block b' or it merges b' and b. During this step, each PE only considers block references within its section of $\mathcal{R}$ to avoid race conditions. This leaves all blocks with valid sizes, unless there is only one block b in the entire region of a PE, and $size(b) < C/4$. To handle this case, a global postprocessing step steals or merges such blocks with blocks in the regions of neighboring PEs. This can be done in parallel by pairing the regions of the PEs according to a binary tree over the PEs. As in the insertion case, updated leaves are tracked in a map $\mathcal{M}$ as rebalancing leaves.

Update Phase The update phase is almost identical to the insertion case. The only difference is that the densities must be compared to minimum densities $\tau_{\min}(\ell)$ for a level ℓ. As there are no auxiliary blocks in a deletion operation, they do not have to be considered when the number of blocks in the segment for a leaf is counted. If the root node does not respect its density, a global rebalancing region must be created to trigger the reallocation of the reference PMA with a smaller size.

Rebalancing Phase The rebalancing phase is less complex than the insertion case as there are no new blocks that need to be merged with existing blocks. The target numbering is simply a standard numbering from left to right of all block references that need to be rebalanced. In the distribution part, each PE first receives an interval of target numbers that it has to rebalance. Using a prefix sum, it maps the interval to a set of rebalancing regions. For the first and last rebalancing region, it descends the respective subtree of $\mathcal{T}$ to find the desired indices (instead of extended indices) in $\mathcal{R}$. Again, the linear scan of segments does not need to consider auxiliary blocks. Rebalancing regions that are not the first or last of a PE are rebalanced entirely.

Processing a rebalancing region in the deletion case resembles the procedure in the sequential, single-element case: All block references are already in $\mathcal{R}$ (without extra blocks in $\mathcal{A}$) in an arbitrary distribution. Redistributing them uniformly therefore does not require any merging. Instead, two simple iterations suffice: One to write block references densely to auxiliary memory and another one to write them back uniformly.

4.2.4. Analysis

As the BBPMA adds auxiliary data structures compared to the BPMA, we begin by analyzing its space consumption. Then, we turn towards an amortized analysis of the parallel running times of batch update operations. We build on the analysis of the sequential, single-element variant in Section 4.1.4. The running times for search and scan queries are identical to those of BPMAs.

Space Consumption Like for the BPMA, we show that the BBPMA requires the same asymptotic space as traditional PMAs. To this end, we bound the space consumption of all

additional auxiliary data structures that are used for the BBPMA, but not for the BPMA. The claim is then proven in combination with Lemma 2. Again, the space consumption of a single element is defined as constant.

Lemma 6. *The additional auxiliary data structures required for a BBPMA containing N elements compared to the BPMA have a space consumption of $O(N)$.*

Proof. The rebalancing tree is a binary tree over the segments of the reference array. The number of leaves is $L = O(N/\log N)$ so that the total number of nodes in the binary tree is $O(N/\log N)$. The space consumption per node is $O(1)$. Therefore, the space consumption of the tree is $O(N/\log N)$. The same holds for the update map $\mathcal{M}$ as it stores at most L entries, one for each leaf.

The only remaining auxiliary data structure is the auxiliary storage $\mathcal{A}$ that stores block references of new blocks that are created during a batch-parallel insertion operation for a batch of size k. In the worst case, two auxiliary blocks are created per batch element: If an element has to be inserted into a block b with $size(b) = C$, b is split into two blocks. Both references for the new blocks are stored in $\mathcal{A}$. Therefore, the total number of references for auxiliary blocks is in $O(k)$. After the operation, $N \geq k$ holds as each batch element is either inserted or an element with the same key is already present in the data structure. As each block reference requires space $O(1)$, the total space required for all auxiliary block references is $O(k) = O(N)$. □

Model of Computation We analyze our data structures in the *Parallel Random Access Machine* (PRAM) model [24]. Here, p PEs all access a shared memory. We use the concurrent read, exclusive write (CREW) variant, indicating that concurrent read accesses to the same memory location are allowed, while concurrent write accesses to the same location are forbidden. To analyze the parallel performance of our algorithms, we analyze the total work $W(p)$ and the parallel execution time $T(p)$ for an execution with p PEs for individual phases of the algorithm. The work is the collective number of operations performed by all PEs running the algorithm, while the parallel execution time is the time required to run the algorithm in parallel with the given number of PEs. For the complete algorithm, we additionally analyze the span $S = \inf_p T(p)$ of the entire batch-parallel insertion. The span is the optimal parallel execution time that can be reached with any number of PEs.

Amortized Analysis of Update Operations We start the analysis by extending Theorem 1 to the batch insertion case but prove a bound for the amortized number of blocks that need to be moved during rebalancing, rather than for the execution time. We then use this abstract result to analyze the running time of our batch-parallel insertion algorithm.

Theorem 2. *After the insertion of a sorted batch of size k into a BBPMA with N elements, the amortized number of block references that must be moved to rebalance the reference PMA and merge block references is $O(k \log N)$.*

Proof. Theorem 1 shows that amortized $O(\log N)$ block references must be moved in the single-element insertion case. We extend this argument to the batch insertion case by proving that the batch insertion algorithm is consistent with the proof of Theorem 1. Then, the claimed number of block moves follows by the repeated application of Theorem 1 for

each of the k batch elements. The batch insertion algorithm has three important differences in comparison to the single-element case:

Most importantly, the BBPMA processes elements in batches. Hence, it rebalances only after k elements were inserted, whereas the BPMA performs a rebalancing operation after every single-element insertion. In this proof, we treat each insertion of an element as part of a batch insertion like a single-element insertion in Theorem 1. Opposed to the single-element case, a single batch insertion operation can require the rebalancing of *multiple* rebalancing regions. Consider the set of rebalancing regions $\mathcal{V}'$ (identified by the root nodes of their subtrees) that would be created if a suitable rebalancing region was created for each segment that were changed by the batch insertion, independent of any other necessary rebalancing regions. Theorem 1 shows that there are sufficiently many tokens for the regions in $\mathcal{V}'$. However, rebalancing regions in the BBPMA are not independent of each other. Two cases must be distinguished for a rebalancing region $r(u)$ from $\mathcal{V}'$: First, $r(u)$ can be *unique*, i.e., no part of $r(u)$ is contained in any other region. In this case, $r(u)$ can be treated just as in the sequential case. Second, $r(u)$ can be *dominated* by another region $r(v)$ such that u is in the subtree of v. In that case, $r(u)$ does not need to be rebalanced individually as it will be rebalanced as part of $r(v)$. No tokens are needed for $r(u)$, and all tokens that are required for $r(v)$ are independent of $r(u)$ as they originate from a child of v that does not respect its threshold, while u respects its threshold. No other cases are possible as the regions that are defined by the binary tree cannot partially overlap each other.

The second difference concerns inverse merge execution of a batch insertion. Blocks that are added to the reference PMA can be completely full, seemingly contradicting the assumption made in the proof of Theorem 1 that each block has at most $C/2$ elements at some point. However, the block that is added started completely empty. All elements in it are inserted from the current batch or the block that elements are merged from. In either case, the respective tokens can be added to the accounts of the block, as new tokens that are charged to the batch insertion or as existing tokens from the merge block. Even if the block that is added is entirely full, its account already contains the tokens that might be required for its rebalancing.

Third, managing the block references is more complicated in BBPMAs. As there might not be enough space within $\mathcal{R}$, block references are written to $\mathcal{A}$ instead. Therefore, we extend the argument made in Theorem 1. During the rebalancing of a region $r(u)$ of some node u in the rebalancing tree, tokens can also be taken from the accounts of all auxiliary blocks that are children of blocks in $r(u)$. Beyond this, it is irrelevant if a block reference is stored in $\mathcal{R}$ or $\mathcal{A}$ as we only consider the *number* of blocks that needs to be moved. The additional effort for the merging of block references from $\mathcal{R}$ and $\mathcal{A}$ is considered in Lemma 10. □

To analyze the parallel performance, we consider the five phases of the batch insertion (see Section 4.2.2) individually in Lemmas 7 to 11. Theorem 3 summarizes the results and analyzes the total span of a batch-parallel insertion into the BBPMA.

Lemma 7. *The insertion distribution phase for inserting a sorted batch of size k into a BBPMA with N elements requires work $O(p(\log N + \log k))$ and parallel time $O(\log N + \log k + \log p)$.*

Proof. Each PE uses time $O(\log(N/\log N)) = O(\log N)$ to find its splitter in $\mathcal{R}$ and time $O(\log k)$ for the exponential search on the batch. By comparing neighboring PEs hierarchically, work $O(p)$ and parallel time $O(\log p)$ is needed to ensure that all PEs work on disjoint sections of $\mathcal{R}$. A PE that participates in an inverse merge execution spends parallel time $O(\log C) = O(\log \log N)$ to find the range of elements that it merges from the merge block. □

Lemma 8. *The insertion execution phase for inserting a sorted batch of size k into a BBPMA with N elements requires amortized expected work $O(k \log N)$ and amortized expected parallel time $O(k/p \log N)$.*

Proof. We first consider the work that is required to insert individual elements, starting with standard merge executions. Consider an element e that is inserted into block b. In the worst case, it is the only element for b. The exponential search on $\mathcal{R}$ for b requires time $O(\log(N/\log N)) = O(\log N)$. Time $O(\log N)$ is required to insert the element into the block and split the block, if necessary. If an element e is inserted by inverse merge execution, no search on the PMA or the merge block is necessary as the information was already computed in the insertion distribution phase. At most $O(\log N)$ elements need to be merged from the existing block so the new block can be written in time $O(\log N)$. In both cases, new blocks can be added to $\mathcal{A}$ in $O(1)$. Using a hash map, the respective segment can be added to $\mathcal{M}$ in amortized expected time $O(1)$. The total work for all element insertions is $O(k \log N)$.

To prove the bound on parallel time, we need to prove that each PE processes $\Theta(k/p)$ elements. The splitters are chosen from the batch with equal distance so that each PE starts with $\Theta(k/p)$ elements. Unless multiple splitters are mapped to one block, the maximum imbalance in the work between the PEs is a constant factor. Suppose instead that q splitters map to one block b. In standard insertion execution, all elements that need to be inserted into b would be processed by a single PE, possibly impairing the work balance by a factor of $q = \Omega(1)$. In this case, the distribution algorithm ensures the usage of inverse merge execution, so that the work can be evenly distributed between $\Theta(q)$ PEs. The PEs then independently process their section of the elements in parallel. Therefore, the maximum imbalance is bound to a constant factor, even if the distribution of batch elements to the blocks is highly skewed. □

Lemma 9. *The update phase for inserting a sorted batch of size k into a BBPMA with N elements requires amortized expected work $O(k \log N)$ and amortized expected parallel time $O((1 + k/p) \log N)$.*

Proof. We first consider the upwards pass. Let m be the number of leaves that need to be updated (recounted or rebalanced) in $\mathcal{T}$. It consists of the number of leaves that were changed during the current insertion phase and the number of leaves that were rebalanced in the previous rebalancing phase. The insertion phase changes at most $O(k)$ leaves. Theorem 2 bounds the number of block references that need to be moved during the previous rebalancing phase. Each moved block reference requires the update of at most 2 segments. Therefore, the number of updated leaves is bounded in the same way:

$$m = O(k \log N).$$

On the lowest level, each leaf requires recounting a segment of constant length, requiring total work of $O(m)$. On all higher levels combined, the number of inner nodes that need to be updated is at most

$$\sum_{i=1}^{h} \frac{m}{2^i} = O(m).$$

Each node requires $O(1)$ time to update and check whether it needs to be marked as potential rebalancing region. Using a hash map, adding the parent node for the next level takes expected amortized work $O(1)$. Together, this yields the total work bound stated above.

Levels of the tree are updated sequentially, requiring execution time of $O(h)$. On each level, individual nodes are processed in time $O(1)$. As the work is balanced between the PEs, the global time of a PE to update the nodes that it processes across all levels is $O(m/p)$. This yields the parallel time stated above.

The downwards pass only considers nodes that were already considered in the upwards pass. The work per node is $O(1)$ so that the total work and parallel time are identical to that of the upwards pass. □

Lemma 10. *The rebalancing distribution phase for inserting a sorted batch of size k into a BBPMA with N elements requires amortized expected work*

$$O\left(k \log N + p(\log k + \log\log N)\right)$$

and amortized expected parallel time

$$O\left(k/p \log N + \log p + \log k + \log\log N\right).$$

Proof. The update phase finds the minimum set of rebalancing regions. Combined with Theorem 2, the number of rebalancing regions is bounded by $O(m) = O(k \log N)$. The parallel prefix sum of their sizes can be computed in work $O(m)$ and parallel time $O(m/p + \log p)$ [13].

Each PE performs $O(1)$ binary searches on the prefix sum that require parallel time $O(\log m) = O(\log(k \log N)) = O(\log k + \log\log N)$. The subtree for any rebalancing region contains at most $O(m)$ leaves so that it can be descended in time $O(\log m)$. Each PE descends subtrees for at most two regions. The final linear search within a segment is possible in constant time. □

Lemma 11. *The rebalancing execution for inserting a sorted batch of size k into a BBPMA with N elements requires amortized expected work $O(k \log N)$ and amortized expected parallel time $O(k/p \log N)$.*

Proof. The work for moving a block reference is $O(1)$ and $O(k \log N)$ block references need to be moved. Due to the rebalancing distribution, the work is balanced between the PEs and no further searches are necessary. Because of the density invariants, iterating the reference PMA generates no asymptotic overhead. Theorem 2 bounds the total work and — due to the work balance — the parallel time. □

Theorem 3. *Assuming $p \leq k$, inserting a sorted batch of size k into a BBPMA with N elements and p PEs requires amortized expected work*

$$O(k \log N + p \log k)$$

and amortized expected execution time

$$O\left(\frac{k}{p} \log N + \log p + \log k\right).$$

The amortized expected span is

$$O(\log N + \log k).$$

Proof. Proven by the combination of Theorem 2 and Lemmas 7 to 11. The span is achieved with $p = k$. □

4.3. Batch-Parallel Buffered Packed Memory Array with Global Rebalancing

Our BBPMA uses the block indirection of the BPMA and performs batch-parallel insertions. We showed that this enables rebalancing in work that is dominated by $O(k \log N)$, rather than $O(k \log^2 N)$ in the only previous solution for a batch-parallel insertion into a PMA-based data structure. On the other hand, rebalancing for batch-parallel insertions comes at significant overhead — both regarding the complexity of the data structure as well as practical running times. We study whether a PMA data structure with the block indirection still requires the complete, parallel rebalancing that is used in the BBPMA. To this end, we propose a simplified variant, the *Global Rebalancing Batch-Parallel Buffered Packed Memory Array* (GBPMA). Similar to the BBPMA, it uses the block indirection and supports batch-parallel update operations. However, it avoids the intricate rebalancing scheme using *global rebalancing*. In simple terms, global rebalancing replaces frequent local rebalancing operations by simple global rebalancing operations, while trying to reduce their frequency.

4.3.1. Data Structure

Analogous to the BBPMA, the GBPMA stores elements in blocks of capacity $C = \Theta(\log N)$ and maintains references to the blocks in a reference PMA $\mathcal{R}$. It uses the same invariants as the BBPMA but requires a block to contain at least $C/2$ elements as we do not consider deletions. Before defining the remaining parts of the data structure, we give an overview over global rebalancing. The objective is to perform insertions without any rebalancing as long as possible. Once any rebalancing is needed locally, the entire reference PMA is rebalanced globally with additional gaps, so that the cost can be amortized over the next insertions. The insertion distribution phase proceeds analogous to the BBPMA. Afterwards, each PE checks whether it can guarantee that its section of the reference PMA has sufficient space for the maximum number of blocks that it can generate during the insertions. If this is the case for all PEs, no rebalancing is necessary. If any PE might need more space for its blocks than its section of $\mathcal{R}$ has, a global rebalancing operation is triggered, which enlarges $\mathcal{R}$ so that it has a constant *growing factor* $g = \Theta(1)$ more space than needed.

Contrary to the BBPMA, there is no rebalancing tree, as no rebalancing regions are needed. Instead, global rebalancing requires fast range queries for the number of elements that are stored in a range of the reference PMA. To this end, the GBPMA uses the *size manager* $\mathcal{S}$, a simple linear data structure that maintains the collective number of *elements* that is stored in the blocks of each segment of $\mathcal{R}$. The PEs use the size manager to check whether their blocks fit into their region of $\mathcal{R}$. In total, the GBPMA is defined as a two-tuple

$$(\mathcal{R}, \mathcal{S})$$

consisting of the reference PMA $\mathcal{R}$ of size P and the size manager $\mathcal{S}$. As before, $M = \Theta(N/\log N)$ holds. However, global rebalancing does do not guarantee any bounds regarding P. As the layout of $\mathcal{R}$ is identical, search and scan queries work just as in the BPMA. The information stored in $\mathcal{S}$ can be used for rank, select and counting queries.

4.3.2. Batch-Parallel Insertion

The batch-parallel insertion operation proceeds in three phases. The first phase is identical to the insertion distribution phase of the BBPMA described in Section 4.2.2. After the insertion distribution, the *size estimation* phase checks whether the insertions can be performed in-place in the current reference PMA, or whether the insertions need to be performed *out-of-place* in a larger reference PMA. Finally, the insertion execution inserts the batch in the respective way.

Size Estimation In the insertion distribution, each PE is assigned a section of the reference PMA $\mathcal{R}_i$ and a corresponding section of the batch $\mathcal{B}_i$. In the size estimation phase, each PE checks whether it can guarantee that the block references for all blocks that it will create in the insertions fit into $\mathcal{R}_i$. To this end, the size estimation phase computes the maximum number of elements n that must be contained in all blocks after the insertions. From there, it computes the maximum number of blocks m that can be required for n elements. If $\mathcal{R}_i$ has sufficient space for m blocks, the PE could proceed with in-place insertion. The insertion is executed in-place if all PEs can insert in-place. Otherwise, out-of-place execution is used.

To compute n, the number of elements in $\mathcal{R}_i$ and $\mathcal{B}_i$ is added. While the number of elements in $\mathcal{B}_i$ is clear, the number of elements in $\mathcal{R}_i$ is computed with a range query using $\mathcal{S}$. To this end, the PE aggregates the number of elements for all segments in $\mathcal{S}$ that are entirely contained in $\mathcal{R}_i$. Then, it adds the number of elements from segments that are only partially contained in $\mathcal{R}_i$.

Once an upper bound to the total number of elements n is established, an upper bound to the required number of blocks m is computed using the minimum block size. Both insertion variants ensure that each block contains at least $C/2$ elements. Therefore, the number of required blocks is at most

$$m = \frac{n}{C/2} = \frac{2n}{C}.$$

Insertion Execution: In-Place Variant Consider a PE that received a section of the reference PMA $\mathcal{R}_i$ and a corresponding section of the batch $\mathcal{B}_i$. All PEs asserted in the size estimation that their section of the reference PMA has sufficient space for the insertions. The in-place insertion execution proceeds similarly to the insertion execution of the BBPMA, but there are two important differences. As it is certain that there is enough space in $\mathcal{R}_i$, block references are always written to $\mathcal{R}$, and there is no need for auxiliary block storage. To ensure that all block references can be added, existing blocks must be shifted away, if necessary. This has to be possible in both directions to ensure that all cells in $\mathcal{R}_i$ are usable.

The second difference is that the entries in $\mathcal{S}$ must be kept up-to-date. Instead of using an update phase, changes in $\mathcal{R}$ are reflected in $\mathcal{S}$ on the fly. Changes can arise from two operations: First, simply adding elements to an existing block or adding a new block reference at a previously empty index in $\mathcal{R}$ increases the number of elements in the segment accordingly. Here, $\mathcal{S}$ is updated by simply adjusting the value for the respective segment. The more complex update is necessary if block references are shifted within $\mathcal{R}$ to free an entry for a new block reference. In this case, the counts for all segments that are affected by the change must be updated. As block references are always only shifted by

one position, this can be done by considering only those block references that crossed a segment border. Their number of elements must be subtracted from one and added to the count of the other segment, depending on the direction of the shift.

Insertion Execution: Out-of-Place Variant The out-of-place insertion execution is used if any PE was unable to guarantee that it can perform all insertions within its section of the reference PMA. A new reference PMA $\mathcal{R}_{\text{new}}$ is allocated with a capacity that is a factor of g larger than the sum of the total maximum number of blocks that the PEs can write. Then, each PE is assigned a section of $\mathcal{R}_{\text{new}}$ so that it can write all its block references into it.

Now, the insertion execution can proceed as described for the in-place variant above. However, it now has to write *all* block references — of new and *existing* blocks — to $\mathcal{R}_{\text{new}}$. To this end, $\mathcal{R}_i$ is iterated linearly, rather than with exponential search, and all references for blocks that do not receive new batch elements are copied to $\mathcal{R}_{\text{new}}$. Each PE writes elements densely to its section of $\mathcal{R}_{\text{new}}$. Once all PEs are done, $\mathcal{R}$ is resized to the same size as $\mathcal{R}_{\text{new}}$, and each PE writes its block references back from $\mathcal{R}_{\text{new}}$ to $\mathcal{R}$, distributing them uniformly in its section. After this operation, $\mathcal{S}$ is recomputed.

5. Implementation Details

This chapter presents details about our implementation of the BBPMA and the GBPMA. We first outline some simplifications in the implementation compared to the description in Section 4.2.2 and describe the fundamental data structures of our implementation. Finally, we specify how our insertions are parallelized. Our data structures are implemented in C++20. The implementation is available at `https://github.com/moritzpotthoff/Buffered-Packed-Memory-Array`. Both BBPMA and GBPMA are able to store elements of an arbitrary type for which a `keyOf()` function is available that extracts the key of the element.

Simplifications in the Implementation We implement batch-parallel insertions, search queries, and scan queries, but do not consider deletions. The description uses a block size $C = \mathcal{O}(\log N)$. For simplicity and improved cache efficiency, we instead use a sufficiently large constant $C = \mathcal{O}(1)$ that is greater than $\log N$ for practical values of N. The constant block size also avoids the reconstruction of blocks that is otherwise needed if the number of elements changes sufficiently. Since deletions are not considered, out implementation ensures a minimum block size of $C/2$ instead of $C/4$ in the description. As block sizes are typically small enough, we use linear search instead of binary search on blocks. Section 6.2.1 contains details concerning the specific values chosen for C.

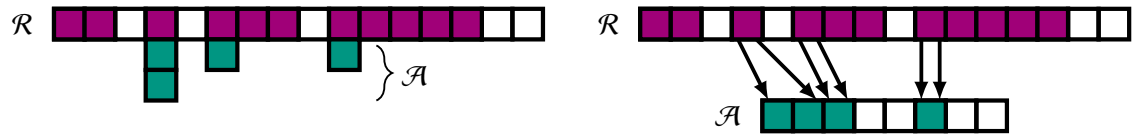
At two points, our implementation proceeds sequentially instead of parallelly: In the insertion distribution phase, we sequentially check whether any PEs work on the same block. In the rebalancing distribution phase, we compute the prefix sum over the rebalancing regions sequentially.

Finally, the update phase only uses the upwards pass through the rebalancing tree. The upwards pass already generates a rebalancing region for each node that is marked as potential rebalancing node in the description. All dominated rebalancing regions are then filtered out in a post processing step. A rebalancing region $r(u)$ is dominated if u is in the subtree of another node v for which a rebalancing region was created. This is checked by ascending the path from u to the root of $\mathcal{T}$. If any node on the path is marked as potential rebalancing tree, $r(u)$ does not need to be rebalanced. The post processing step checks this in parallel for all rebalancing regions. While this procedure is asymptotically less efficient, it saves the update phase from having to sequentially scan all levels twice, which is hard to parallelize efficiently in practice.

Data Structures Our reference PMA is implemented as an array $\mathcal{R}$, where each entry is a structure containing head and size of, as well as the pointer to its block. Storing the size in $\mathcal{R}$ (rather than with the blocks) improves memory efficiency of the update phase. A global pointer to the last entry of $\mathcal{R}$ that is currently used improves efficiency for searches and scans. While a rebalancing region is rebalanced, block references are densely stored in a

secondary array $\mathcal{R}_{\text{swap}}$ of the same capacity as $\mathcal{R}$. Here, each PE uses the same region that it will write back to in $\mathcal{R}$.

The container for references of auxiliary blocks $\mathcal{A}$ is also an array. It is stored persistently with the data structure to avoid repeated memory allocations. Its capacity is adjusted using the bound on the number of auxiliary blocks shown in the proof of Lemma 6. Analogously, each PE is allocated a sufficiently large section of $\mathcal{A}$ depending on the number of batch elements that it processes. It is crucial that auxiliary blocks can be found from the cell in $\mathcal{R}$ from which they were created. The description uses a simplified representation of the auxiliary blocks. In practice, an additional array $\mathcal{P}$ of the capacity of $\mathcal{R}$ maintains the respective range of auxiliary blocks in $\mathcal{A}$ using two pointers for each cell in $\mathcal{R}$. A comparison between the two representations can be found in Figure 5.1. The pointers also allow to efficiently query the number of auxiliary blocks at each cell of the reference PMA. During an inverse merge execution, multiple PEs write the blocks that make up the auxiliary blocks of a single cell. To this end, each PE first receives a region of the auxiliary block array that it writes to. Afterwards, the blocks in combined region for the cell are compacted and the pointers of the cell are updated.



(a) Simplified representation (b) Physical representation

Figure 5.1.: Representation of a reference PMA $\mathcal{R}$ with additional auxiliary blocks $\mathcal{A}$. In the simplified representation, we draw auxiliary blocks below their parent blocks (Subfigure a). In the physical representation (Subfigure b), pointers for each cell of $\mathcal{R}$ define the range of auxiliary blocks in $\mathcal{A}$.

For each size P of the array used for the reference PMA, the rebalancing tree is static. Therefore, it is stored inline in a fixed-size array. The update map $\mathcal{M}$ is implemented using a concurrent bucket hash map approach similar to that used in the implementation of CPMA [26]. It uses $O(p)$ sequential hash maps as buckets and distributes elements to the hash maps using the hash values of their keys. Each hash map is guarded by a lock. Preliminary practical evaluations show good distribution of the elements across the buckets and low contention. During the update phase, elements in $\mathcal{M}$ are processed in parallel by parallelizing over the buckets. Each PE repeatedly receives a bucket and processes all entries in it sequentially. Each block b is an `std::array` of fixed capacity C. The block size is only maintained in the reference entry for b in $\mathcal{R}$.

Memory Management Depending on N and k, the data structures need to increase their capacity. The auxiliary block storage can simply be resized before the insertion execution begins. To avoid having to copy their contents, resizing $\mathcal{R}$, $\mathcal{R}_{\text{swap}}$, $\mathcal{T}$ and $\mathcal{P}$ is more involved. First, it can be observed that it is only necessary to resize $\mathcal{R}$ if a rebalancing region was created for the root node of $\mathcal{T}$. In this case, the entire content of $\mathcal{R}$ will be rewritten during the rebalancing execution. Therefore, we interweave the rebalancing execution and the resizing processes. If resizing is necessary, the rebalancing distribution

phase starts by resizing $\mathcal{R}_{\text{swap}}$ to the new size. It uses a size that is a constant factor greater than the number of block references that need to be stored. See Section 6.2.1 for details. At this point, $\mathcal{R}_{\text{swap}}$ is empty, so that no entries need to be copied. After the block references were written from $\mathcal{R}$ to $\mathcal{R}_{\text{swap}}$, $\mathcal{R}$ and $\mathcal{P}$ can be reallocated to the same size as $\mathcal{R}_{\text{swap}}$. Additionally, the array that stores $\mathcal{T}$ is reallocated accordingly. The information stored in $\mathcal{T}$ does not need to be recomputed here: In the next step, block references will be written from $\mathcal{R}_{\text{swap}}$ back to $\mathcal{R}$. The leaf of any segment that receives a block reference will be added to $\mathcal{M}$ as a recounting node. Therefore, the update phase of the following batch insertion will recompute the necessary parts of $\mathcal{T}$, together with additional changes from the insertions of that operation.

Parallelization Our implementation is parallelized using OpenMP. The distribution phases store information about the tasks for each PE, and each OpenPM thread executes its respective tasks. We pin threads to cores for improved performance and reproducible results. During the batch-parallel insertion, threads mostly write to disjoint regions of all data structures. Therefore, no synchronization is needed. The only data structure in which multiple PEs access the same data is $\mathcal{M}$. Its synchronization mechanisms are described above. We use oneAPI Threading Building Blocks (oneTBB)[1] in our implementation. Its concurrent vector stores the rebalancing regions that are generated in the update phase. As blocks can be allocated concurrently during the insertion execution phase, we use the `scalable_allocator` from oneTBB to improve efficiency. All other reallocations described above proceed sequentially.

[1]Formerly known as Intel TBB. Documentation available at `https://spec.oneapi.io/versions/latest/elements/oneTBB/source/nested-index.html` (last visited 07/17/2023).

6. Evaluation

We analyze the performance of our batch-parallel data structures using an extensive experimental evaluation. First, we describe our experimental setup including experiments and different inputs. Then, we present an experimental evaluation of the BBPMA as well as the GBPMA and compare their performance to that of two competitors, one based on search trees and one based on PMAs without the block indirection.

6.1. Experimental Setup

We evaluate all data structures using implementations in C++20 that are compiled with GCC 10.3.0 with optimization flags `-march=native` and `-O3`. All evaluations were run on a machine with an AMD EPYC Rome 7702P processor with 64 cores (128 logical threads) on a single socket, clocked at 2.0–3.35 GHz, with 1 024 GB of DDR4 memory (3 200 MHz) and 256 MB of L3 cache. The L3 cache is distributed into 16 sections of 16 MB each. Each section is shared by four cores.

Implementations and Competitors We compare our data structures to two competitors: The Parallel Search Tree (PST) by Akhremtsev and Sanders [2] and the CPMA by Wheatman et al. [26]. A description of the data structures can be found in Sections 3.2 and 3.3. For both competitors, we use implementations provided by the authors.[1] We use the configuration provided by the authors unless it is noted otherwise. For the CPMA, we use a variant without compression in PMA segments and without the optimizations for search queries that are described in [27]. While this is not the optimal variant of the CPMA, it resembles traditional PMAs more closely so that we can better examine the influence of our block indirection. Additionally, it is unclear how effective the compression is for general workloads, where large, unsorted values need to be stored with the elements. As it is uncompressed, we identify this variant of the CPMA as *uCPMA* in our evaluations for clarity. The implementation of the parallel search tree does not allow sequential execution, so we only consider it for at least 2 threads. For the BBPMA, we use the implementation that is outlined in Chapter 5. The code is available at `https://github.com/moritzpotthoff/Buffered-Packed-Memory-Array`.

Evaluation Inputs In the following, we introduce the six distributions that our inputs are based on. An approximation of the density functions for relevant distributions can be found in Figure 6.1. The uniform and normal distributions use the respective distributions

[1] The implementations are available at `https://git.scc.kit.edu/akhremtsev/ParallelBST` and `https://github.com/wheatman/Packed-Memory-Array`, respectively (both last visited 07/17/2023).

provided in the C++ standard library. For the zipf distribution, we use an implementation by Lucas Lersch that is published under the MIT License.[2]

- *uniform*: A simple uniform distribution.
- *normal*: A normal distribution with a standard deviation of 2 % of the length of the key range.
- *dense normal*: A normal distribution with a standard deviation of 0.2 % of the length of the key range.
- *zipf*: A zipfian distribution with a skew parameter of 0.99.
- *ascending*: A uniform distribution that generates keys that are sorted across the batches: If two batches $\mathcal{B}_1$ and $\mathcal{B}_2$ are processed in this order, the ascending distribution ensures that all keys from $\mathcal{B}_2$ are greater than all keys in $\mathcal{B}_1$.
- *descending*: Like the ascending distribution, but all keys from $\mathcal{B}_2$ are smaller than all keys in $\mathcal{B}_1$. Within each batch, keys are sorted in ascending order.

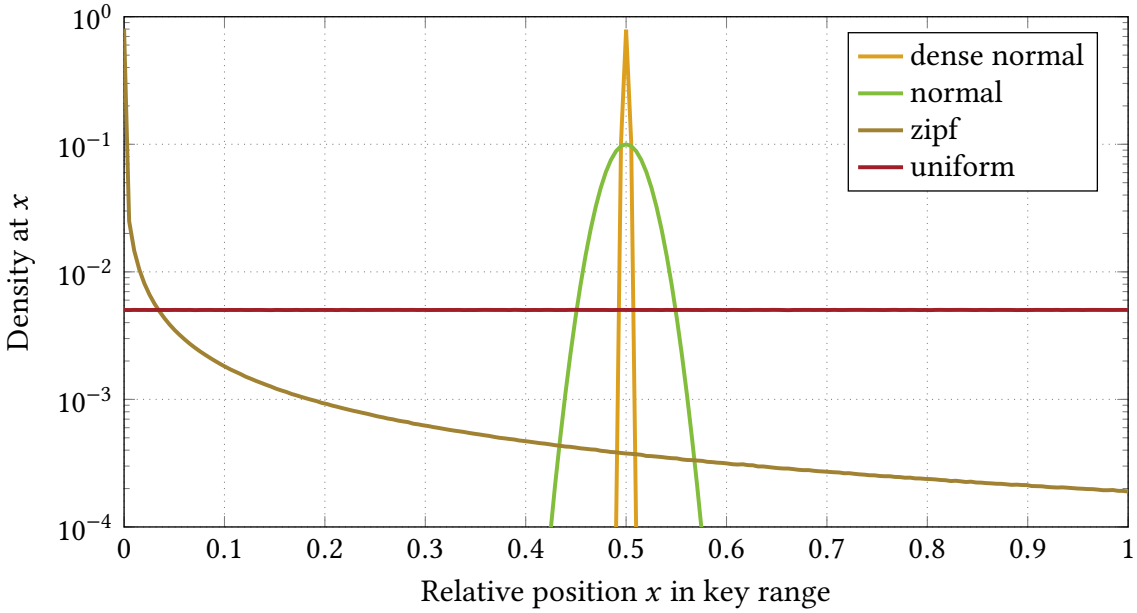


Figure 6.1.: Approximation of the density functions for the distributions. The ascending and descending distributions are omitted as their keys follow the uniform distribution.

We generate an *input* for our evaluations by combining two distributions: The *prefill distribution* is used to prefill the data structure before the measurements, while the *measured distribution* generates elements for the operations that are used during the measurements of running times. We use eight different inputs, which are described in Table 6.1. The first

[2]Available at `https://github.com/llersch/cpp_random_distributions` (last visited 07/17/2023).

six inputs are obtained by combining uniform prefill with the respective distribution as measured distribution. They are identified by the name of their measured distribution. Additionally, we add two inputs, *ascending** und *descending** that also use ascending or descending *prefill*. In particular, the ordering between batches is also maintained between batches of the prefill and batches that are measured. The standard ascending and descending inputs always insert an entire batch into a small section relative to the uniform prefill, where the section slides across the data structure with the batches. Opposed to that, the starred variants always insert an entire batch at the end (ascending*) or beginning (descending*) of the current data structure.

Input	Distribution		Locality	Insertion point
	Prefill	Measured		
uniform	uniform	uniform	very low	everywhere
normal	uniform	normal	low	center
dense normal	uniform	dense normal	medium	center
zipf	uniform	zipf	high	left
ascending	uniform	ascending	high	moving
descending	uniform	descending	high	moving
ascending*	ascending	ascending	very high	right
descending*	ascending	descending	very high	left

Table 6.1.: Description of the prefill and measured distributions for the eight inputs with an informal characterization of their locality and the point at which elements from the measured distribution are inserted relative to the prefill distribution.

Experiments We evaluate batch-parallel insertions, searches and scans. In the experiments, we use the data structures to store only keys (no mapped values) and use 64 bit integers as keys. We use keys in the range $[1, 10^{13}]$ for elements, and the key 0 is our null key. All inputs are generated ahead of the experiments so that operations on the data structure can be run without interruptions. For evaluations of the insertion operation, we report the average running times of five iterations, measuring the time required to insert batches from the measured distribution. Running times for prefilling are not measured in the experiments. For search and scan queries, we use the uniform input. Search queries search for keys drawn from the uniform distribution. Scan queries scan a key range $[a, a + \ell)$ where a is a key that is generated by the uniform distribution and the range length ℓ is generated by the zipf distribution. This ensures that scan queries of a variety of range lengths are generated.

The experiment code can be found in our implementation. We use *SqlPlotTools* by Timo Bingmann to create the tables and plots for experiment results.[3] It is published under the GNU GPL v3 license.

[3]Available at `https://github.com/bingmann/sqlplot-tools` (last visited 07/17/2023).

6.2. Batch-Parallel Buffered Packed Memory Array

We first present results for the tuning of important parameters before evaluating the performance of the tuned variant in depth. We select two sets of parameters, the *insertion configuration* and the *scan configuration*.

6.2.1. Parameter Exploration

The performance of a BBPMA is mainly influenced by the block size C, the segment size S and the density configuration. For insertions, the density configuration is defined by two parameters, the maximum density on the highest level $\tau^h_{\max}$ and the growing factor g. The growing factor is multiplied to the number of block references that need to be stored to determine the new size of $\mathcal{R}$ when the reference PMA is reallocated.

We evaluate the performance of our data structure regarding batch-parallel insertions, scans and search queries for a wide range of combinations of the three tuning parameters mentioned above. In this way, we are able to consider interdependencies between the parameters. We repeat the experiments using uniform, dense normal and zipf inputs to ensure that the tuned data structure performs well on a wide range of inputs without being overfitted to a specific scenario. The five remaining inputs enable the validation of the universal performance using scenarios that were not used for the parameter tuning.

The block size is the most important tuning factor as it characterizes our main contribution. Therefore, we evaluate the data structure for all combinations of block sizes C and segment sizes S from

- $C \in \{16,\ 32,\ 64,\ 128,\ 256,\ 512,\ 1\,024, 2\,048,\ 4\,096\}$ and
- $S \in \{256,\ 512,\ 1\,024,\ 2\,048,\ 4\,096\}$.

Preliminary experiments showed that the density parameters have relatively little influence on the performance. Therefore, we only consider three density configurations. We use symbolic names α, β and γ for simplicity:

- Density configuration α: $\tau^h_{\max} = 0.9,\ g = 1.8$
- Density configuration β: $\tau^h_{\max} = 0.9,\ g = 1.2$
- Density configuration γ: $\tau^h_{\max} = 0.7,\ g = 1.8$

For the parameter tuning evaluations, we use a batch size of 10^6 elements and insert 100 batches as prefill and 100 batches during the measurements. Afterwards, we perform 1 000 search and scan queries each. For search queries, we report the average running time of a query. We distinguish scan queries into two categories. *Short* scan queries scan up to 1 000 elements while *long* scan queries scan more elements. For both, we report the throughput of scanned elements per second. The full results including search and scan queries (omitting some irrelevant configurations for readability) are given in Tables A.1 to A.9.

Analysis We start the analysis of the results with a focus on insertions. Figure 6.2 gives an overview of the insertion performance for the density configurations α and γ. A preliminary evaluation showed that density configuration β has no advantages. As we are interested in a scalable configuration, we consider the case for $p = 64$ threads. The plots show significant differences in the behavior for the different inputs. On the contrary, the two density configurations behave very similarly, except for the zipf input. Here, the optimal case for density configuration α is around 7 % better than for density configuration γ. Therefore, we consider density configuration α for the remainder of this section.

For the uniform input, block size 128 is optimal for all segment sizes, with 10–28 % higher throughputs than other block sizes depending on the segment size. The best results are achieved with segment sizes 1 024 and 2 048 which achieve almost identical throughput. The differences between the segment sizes are small: For a block size of 128, the optimal segment size of 2 048 is only around 12 % faster than the worst evaluated segment size 4 096. On the dense normal input, the best throughput is achieved with a block size of 1 024, but block sizes 128, 256 and 512 are only between 3 % and 6 % slower for all segment sizes. The best configuration overall uses block size 1 024 and segment size 512. The throughput for a block size of 128 and segment size 1 024 is only 3.5 % lower. The dense normal input has much more focused insertions than the uniform input. Here, the reduced rebalancing work gained from larger block sizes seems to make up for more expensive operations on the blocks. The zipf input again achieves the best throughput for a block size of 128 for all competitive segment sizes, beating the next best segment size by 11–24 %. The segment sizes 256 and 512 achieve practically identical optimal throughputs. The very focused insertions of the zipf input seem to prefer smaller segment sizes. The performance gain is likely achieved because the size of rebalancing regions can be controlled more finely and counting a segment is faster. Larger segment sizes are prone to generating rebalancing regions that are unnecessarily large.

This exemplary study showed that a small set of configurations seems to be competitive overall. We use the configuration with density configuration α, block size 128 and segment size 1 024 as a *candidate* for our final configuration. To ensure that it provides the best possible tradeoffs, we study more detailed results in Table 6.2. For each input, it shows the insertion throughput results for those configurations that are competitive in one of the scenarios considered in Figure 6.2. Additionally, it contains the relative throughput compared to the candidate configuration. As no combination with density configurations β or γ is dominant, we focus on density configuration α for simplicity.

For the uniform input, the configuration with $C = 128$ and $S = 2\,048$ is optimal, while the candidate configuration is less than 1 % slower. The dense normal input favors the configuration with $C = 1024$ and $S = 512$. It is 3.6 % faster than the candidate. However, it is almost 24 % slower for the uniform input. The zipf input achieves optimal throughput with a configuration of block size 128 and segment size 512, which is 2.3 % faster than the candidate. For the uniform input, it is almost 3 % slower than the candidate.

To select a universal configuration for our data structure, we need to weigh the importance of the three inputs we used. The uniform input is a standard case and resembles many inputs that occur in practice. Conversely, the dense normal input models input distributions with insertions that are more focused to a specific key range. Finally, the

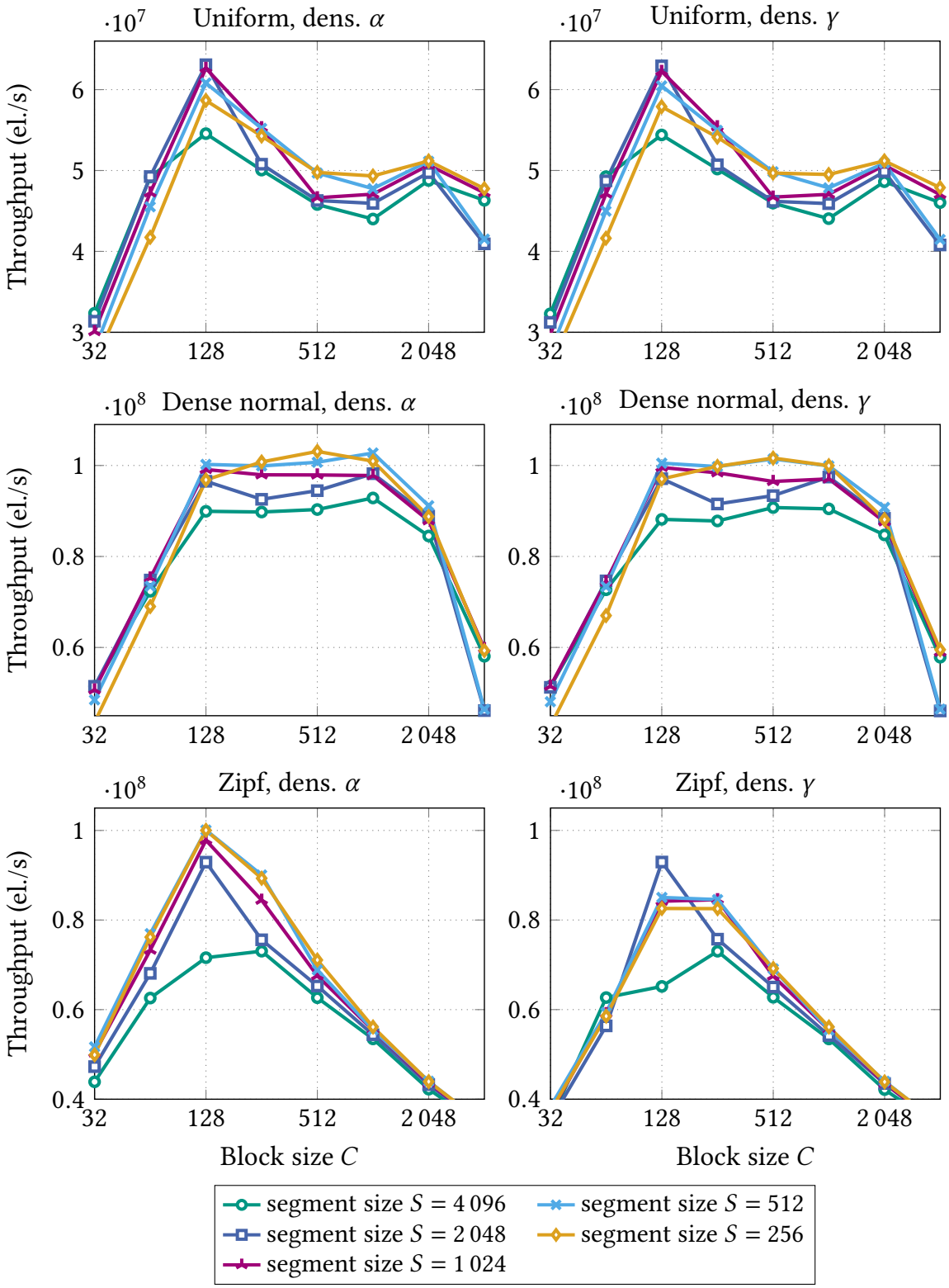


Figure 6.2.: BBPMA parameter tuning results for three inputs and the density configurations α ($\tau^h_{\max} = 0.9$, $g = 1.8$, left) and β ($\tau^h_{\max} = 0.7$, $g = 1.8$, right).

Input	Size		Throughput	Rel. thr.
	Block	Segment	($\cdot 10^6$ el./s)	(% opt.)
uniform	128	512	60.82	97.10
		1 024	62.63	100.00
		2 048	**63.08**	**100.71**
	1 024	512	47.76	76.25
dense normal	128	512	100.23	101.13
		1 024	99.11	100.00
		2 048	96.53	97.39
	1 024	512	**102.74**	**103.66**
zipf	128	512	**100.08**	**102.33**
		1 024	97.80	100.00
		2 048	92.92	95.01
	1 024	512	55.49	56.74

Table 6.2.: Summary of the BBPMA parameter tuning results for density configuration α. Insertion throughput results are for $p = 64$, the relative performance is compared to the optimum configuration for the input. Bold values are the best for their input.

zipf input is very skewed and fewer practical workloads will resemble it. Therefore, we consider the uniform input as most important, and the zipf input as least important.

Based on this consideration, optimal performance on the uniform input is crucial. While the candidate configuration was not optimal for each input, it achieves consistently high throughputs for all inputs. In particular, it offers better tradeoffs than other configurations that were competitive for certain scenarios. Therefore, we define the *insertion configuration* as follows: It uses density configuration α ($\tau^h_{\max} = 0.9$, $g = 1.8$), block size $C = 128$ and segment size $S = 1\,024$.

Focus on Scan Performance So far, we only considered the insertion throughput in our evaluations. The configuration derived from that allows for very fast insertions but does not consider search or scan speed. As shown in Appendix A.1, larger block sizes significantly improve performance for search or scan queries, independent of the other tuning parameters. The larger blocks store more elements densely, so that more elements can be scanned very efficiently before a less efficient linear search for the next block reference is needed on the reference PMA. Moreover, the reference PMA is smaller which improves search performance. The results indicate that a wide variety of tradeoffs between insertion performance and performance for scans and searches can be realized.

Besides the insertion configuration described above, we choose a *scan configuration* that favors scan performance while only impairing insertion performance to a reasonable degree. To this end, Table 6.3 shows results for all block sizes that achieve at least 75 % of the performance of the insertion configuration. As the influence of the block size on the scan performance is mostly independent of the input, the segment size, and the density

configuration, we focus on the uniform input, density configuration α and the best segment size for each block size with regards to insertion throughput.

Size		Thr. ($\cdot 10^6$ el./s)			Rel. perf. (% opt.)		
Block	Segment	Ins.	Scan		Ins.	Scan	
		$p = 64$	short	long	$p = 64$	short	long
64	2 048	49.26	15.13	256.57	78.64	90.19	86.09
128	**1 024**	**62.63**	**16.77**	**298.03**	**100.00**	**100.00**	**100.00**
	2 048	63.08	16.48	298.64	100.71	98.27	100.20
256	1 024	55.29	19.34	424.57	88.28	115.30	142.46
512	256	49.76	21.81	565.31	79.45	130.03	189.68
1 024	256	49.33	28.32	672.76	78.76	168.80	225.74
2 048	**256**	**51.19**	**28.82**	**842.94**	**81.73**	**171.84**	**282.84**
4 096	256	47.76	23.57	977.49	76.25	140.54	327.98

Table 6.3.: BBPMA Parameter tuning results for the uniform input, density configuration α and $p = 64$. We consider all block sizes for which an insertion throughput of at least 75 % of the value for the insertion configuration can be achieved. For each block size, we select the optimal segment size with regards to insertion throughput. Additionally, the table contains the insertion configuration (upper bold line). The scan configuration is the lower bold line. The relative performance is in comparison to the respective value achieved by the insertion configuration.

The results show that it is possible to increase scan query performance drastically (by up to 3.27x) while still maintaining acceptable insertion throughput. For long scan queries, larger blocks monotonically enable faster scans, whereas a saturation seems to be reached at block size 2 048 for short scans. With even larger blocks, a scan for less than 1 000 elements has to scan a large portion of the block unnecessarily before reaching its elements. To achieve a good tradeoff between insertion throughput and scan performance for short and long ranges, we select the configuration with block size $C = 2\,048$ and segment size $S = 256$ as our *scan configuration*. It achieves 81.7 % of the insertion performance relative to the insertion configuration while being a factor of 1.71 and 2.82 times faster for short and long scans, respectively. Table 6.4 gives an overview over the insertion and scan configuration.

Configuration Name	Density		Size	
	$\tau^h_{\max}$	g	Block	Segment
Insertion	0.9	1.8	128	1 024
Scan	0.9	1.8	2 048	256

Table 6.4.: Overview over the two configurations used for the evaluation of the BBPMA.

6.2.2. Evaluation

After tuning the BBPMA in the previous section, we now analyze the performance of the batch-parallel insertion operation more closely, focusing on the insertion configuration. Scan queries as well as the scan configuration will be analyzed in comparison to the competitors in Section 6.4.

Running Times by Phases We analyze the behavior of our batch-parallel insertion algorithm in depth by considering the contribution of the five phases of the insertion algorithm (see Section 4.2.2) to the overall running time. Figures 6.3 and 6.4 give an overview of the contributions for all inputs.

We begin with an analysis of the normal, dense normal, zipf, and uniform inputs which are shown in Figure 6.3. In the sequential case, the running time for executing the insertions clearly has the most influence, requiring at least 80 % of the running time. The remainder of the time is dominated by the rebalancing execution time. The update phase requires a short but significant time, while both insertion and rebalancing distribution phases are not relevant. In the sequential case, the inputs behave very similarly.

In the inputs considered here, insertions are — to a varying degree — distributed relative to the uniform prefill. They require insertions and rebalancing across the entire data structure. This means that insertions are needed for many blocks, but each block likely only receives few elements. These distributed insertions are more expensive than if few blocks require insertions, and each block receives a large portion of the batch. On the other hand, inserting few elements per block makes block splits less likely. Therefore, large rebalancing regions are rare. Additionally, rebalancing is done on the block level, rather than on the element level, so that less work is needed in general. Together, this explains why significantly more time is needed for the insertions than for rebalancing. The update phase is significantly faster than the rebalancing phase in the sequential case, although rebalancing a section requires that it was considered multiple times in the update phase. This shows that the read-only update phase is very efficient in the sequential case.

There is little difference in the relative running times of the phases for the different inputs, but the overall times vary. The more distributed the insertions are, the longer it takes to execute them. This underscores the argument made above: Inserting batches that are widely distributed across the data structure is more costly. While the zipf input requires the least time for the insertion phase, its rebalancing time is relatively the highest. This shows that if the input is more skewed, rebalancing work increases. But even with relatively more rebalancing work, the zipf input performs best across all inputs which shows that the block indirection helps with efficiently handling inputs that are traditionally hard for PMA-based data structures.

We now focus on the running times for more threads. For eight threads, good speedups of 5–6.5 are achieved for the normal, dense normal and uniform inputs. The insertion execution phase behaves best and scales almost perfectly. Here, the distributed insertions are beneficial as the work is easily parallelized and efficiently executed in parallel. The rebalancing time is improved as well, but the speedup compared to the sequential execution is slightly lower. The update phase is faster for two threads than for one but does not show additional speedup for more threads. Opposed to the other phases, it is much harder to parallelize: There is less work to do in the first place, and the work is distributed across the

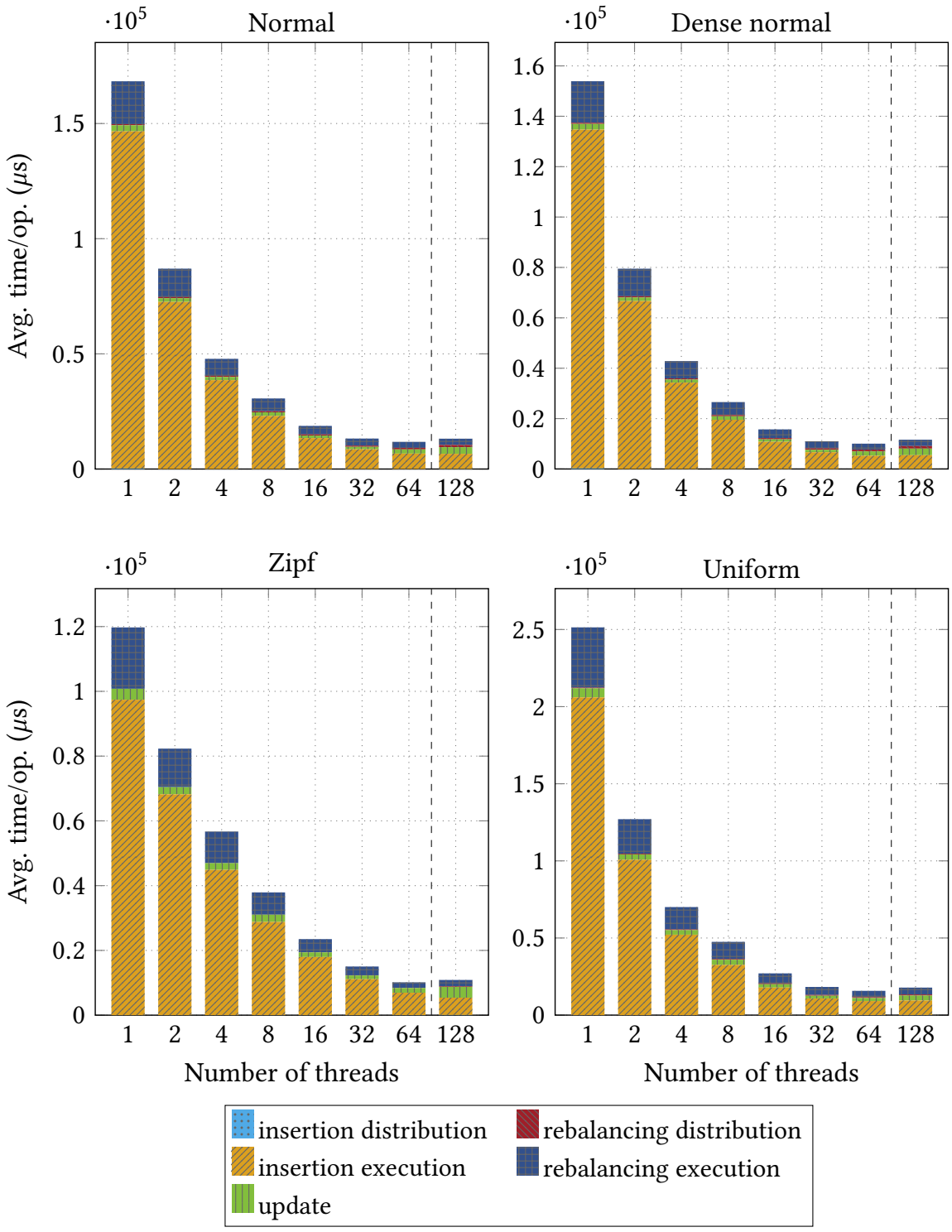


Figure 6.3.: Contribution of the five phases of an insertion to the total running time per batch insertion in the BBPMA with the insertion configuration for the respective input. The experiments use 10^8 elements of prefill and measure 100 batch insertions of 10^6 elements each. The plots show the maximum time a thread spends in the respective region.

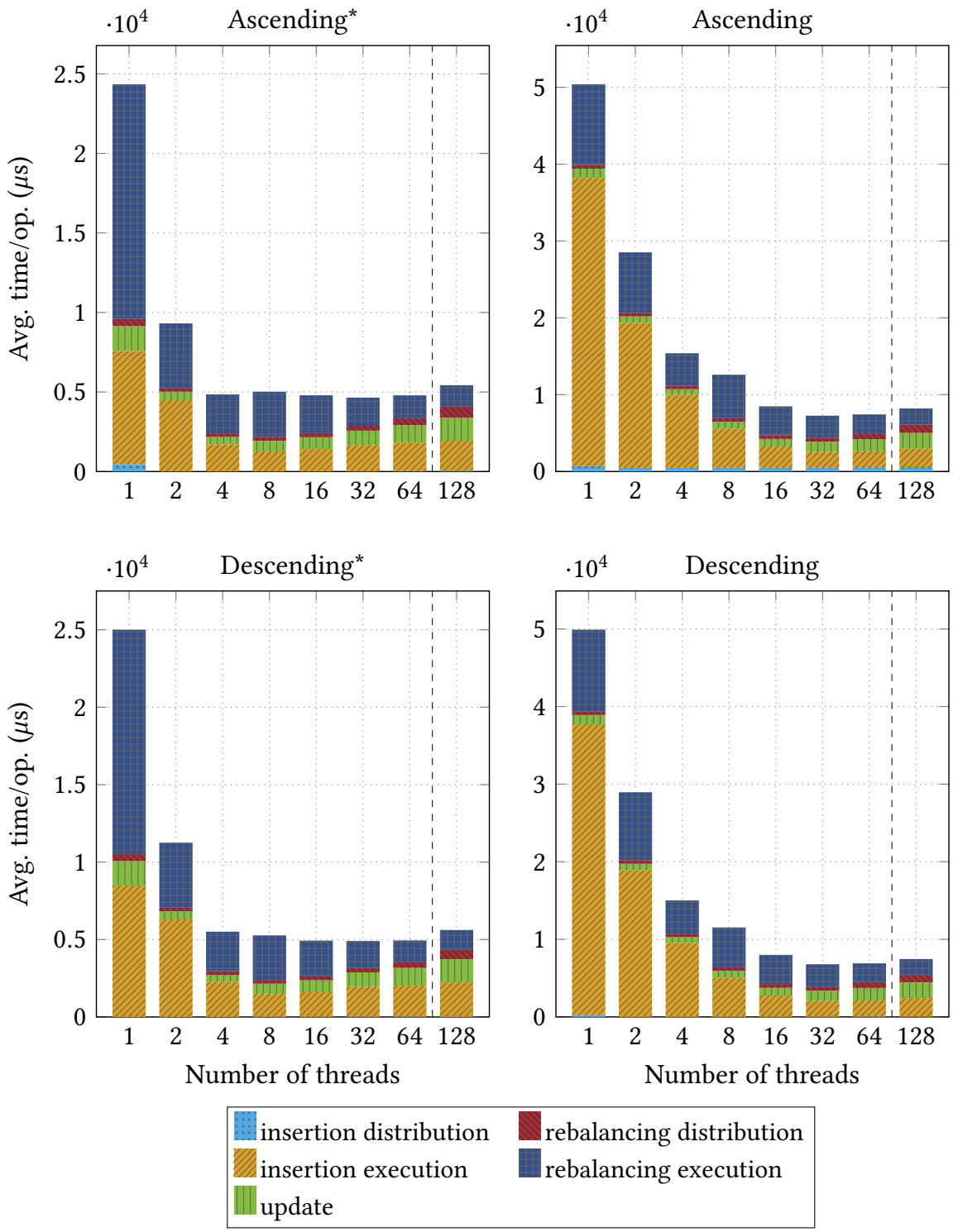


Figure 6.4.: Contribution of the five phases of an insertion to the total running time per batch insertion in the BBPMA with the insertion configuration for the respective input. The experiments use 10^8 elements of prefill and measure 100 batch insertions of 10^6 elements each. The plots show the maximum time a thread spends in the respective region.

levels of the tree, which have to be processed sequentially. The amount of work decreases exponentially with the levels. Additionally, a node on higher levels only requires adding the values for its two children, while a leaf requires scanning the entire segment. More detailed evaluations revealed that while the lowest level can be parallelized efficiently, this is no longer true for the higher levels.

For 64 threads, the insertion execution phase still makes up the largest part of the running time, but the rebalancing phase and the update phase are much more relevant. While the phases with large amounts of distributed work scale well, this is not true for the update phase, which therefore becomes a bottleneck for the parallel insertion throughput. Using more threads than cores makes the insertions slower.

Figure 6.4 shows the results for the sorted inputs. We first consider the ascending and descending inputs, which use uniform prefill. In the sequential case, the insertion execution phase still makes up 80 % of the running time. Relative to the uniform prefill, each batch is inserted into one section of the reference PMA, so that elements are still distributed across a number of blocks. However, the section is much smaller than in the previous cases (by expectation, around 1 % of the reference PMA), which explains why the running times are up to 5 times faster than in Figure 6.3. Relative to the faster overall times, the distribution phases for insertion and rebalancing are now visible, but do not make up a significant share of the sequential running time.

Scalability is good for up to four threads but deteriorates for more threads. Once more, the update phase does not scale well, and the distribution phases have a larger share in the running time, which hinders scalability. The running time of the insertion distribution phase does not change with a higher number of threads. While the work is parallelized trivially between the threads, the work increases linearly with the number of threads, and the running time is dominated by the span. The update phase performs worse for 32 or more threads. Here, the overheads induced by the parallelization outweigh the benefits.

Finally, the ascending* and descending* inputs have the most skewed insertions. Here, the relation between insertion and rebalancing execution phase are inverted, and the insertion phase is much faster than the rebalancing phase in the sequential case. For these inputs, the entire batch is inserted into a single block at one of the ends of the data structure. The batch is efficiently split up into blocks, and only relatively few elements from the existing block have to be merged into it. On the other hand, many new blocks are created at either end of the data structure, which creates a significant imbalance. Rebalancing the block references is comparably expensive in the sequential case. As the behavior is almost identical between the ascending* and descending* inputs, the data structure does not differentiate whether the elements are inserted at the front or end of the data structure. This highlights a drawback of using the rebalancing tree to generate rebalancing regions: Its potential regions are predefined based on the position of the input. In the worst case, the update phase decides to rebalance a large number of blocks from the end of the data structure to the left, where existing blocks must be moved. It can overlook that there might be empty slots to the right, so that rebalancing to the right would be much easier. The sequential time for the update phase remains similar to the previous inputs but has a larger relative share.

In the parallel cases, the insertion phase scales well for up to four threads. With more threads, the running times stagnate. Here, the insertions might be memory-bound, as they

copy a large number of elements with a high degree of parallelization. The update phase only scales well for two threads and becomes slower for a higher number of threads. Here, the update phase has little work on each level, so that it is even harder to parallelize. From one to two threads, the rebalancing execution phase exhibits a superlinear speedup of around 3. First of all, this underscores that our rebalancing distribution algorithm can parallelize even the rebalancing of a single, large rebalancing region. The most likely cause for the superlinear speedup is the additional cache capacity that is provided by additional cores. To an even larger degree than for the more regular inputs, the parallel running times are hindered by the update and distribution execution phases which do not scale as well as the execution phases.

Strong Scaling Evaluation We examine the same evaluation setup again, but now focus on the overall throughput that is achieved for each input. The results are summarized in Figure 6.5. Besides the previous batch size of 10^6, we additionally consider a smaller batch size of 10^5 to investigate the influence of the batch size on the strong scaling behavior.

We first consider the results for batch size 10^6. The results reflect the findings from the analysis of the running time of the individual phases: The more distributed the inputs are, the worse the overall throughput. The throughput clearly distinguishes the inputs into three groups. The speedups are relatively good for up to 32 threads, but no additional speedups are achieved for more threads.

The lowest throughputs are achieved for the inputs dense normal, normal, zipf, and uniform. The data structure can process between 64 million (uniform) and 100 million (dense normal) insertions per second with 64 threads. The relative speedup ranges from 12 (zipf) to 16 (uniform). It is greater for the inputs where the share of the insertion execution time is higher, as the phases that do not scale well are less relevant.

The sorted inputs achieve significantly higher overall throughputs. For 64 threads, the BBPMA processes up to 139 million (descending) and 223 million (descending*) elements per second. It achieves the best results for the inputs that are traditionally the hardest for PMAs. This shows that the block indirection significantly improves the robustness of PMA-based data structures against skewed input and that our batch-parallel rebalancing scheme is effective.

The relations between the inputs are identical for smaller batches of size 10^5. With the smaller batches, the overheads that are necessary to distribute the batch insertion must be amortized over less elements. The smaller batches make it more likely that many blocks receive few elements each. Therefore, the throughputs are smaller by factors of around 2–3, depending on the input. Moreover, the scalability is worse for smaller batches as the phases that do not scale well become more significant. This indicates that large batches are necessary for the efficient parallelization of the BBPMA.

Weak Scaling Evaluation The strong scaling evaluation results indicate that the scalability of the batch-parallel insertion operation depends on the batch size and is limited for a fixed batch size. Figure 6.6 presents the results of a weak scaling experiment, where the batch size scales with the number of threads. Specifically, it measures 100 batch-parallel insertions of batches each containing 50 000 elements per thread. Performing insertions in larger batches improves the throughput even for a fixed number of threads. To isolate the influence of a varying number of threads, we evaluate the *execution times* of batch

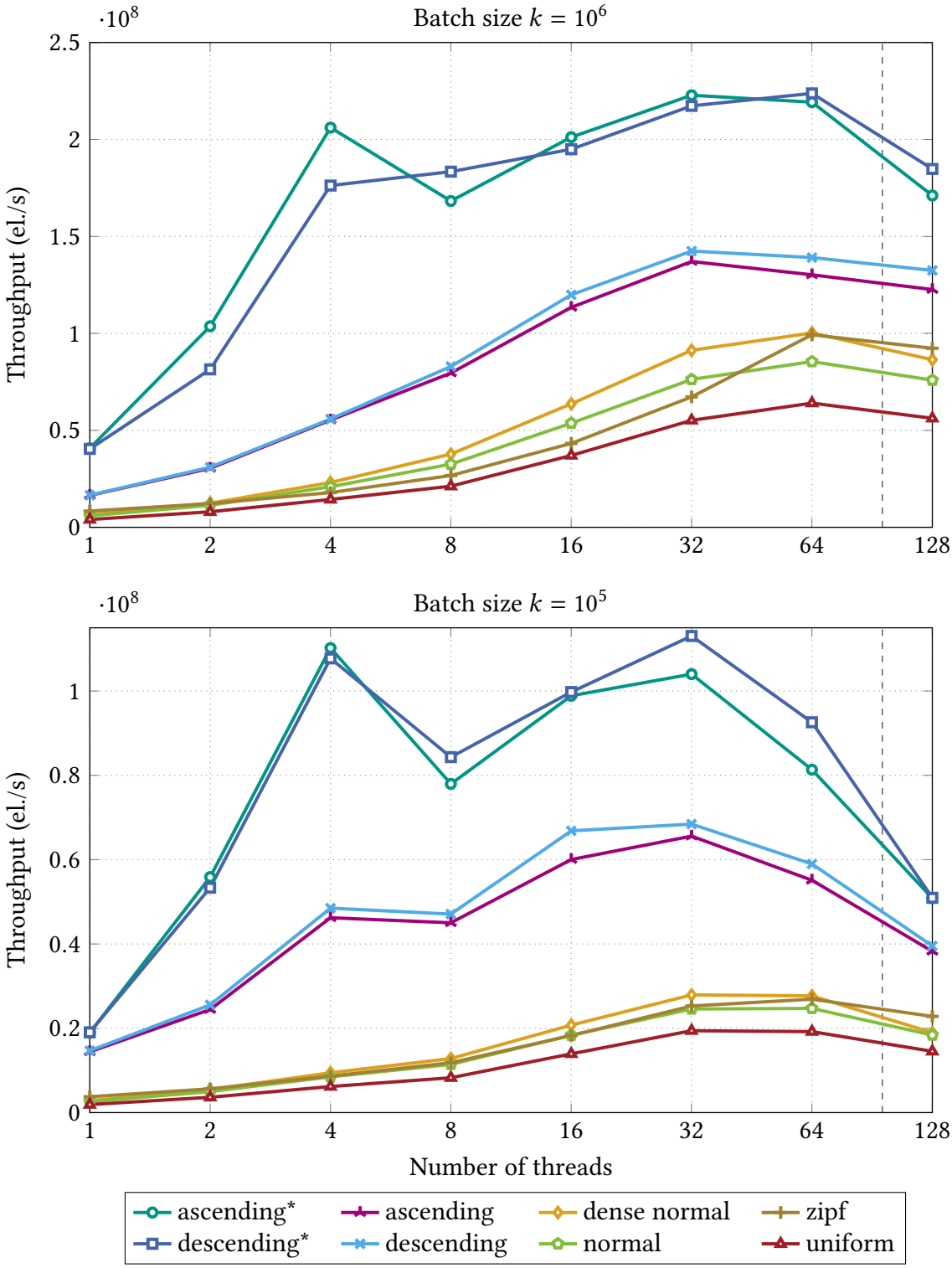


Figure 6.5.: Strong scaling evaluation for the BBPMA with the insertion configuration, different inputs and two batch sizes. All evaluations were run with 10^8 elements of prefill and measured batch insertions for 10^8 elements.

insertions, rather than the throughput. As we insert more elements in the measured insertions, we use $2 \cdot 10^8$ elements of prefill to reduce the influence of growing the data structure for a high number of threads. Optimal scalability would be achieved if running times remain constant across all numbers of threads.

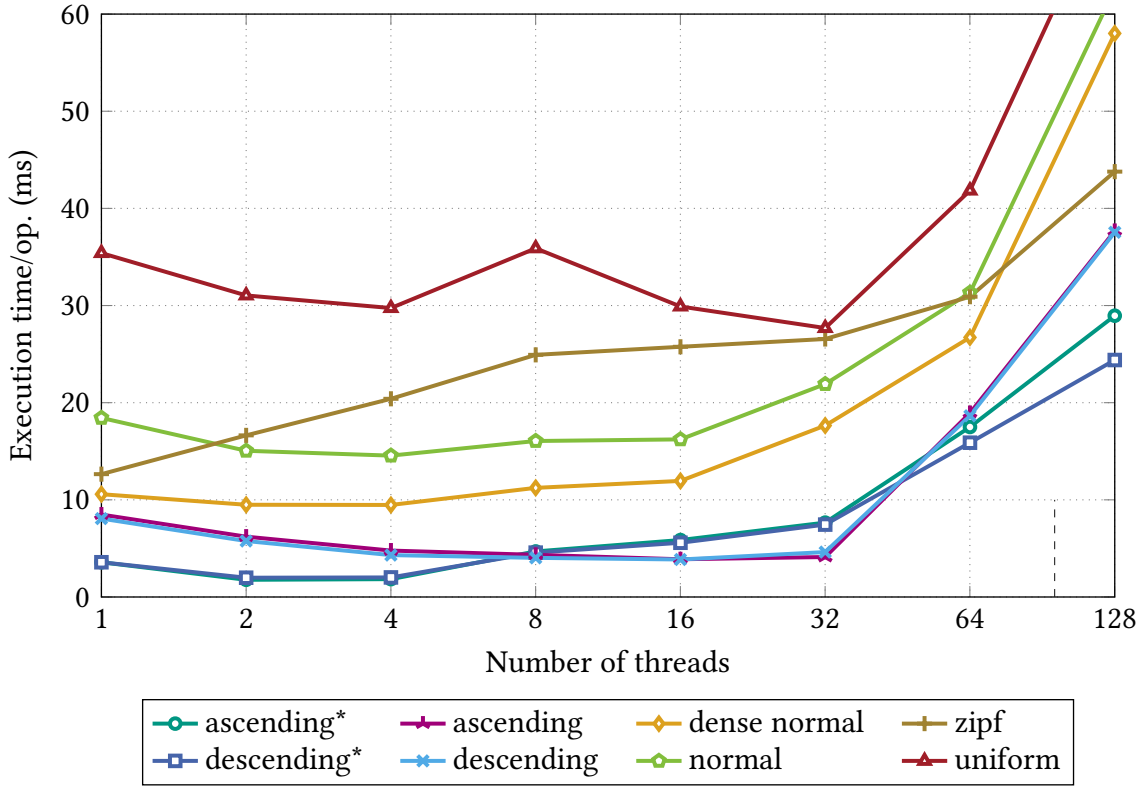


Figure 6.6.: Weak scaling evaluation for the BBPMA with the insertion configuration for all inputs. All evaluations are run with $2 \cdot 10^8$ elements of prefill and measure 100 batch insertions each, where batches contain 50 000 elements per thread.

The scalability depends on the input. For the uniform input, scalability is good for up to 32 threads. For 64 threads, the running time is 1.18 times slower than in the sequential case, which shows that the insertion algorithm does not scale well for this case. The scalability for the normal, dense normal and zipf inputs is slightly worse — they scale well for up to 16 threads but exhibit a slowdown of up to 2.1 for 32 threads, and a factor of up to 2.4 for 64 threads. These inputs require more rebalancing work, so that more overhead is necessary in the distribution. For the ascending, descending, ascending* and descending* inputs, the scalability is very good for up to 32 threads. For 64 threads, they have a slowdown of up to 5 compared to the sequential variant. For all cases, the insertions do not scale well if more threads than cores are used.

The weak scaling evaluation shows that the insertion operation with sufficiently large batches scales optimally for up to 16 threads for all inputs. For 32 threads, the scalability is still optimal for inputs that do not require much rebalancing or insert their elements extremely locally. For the other distributions, the insertion has a slowdown of up to 2. For 64 threads, the insertions do not scale well in any cases. Here, the additional overheads no

longer justify the additional threads, even if batches are large enough. Figures 6.3 and 6.4 show that the update phase becomes significantly slower for 64 threads in comparison to 32 threads. Against this background, it seems likely that the update phase is responsible for the lack of scalability for 64 threads.

Data Structure Size Influence Figure 6.7 studies the influence of the prefill size on the insertion throughput. The previous relations between the inputs are reflected around the prefill of 10^8 elements. For less prefill, the relations remain almost the same. If the prefill is much smaller than the measured workload, there are no more differences between the normal, dense normal and uniform inputs. Here, the structure of the PMA quickly adapts to the new input and performs the insertions almost equally fast, independent of the inputs. The better throughput of the zipf input is caused by duplicate elements, which are counted as insertions but require less work. For the more regular inputs dense normal, normal and uniform, the throughput slowly drops with a larger data structure. If the prefill gets a factor of 10^3 larger, the throughput drops by between 10 % for the dense normal input and around 70 % for the uniform input.

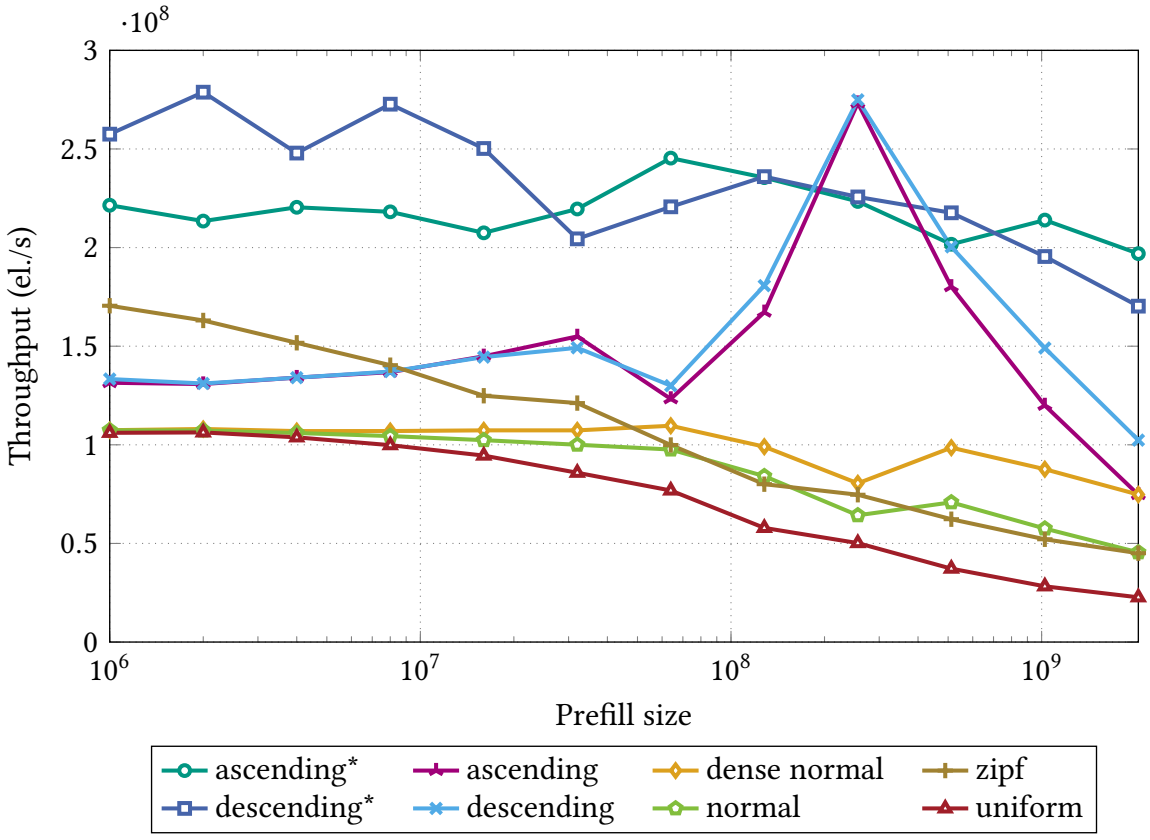


Figure 6.7.: Throughput for the insertion of 10^8 elements in batches of size $k = 10^6$ into the BBPMA with the insertion configuration after the data structure was prefilled with the respective number of elements.

The throughputs for the ascending* and descending* inputs are less prone to degradation due to more prefill. Here, an entire batch is always inserted at the right or left end of the

current data structure. There is always only a single, large rebalancing region which is efficiently rebalanced in parallel. An interesting behavior can be observed by the ascending and descending input. The throughput for both inputs has a distinct peak with a prefill of 256 million elements, while it is lower for both less and more prefill. These inputs insert each individual batch into a small region of the reference PMA which slides across the reference PMA with the batches. The peak likely forms because the prefill of 256 million elements is sufficient so that each batch insertion can insert all its elements into the respective segments without requiring much rebalancing work. While the segments remain with a high density, no future insertions will insert elements into them, so that the rebalancing cost must never be made up. With less prefill, the batch is inserted into a smaller section of the reference PMA, so that more rebalancing is needed. With even more prefill, the range in which a single batch is inserted becomes larger again, so that the insertions are less efficient.

6.3. Batch-Parallel Buffered Packed Memory Array with Global Rebalancing

We briefly evaluate the performance of the GBPMA, the variant of the BBPMA that uses a simplified global rebalancing routine. Analogous to the BBPMA, we first tune important parameters and then evaluate the performance of the data structure.

6.3.1. Parameter Exploration

The GBPMA has two important tuning parameters: The growing factor g that determines the size of the new reference PMA after a global rebalancing operation, and the block size C. For simplicity, we tune the two parameters independently.

Growing Factor Recall that the GBPMA aims to be efficient by using a sufficiently large growing factor so that the expensive out-of-place insertions happen rarely. At the same time, a higher growing factor impairs the efficiency of other operations as it decreases the density in the reference PMA. Figure 6.8 shows the influence of the growing factor on the frequency of out-of-place insertions with block size $C = 128$. This evaluation does not prefill the data structure, so that the behavior can be analyzed even for situations in which the data structure grows.

To analyze the results, we must distinguish between the inputs. The uniform input is clearly the most suitable for the GBPMA. An out-of-place insertion adds a factor of g of additional space which subsequent operations can use. As the batch elements are distributed uniformly, all PEs are expected to receive a region of similar length and a similar number of batch elements. In this way, many subsequent operations can fill up the additional space. If an insertion requires out-of-place execution again, all regions of the PE are almost full, so that the additional space can be used effectively. Therefore, even a small growing factor of 1.1 suffices for less than 10 % of out-of-place insertions. With a growing factor of 1.8, only 1 % of the insertions has to be executed out-of-place.

This effect is also visible — but less strong — for the more focused inputs normal, dense normal and zipf. Here, it is increasingly more likely that a single PE is unable to insert all its elements in-place. In this case, it triggers a global out-of-place execution, although most threads would still have sufficient space for more in-place insertions. Therefore, higher growing factors are needed, and even with a growing factor of 1.8, it is still not possible to achieve less than 9 % of out-of-place executions.

Finally, the sorted inputs take this development to the extreme: They insert all of their batches into very focused regions of the reference PMA so that it is almost never possible to use an in-place insertion with the growing factors evaluated here, except for the ascending input and a high growing factor. The growing factor adds space globally, but it is only required locally. Therefore, each local insertion triggers expensive, global rebalancing. With low growing factors, the data structure is not expected to perform well for such inputs.

To tune our data structure, we focus on the input that it is expected to perform well on and use a growing factor of $g := 1.8$. It uses little space but still consistently ensures few out-of-place insertions for the four relevant inputs.

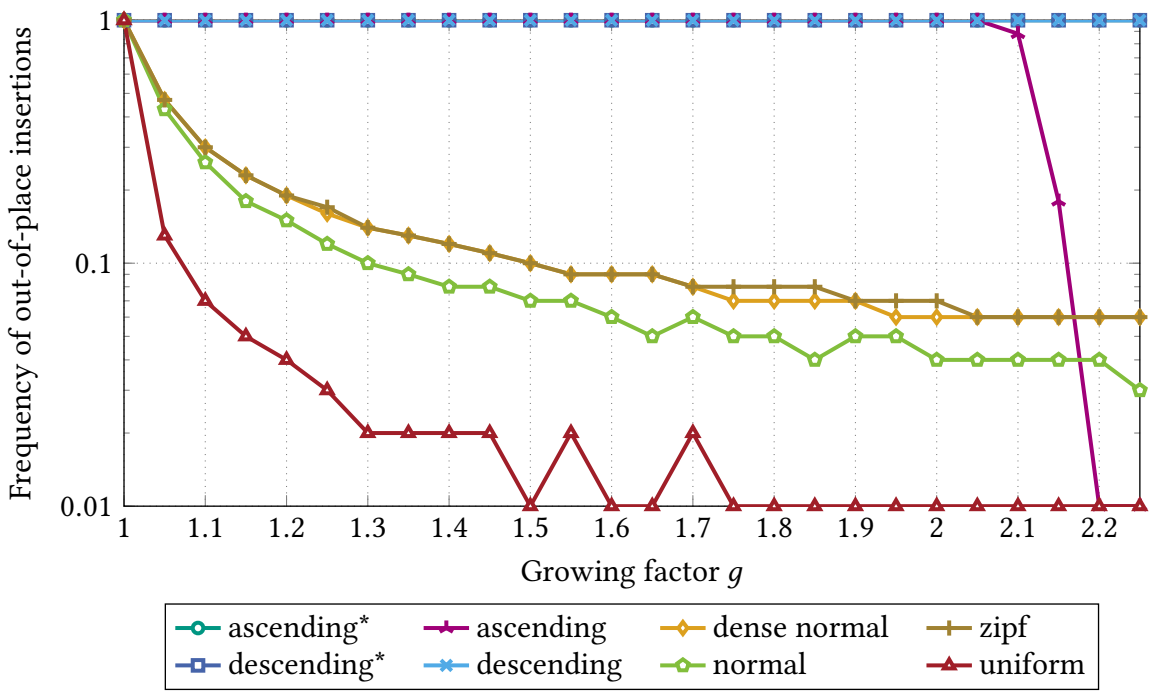


Figure 6.8.: Frequency of out-of-place insertions when using the given growing factor in the GBPMA. The evaluation uses no prefill and performs 100 measured insertions of batch size 10^6 and block size $C = 128$.

Block Size The influence of the block size on the insertion throughput is evaluated in Figure 6.9. As we use this evaluation to decide for a block size, we need to be careful which results we consider: The previous evaluation indicated that the data structure likely will not perform well for the sorted inputs — which can be expected due to the simplified global rebalancing. Therefore, Figure 6.9 only considers the other inputs. Results for all inputs can be found in Figure A.1. In fact, the sorted inputs perform better for larger blocks, but this is likely because larger blocks can buffer more insertions before a global rebalancing operation is required.

For the remaining inputs, it can be seen that block sizes 32, 64, and 128 perform the best for most inputs. Block size 128 seems to be ideal as it is only slightly slower than the other candidates for the uniform, normal and dense normal inputs while it is significantly better for the zipf input. Therefore, we use the GBPMA with a growing factor of 1.9 and block size of $C = 128$.

6.3.2. Evaluation

After the parameter tuning, we conclude the evaluation of the GBPMA with a brief study of the strong scaling behavior of insertions. Figure 6.10 shows the results for all inputs. It must be noted that — opposed to the parameter tuning experiment for the growing factor — we prefill the data structure with the same number of elements that will ultimately be inserted. Therefore, we expect the influence of out-of-place insertions to be lower than indicated by Figure 6.8.

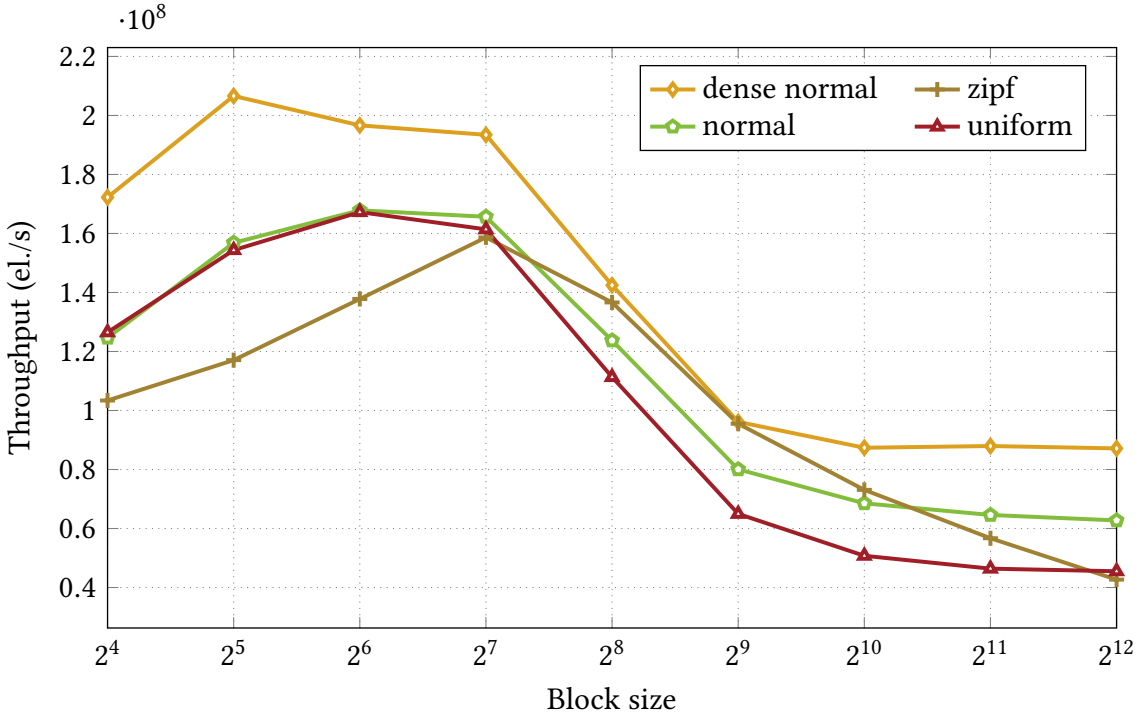


Figure 6.9.: Insertion throughput in the GBPMA for different block sizes, using a growing factor of 1.8 and 64 threads. Results for all inputs can be found in Figure A.1.

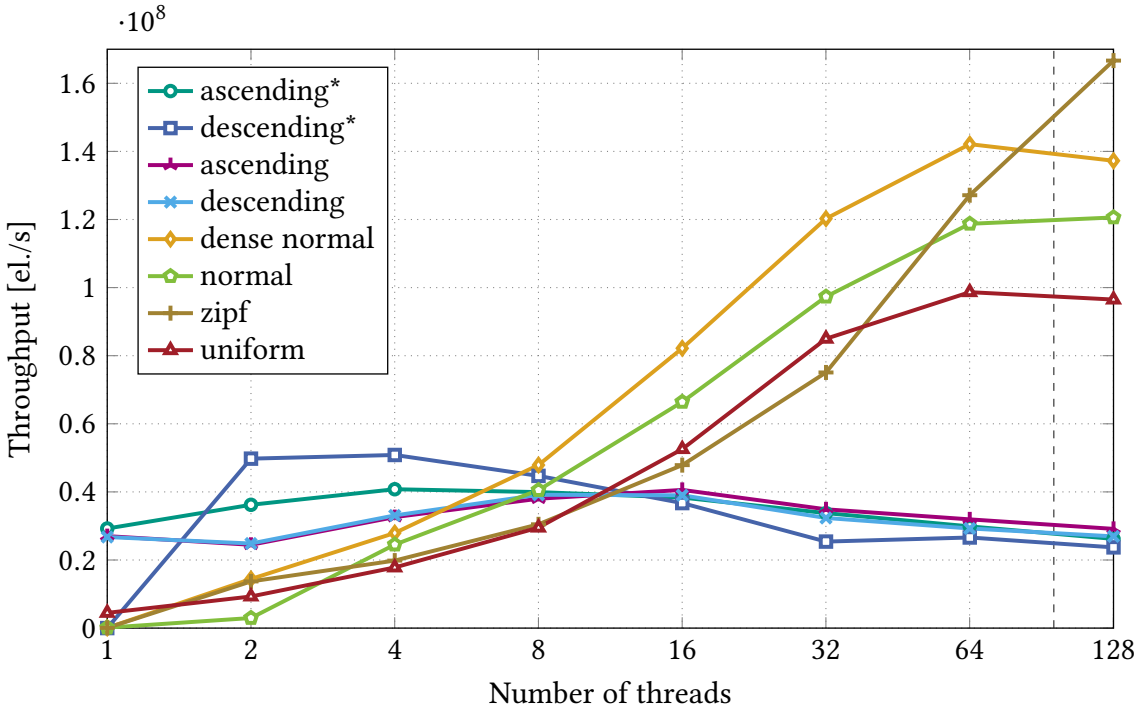


Figure 6.10.: Strong scaling evaluation for the GBPMA and all inputs. All evaluations are run with 10^8 elements of prefill and measure batch insertions for 10^8 elements, in batches of size 10^6.

As anticipated, the GBPMA does not scale well for the sorted inputs, where no speedup is achieved by the parallel variants. Surprisingly, the sequential throughput is considerably better than for other inputs, except for the descending* input. Here, there is only a single thread with one global region, so that it is more likely that there is enough space for an in-place insertion. The in-place insertions for the skewed inputs are fast, similar to previous evaluations. However, the good sequential performance cannot be transferred to the parallel case. Here, the regions of the threads become smaller and it is more likely that a thread can no longer insert in-place. The global out-of-place rebalancing operations become more frequent, and the parallelization does not ensure good work balance.

On the other hand, the more regular inputs normal, dense normal, zipf, and uniform scale well. For the uniform input, relative speedups of 19 and 22 are achieved for 32 and 64 threads, respectively. The growing factor enables them to perform a large part of the insertions in-place. No overhead is needed for generating, distributing, and processing any rebalancing regions. As the out-of-place insertions occur very rarely, they do not affect the overall throughput. Similar to the BBPMA, the uniform input performs worst. Again, it inserts elements into blocks in all regions of the PMA which is particularly expensive. The normal and dense normal inputs achieve higher throughput. They benefit from inserting into less blocks, which is more efficient. The very good throughput for the relatively skewed inputs indicates that the global rebalancing scheme seems to work well in this evaluation. Figure 6.8 shows that few out-of-place insertions are needed with the selected growing factor. This indicates that the GBPMA is particularly well suited for workloads that have uniformly distributed insertions and keep the data structure size stable over time.

6.4. Comparison of the Data Structures

In this section, we compare the performance of our new data structures with each other and with the two competitors. We evaluate the performance for insertion and scan queries.

Data Structure Selection For our contribution, we consider the BBPMA in its two configurations (described as *BBPMA (insert)* and *BBPMA (scan)*, see Table 6.4 for details) as well as the GBPMA. With the competitor data structures, we aim to contextualize the performance of our new data structures in the tradeoff between insertion and scan performance. Traditionally, PMA-based data structures are more efficient to scan, but less efficient to insert than pointer-based search trees. One of the main research questions of this thesis is how the block indirection affects the positioning along this tradeoff. The theoretical analysis of the (B)BPMA shows that its asymptotical performance for insertions is equal to that of (parallel) search trees, ignoring lower order terms for the BBPMA. The comparison with the PST allows us to analyze how insertions behave practically and how much advantage the BPMA-based approach yields for scan queries. While the block indirection allows us to reach the theoretical efficiency of search trees for insertions, it comes at a cost for scans, as the elements are no longer allocated in contiguous memory, but in blocks. The comparison with the uCPMA allows us to analyze the influence of this design on the scan performance. Additionally, it provides a baseline for the insertion performance that can be reached with a traditional PMA-based approach. While our data structures and the uCPMA ignore duplicate elements, the PST can contain multiple elements with the same key. This might cause some differences in behavior for the zipf input, which generates the most duplicates.

Strong Scaling for Insertions Figures 6.11 and 6.12 show the strong scaling results of all five data structures on all inputs. As we already analyzed the results for our data structures above, we focus on an analysis of the performance of the competitors and how our new data structures compare to them

The behavior for the more regular inputs can be found in Figure 6.11. Except for the zipf input, the uCPMA consistently performs the worst. Its insertions are a factor of 2.5–3 slower than those of the Parallel Search Tree. The scalability is acceptable for up to 64 threads. Meanwhile, the PST achieves significantly higher overall throughput and scales well even for 128 threads. The comparison confirms the traditional weakness of PMAs compared to search trees for insertions.

Even with the scan configuration, the performance of our BBPMA is better than that of uCPMA for all inputs except the zipf input for every number of threads. For 64 threads, it is 20 % slower than the uCPMA for the zipf input, but up to 2.25 times faster for the other inputs. For 128 threads, it is 40 % faster than the uCPMA even on the zipf input. Compared to the PST, the BBPMA with the scan configuration is a factor of less than 2 slower. With the insertion configuration, the BBPMA is faster than the PST for up to 32 threads on all inputs except the uniform input. For a higher number of threads, the scalability is once more limited so that PSTs have an advantage. For 64 threads, the BBPMA with the insertion configuration is around 34 %, 14 % and 8 % slower than the PST for the uniform, dense normal and normal inputs. It is 17 % faster for the zipf input, but this might be partially due to the different treatment of duplicates. The BBPMA with the insertion

configuration is faster than the uCPMA by a factor of 2 for all inputs except the uniform input where it is around 20 % faster. Finally, the GBPMA performs best overall on the regular inputs. It is consistently faster even compared to the PST for up to 64 threads.

The sorted inputs in Figure 6.12 are particularly hard for traditional PMAs as the insertions require lots of rebalancing. We start the analysis with the ascending* and descending* inputs. Here, the uCPMA performs poorly. It achieves only a very low throughput and does not scale. In the ascending* case, the throughput is lower than the simple GBPMA (which does not scale either). In both cases, the throughput for the uCPMA is two orders of magnitude below that of the PST. This shows that the uCPMA suffers from the same problems for hard inputs as traditional PMAs with single-element insertions. Meanwhile, the PST exhibits very good parallel performance. No rebalancing is needed, and large local insertions are fast due to pointer manipulations in the search tree.

The performance varies significantly between our data structures. As seen before, the GBPMA does not scale at all. It is not suited for workloads that do not have regular input patterns. The BBPMA with the insertion configuration beats the PST for up to 32 threads. For 64 threads it does not scale as well as the PST, so that it is a factor of around 2.5 slower than the PST. Interestingly, the most efficient data structure overall is the BBPMA with the scan configuration. It beats all competitors by a factor of up to 5 for a number of threads between four and 64. This might be an artifact of its configuration in combination with the batch size: It uses a block size of 2 048 and a segment size of 256, so that more than 500 000 elements can be stored in a single segment. When inserting a batch of 10^6 elements at the left or right end of the data structure, the required rebalancing work is usually small and occasional larger rebalancing regions are efficiently executed in parallel.

For the ascending and descending input, the uCPMA scales better, but its performance is only in the range of that of the simpler GBPMA. Again, it is about a factor of 10 slower than the PST. The PST achieves the best throughputs and exhibits the best scalability in comparison to all other data structures, including our new data structures.

The BBPMA with the insertion configuration achieves the same throughputs as the PST for up to 16 threads. Afterwards, the additional speedup for the BBPMA is small and the PST clearly performs better. For these hard inputs, even the intricate rebalancing scheme of the BBPMA seems to come to a limit. The pointer-based operations on the PST offer parallel throughput that is a factor of almost 3 higher. Compared to the uCPMA, the BBPMA with the insertion configuration is around 2.5 times faster for 64 threads. While the scan configuration of the BBPMA performs a factor of 2 worse than the insertion configuration for low number of threads, it achieves even slightly better performance for high parallelization. Here, the update phase is likely less relevant as the reference PMA is much smaller for the larger blocks. Once more, the scan configuration achieves better performance than the uCPMA for more than 8 threads. The GBPMA does not perform well and is around one order of magnitude worse than the PST.

In conclusion, the BBPMA with the insertion configuration is consistently faster than the uCPMA on all inputs and for each number of threads. In the parallel case, it is a factor of 1.2–2.5 faster on inputs where the uCPMA scales well, and a factor of around 25 faster on inputs where uCPMA does not scale well. Compared to the PST, it is up to 34 % slower for the regular inputs, and a factor of 2.5–3 slower for very skewed inputs. The BBPMA with the scan configuration is usually slower to insert than the insertion variant, but faster

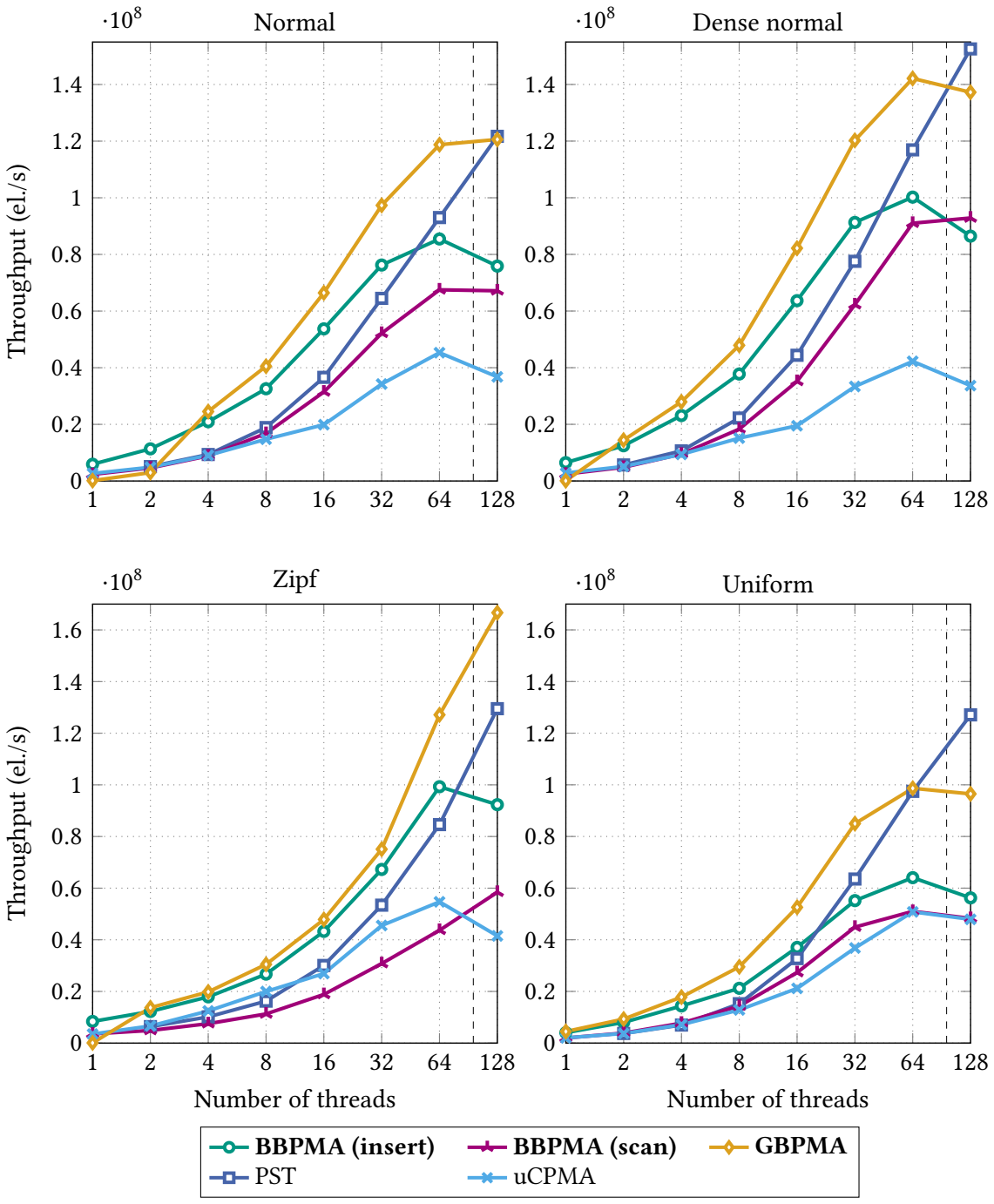


Figure 6.11.: Strong scaling comparison between the data structures for different inputs. All evaluations were run with batch size 10^6 and 10^8 elements of prefill and measured insertions, each.

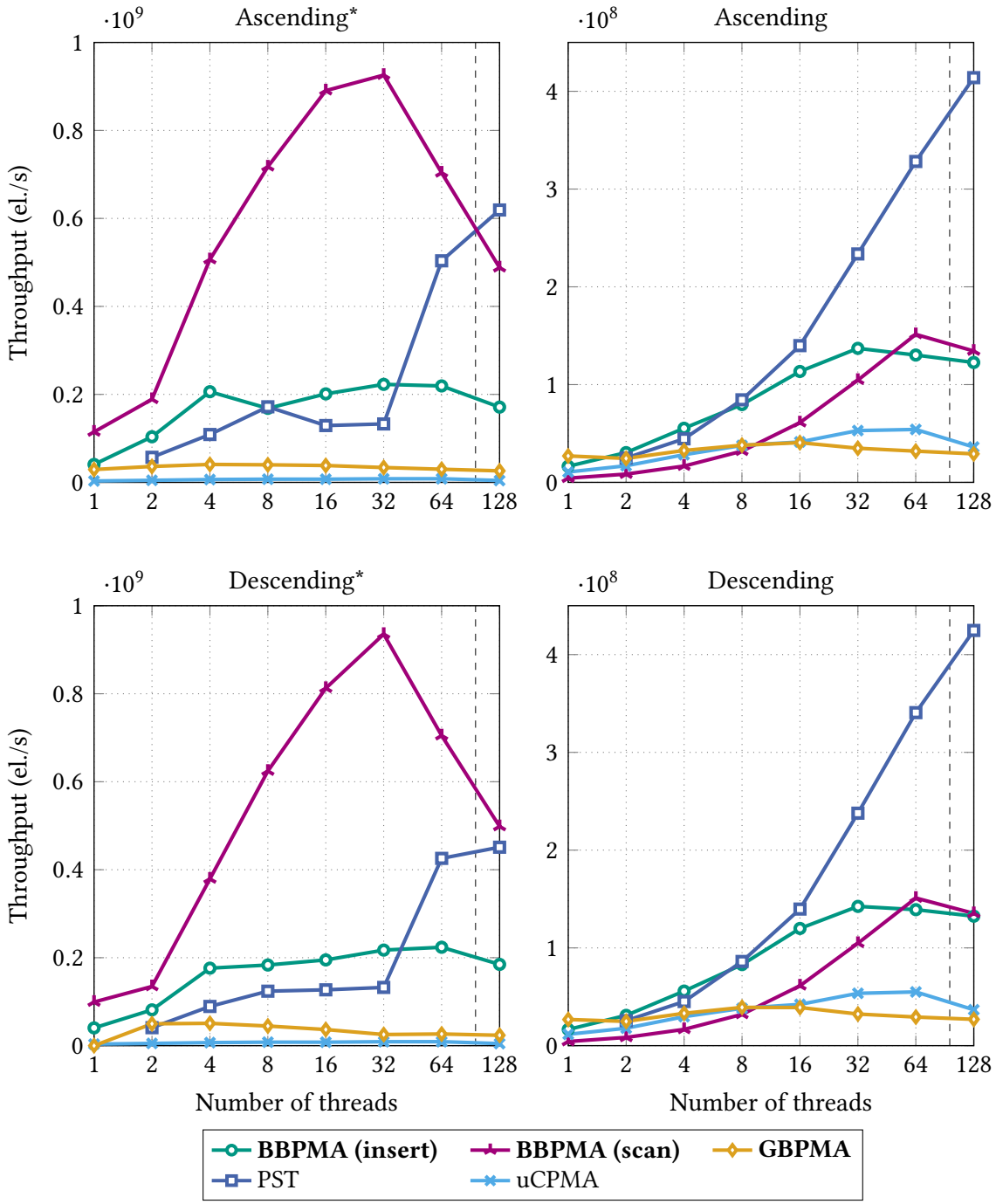


Figure 6.12.: Strong scaling comparison between the data structures for different inputs. All evaluations were run with batch size 10^6 and 10^8 elements of prefill and measured insertions, each.

than the uCPMA for almost all cases. Both variants are far more robust against skewed inputs than the uCPMA. The GBPMA offers very good throughputs for the regular inputs, even beating the PST consistently. However, it does not perform well on skewed inputs, so that its performance on mixed real-world workloads cannot be guaranteed.

Different Batch Sizes So far, we only compared the data structures for batch size 10^6. Figure 6.13 shows the insertion throughputs for 64 threads and batch sizes from 100 to 10^7 elements. We consider the uniform input, where our data structure performs the worst out of the regular inputs compared to the competitors. For all batch sizes, we use 10^8 elements of prefill and then insert 10^8 elements during the measurements to ensure a fair comparison.

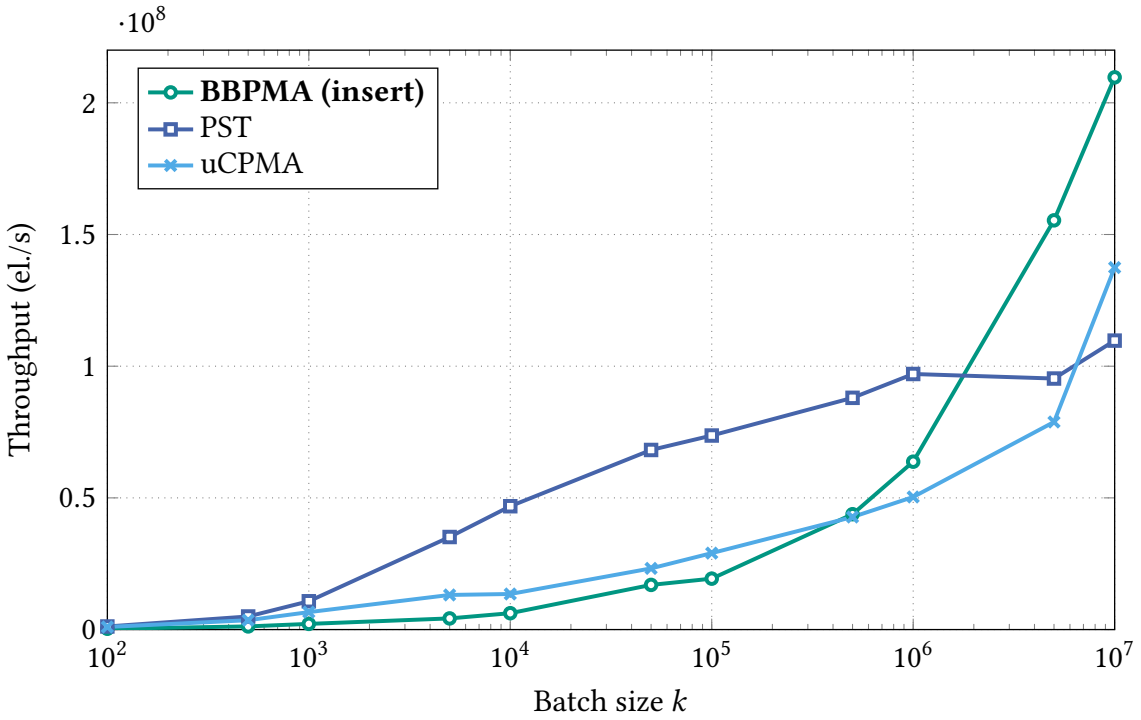


Figure 6.13.: Insertion throughput for different batch sizes. Each data structure starts by inserting 10^8 elements of prefill, before measuring the insertions of 10^8 elements in batches of the given size. All evaluations use the uniform input and $p = 64$.

The PST performs best for all batch sizes up to 10^6, beating the BBPMA and the uCPMA by a factor of 2–5. Up to a batch size of 50 000, the uCPMA is faster than the BBPMA by a factor of 3–7. The comparison to the PST shows that both PMA-based data structures perform significantly worse for small batches. They need to rebalance the data structure for each batch insertion. Even if less of the data structures needs to be rebalanced for smaller batches, substantial time is required to find out where rebalancing is necessary. The BBPMA appears to be affected more significantly than the uCPMA for very small batches. In this case, it is unlikely that multiple elements are inserted into a block, so that the cost of updating a block can be amortized over less insertions. As we consider the

throughput for 64 threads, the BBPMA additionally suffers from worse scalability for small batches.

BBPMAs are already faster than uCPMAs for batches that contain at least 500 000 elements. Here, the block indirection is useful again. With batches of at least 5 million elements, BBPMAs even beat the throughput achieved by the PST. In conclusion, the BBPMA is particularly suited for large batch sizes. Here, it benefits most from efficient insertions due to the block indirection and scales the best.

Scan Queries The main benefit of traditional PMA-based data structures is that they enable faster scan queries than search trees, as the elements are allocated in contiguous memory. We compare the scan performance of all five data structure variants. As the original implementation of the PST does not provide a scan query, we implement it ourselves. Within each leaf, the scan query simply iterates the elements. It navigates between the leaves of the search tree using a linked list between the leaves. As the PST manipulates the search tree without maintaining such a linked list, we artificially establish it after the insertions are completed. We do not consider the time that is required for building the linked list in the scan query evaluation, which gives the PST a slight advantage.

Figure 6.14 shows the throughput of scanned elements that is achieved during scan queries. We distinguish scan queries into seven buckets depending on the number of elements that are scanned. Between the two competitor data structures, the uCPMA clearly offers faster scan queries across all range lengths, conforming with the expectations. Scan queries in the uCPMA have more variation than in the PST, as they depend strongly on the density of elements in the region that is scanned.

For our data structures, we first focus on the BBPMA with the insertion configuration. For very short scans of up to 100 elements, it is around a factor of 2 faster than the uCPMA. Here, a scan likely only considers one or two blocks, where elements are stored densely. For the short scans, the BBPMA also benefits from its faster search queries. For search queries of lengths between 101 and 1 000 elements, the scan performance of the BBPMA is still around 30 % faster than that of the uCPMA. For longer scans, the BBPMA is slower than the uCPMA by up to 35 %. Here, every query considers elements in many blocks and the additional search times for the next blocks lower the overall throughput. Throughout all range lengths, the BBPMA is faster than the PST by a factor of at least 2. The GBPMA performs similarly to the BBPMA with the insertion configuration. The slight differences are likely caused by different densities in the reference PMA, as both data structures use the same block size.

As expected, scan queries perform even better in the BBPMA with the scan configuration. Here, up to 2 048 elements are stored in one block, so that they can be scanned quickly and less new blocks need to be searched. For scans of up to 1 000 elements, it achieves a speedup of around 2 compared to the uCPMA. For longer scans, the speedup is around a factor of 1.25–1.6. For ranges of more than 10 000 elements, the running times of individual queries on the BBPMA are much more consistent than those of the uCPMA, as scans in the PMA have less influence. The BBPMA with the scan configuration is a factor of about 3.5–6 faster than the PST.

In total, we evaluated two variants of the BBPMA. The variant with the insertion configuration comes close to the Parallel Search Tree in insertions and offers significantly

faster scans. With the scan configuration, insertions are slower than those of the PST, but still comparable to the uCPMA. In turn, the scan queries are a factor of 1.25–2 faster than in the uCPMA and up to a factor of 6 faster than in the PST.

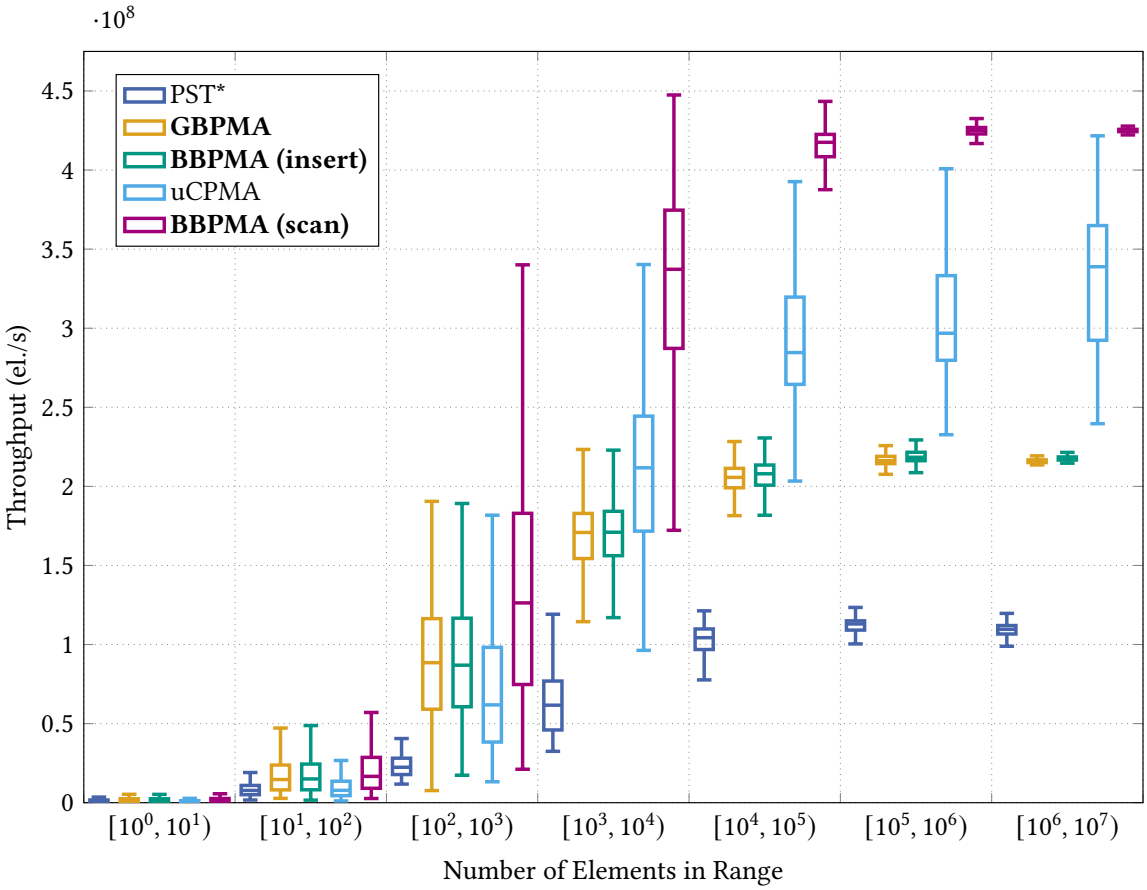


Figure 6.14.: Range query evaluation for different data structures. The evaluation prefills the data structures with 10^8 elements with keys from the uniform distribution and runs 10 000 random scan queries on each data structure. The PST requires an additional postprocessing step to enable fast scan queries. We do not measure the running time required for the postprocessing step, so that the evaluation gives the PST an advantage.

7. Conclusion

To conclude this work, we summarize our contributions and evaluate our results in comparison to the state of the art. We finish with an outlook on future work.

Our Contribution The guiding theme of this work is the introduction of the block indirection in data structures that are based on Packed Memory Arrays (PMAs). Due to their cache efficient scan queries, PMAs are an interesting data structure for use cases that frequently scan sections of the data. However, their insertions are slower than in pointer-based data structures because of their expensive rebalancing routines, so that they can only be used if significantly lower insertion performance can be tolerated. With the block indirection, our new data structures mitigate this effect as they offer significantly faster insertions with only a slight decrease in scan performance for long-range scans compared to traditional PMAs.

Our first contribution is the Buffered Packed Memory Array (BPMA), a new data structure that integrates the block indirection into a traditional PMA for sequential, single-element update queries. The block indirection enables more efficient rebalancing: We prove that our data structure achieves an amortized update cost of $O(\log N)$. In this way, our data structure achieves the same asymptotic update time as pointer-based data structures, bridging the gap between traditional PMAs and search trees.

The second contribution is the Batch-Parallel Buffered Packed Memory Array (BBPMA), an extension of the BPMA that offers batch-parallel update operations. We prove that it transfers the theoretical performance guarantees of the BPMA to the batch-parallel case by enabling batch-parallel insertions of k elements in amortized expected span $O(\log N + \log k)$. Analogous to the sequential case, it improves the update performance by a logarithmic factor compared to the best result for a traditional batch-parallel PMA [26]. At the same time, its performance is close to the theoretical optimal running time that can be achieved in a pointer-based data structure [2]. The only source of expected running times is an internal hash map.

We present an implementation of the BBPMA with insertion, search, and scan queries and extensively evaluate the practical performance of our data structure. Additionally, we compare it to the performance of two competitors: A traditional PMA with batch-parallel updates, but without the block indirection, as well as the Parallel Search Tree, which uses a pointer-based search tree. The comparisons allow us to contextualize the performance of the BBPMA within the state of the art and evaluate the practical influence of our theoretical improvements. When focusing on insertion performance, the BBPMA is 1.2–2.5 times faster on realistic inputs and up to 25 times faster on worst-case inputs than the traditional PMA. At the same time, it reaches up to 30 % better scan performance for short scans, while long scans are up to 35 % slower than in the traditional PMA. Compared to the Parallel Search Tree, insertions are up to 34 % (realistic inputs) or a factor of 2.5–3 (worst-case

inputs) slower, but scan queries are at least 2 times as fast over all scan lengths. The evaluation underscores the results of the theoretical analysis: The insertion performance comes close to that of search trees, while it clearly dominates the insertion performance of traditional PMAs. Particularly, the BBPMA is significantly more robust against worst-case inputs than the traditional PMA. While we maintain the known benefit of batch insertions for traditional PMAs [26], our block indirection and the efficient parallel rebalancing further enhance the robustness and ensure scalability.

No traditional PMA data structure is known that achieves the same update cost as our data structures for all inputs. Theoretical results from the context of online list labeling give a lower bound of $\Theta(\log^2 N)$ work for deterministic rebalancing in a traditional PMA [14, 17]. With the block indirection, we circumvent this lower bound by weakening the property of traditional PMAs that all elements are allocated in contiguous memory. In our data structures, the PMA stores references to blocks, rather than individual elements. While it asymptotically improves update times, this new data structure layout impairs the cache efficiency of scan queries: As elements are allocated in individual blocks, it can be necessary to switch blocks during a scan query. However, pointer-chasing can still be avoided in our data structure: References to subsequent blocks can be found efficiently directly in the reference PMA without having to follow a sequence of pointers first. In this way, our layout enables efficient prefetching of successive blocks, which improves cache efficiency during scan queries. The practical evaluation shows that even with a focus on insertion performance, scan queries in the BBPMA still perform significantly better than in a pointer-based data structure and are only slightly worse than in a traditional PMA. Traditional PMAs are the go-to solution for use cases that require optimal scan performance while tolerating slightly worse insertion performance. With a configuration that focuses on scan performance, the BBPMA offers scan queries that are a factor of 1.25–2 faster than traditional PMAs, while reaching the same insertion performance. Therefore, our BBPMA dominates traditional PMAs for their main use cases.

While the block indirection significantly reduces the amount of work that is necessary for rebalancing, our data structures still use the same full-fledged rebalancing scheme of traditional PMAs. We investigate whether the complexity can be reduced using a simplified variant of the BBPMA that avoids intricate rebalancing operations for batch-parallel updates. By avoiding overheads for rebalancing, it enables significantly improved insertion performance for inputs with regular distributions. However, it does not offer any performance guarantees, and the performance degrades rapidly for skewed workloads. The result justifies our approach of combining the block indirection with rebalancing of traditional PMAs, as our data structure offers performance guarantees and significantly more robust practical performance.

Future Work The evaluation results revealed that the main weakness of the BBPMA is its low scalability for a large number of threads. As our analysis showed, this is mainly caused by the update phase. Its running times are insignificant in the sequential case. However, the speedup for the update phase is very low, so that it becomes a bottleneck in the parallel case and limits the overall scalability. In the data structure description, we already present a variant of the update phase that is asymptotically more efficient than the variant of our implementation. It needs to be evaluated how it performs in practice.

While its asymptotic behavior is promising, it sequentially processes all levels twice. In practice, this could increase the span and therefore offer no improvement. In this case, more engineering work might be necessary on the current update algorithm. Specifically, it could be faster to process nodes on a level sequentially in some cases. Furthermore, the update map that maintains the nodes that need to be processed has a great influence on the update phase. It needs to offer both low contention and good work balance in the parallel processing of the levels. The evaluation also revealed that the BBPMA scales better with larger blocks, but smaller segments, so the parameter tuning is an important factor for the scalability.

While the BPMA reaches the same asymptotical insertion time as traditional search trees, the BBPMA is asymptotically slightly slower than the Parallel Search Tree. It is an interesting theoretical question whether the batch-parallel insertion algorithm can be improved to reach the same asymptotical behavior. Another valuable contribution would be a theoretical analysis of batch-parallel deletions, for which we only outlined an algorithm. The practical evaluation showed varying behavior for the different inputs. A better theoretical understanding of the influence of the input distribution on the performance could be the basis for more informed tuning of our data structure.

Finally, we did not evaluate the performance of the BPMA, our sequential data structure, in practice. As it offers good theoretical performance guarantees, an evaluation of its practical performance would be a key factor in the understanding of the block indirection in PMAs. The good practical performance of the BBPMA appears to be promising for the BPMA. Importantly, we showed that for the batch-parallel case, the block indirection further improves the insertion performance for skewed inputs, as the blocks buffer insertions. Performance degradation for skewed inputs is more problematic for sequential, single-element insertions than in the batch case. Therefore, the BPMA could have an important practical advantage over traditional PMAs. An implementation of the BPMA would be significantly simpler for the BBPMA as it does not require parallelization, work balancing or the explicit rebalancing tree.

Bibliography

[1] Alfred V Aho and John E Hopcroft. "The design and analysis of computer algorithms". In: (1974). URL: http://www.cs.cmu.edu/afs/cs/academic/class/15451-s06/www/lectures/radix.pdf.

[2] Yaroslav Akhremtsev and Peter Sanders. "Fast Parallel Operations on Search Trees". In: *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*. 2016, pp. 291–300. DOI: 10.1109/HiPC.2016.042.

[3] R. Bayer and E. McCreight. "Organization and Maintenance of Large Ordered Indices". In: *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*. SIGFIDET 1970. Houston, Texas: Association for Computing Machinery, 1970, pp. 107–141. ISBN: 9781450379410. DOI: 10.1145/1734663.1734671. URL: https://doi.org/10.1145/1734663.1734671.

[4] M.A. Bender, E.D. Demaine, and M. Farach-Colton. "Cache-oblivious B-trees". In: *Proceedings 41st Annual Symposium on Foundations of Computer Science*. 2000, pp. 399–409. DOI: 10.1109/SFCS.2000.892128.

[5] Michael A Bender, Ziyang Duan, John Iacono, and Jing Wu. "A locality-preserving cache-oblivious dynamic dictionary". In: *Journal of Algorithms* 53.2 (2004), pp. 115–136. ISSN: 0196-6774. DOI: https://doi.org/10.1016/j.jalgor.2004.04.014. URL: https://www.sciencedirect.com/science/article/pii/S0196677404000707.

[6] Michael A Bender, Martin Farach-Colton, and Bradley C Kuszmaul. "Cache-oblivious string B-trees". In: *Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 2006, pp. 233–242. DOI: 10.1145/1142351.1142385.

[7] Michael A. Bender, Richard Cole, Erik D. Demaine, Martin Farach-Colton, and Jack Zito. "Two Simplified Algorithms for Maintaining Order in a List". In: *Algorithms — ESA 2002*. Ed. by Rolf Möhring and Rajeev Raman. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 152–164. ISBN: 978-3-540-45749-7. URL: https://link.springer.com/chapter/10.1007/3-540-45749-6_17.

[8] Michael A. Bender, Alex Conway, Martín Farach-Colton, Hanna Komlós, William Kuszmaul, and Nicole Wein. "Online List Labeling: Breaking the log2n Barrier". In: *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*. 2022, pp. 980–990. DOI: 10.1109/FOCS54457.2022.00096.

[9] Michael A. Bender, Erik D. Demaine, and Martin Farach-Colton. "Cache-Oblivious B-Trees". In: *SIAM Journal on Computing* 35.2 (2005), pp. 341–358. DOI: 10.1137/S0097539701389956. URL: https://doi.org/10.1137/S0097539701389956.

[10] Michael A. Bender, Martin Farach-Colton, Jeremy T. Fineman, Yonatan R. Fogel, Bradley C. Kuszmaul, and Jelani Nelson. "Cache-Oblivious Streaming B-Trees". In: *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA '07. San Diego, California, USA: Association for Computing Machinery, 2007, pp. 81–92. ISBN: 9781595936677. DOI: 10.1145/1248377.1248393. URL: https://doi.org/10.1145/1248377.1248393.

[11] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Bradley C. Kuszmaul. "Concurrent Cache-Oblivious b-Trees". In: *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA '05. Las Vegas, Nevada, USA: Association for Computing Machinery, 2005, pp. 228–237. ISBN: 1581139861. DOI: 10.1145/1073970.1074009. URL: https://doi.org/10.1145/1073970.1074009.

[12] Michael A. Bender and Haodong Hu. "An Adaptive Packed-Memory Array". In: *ACM Trans. Database Syst.* 32.4 (2007). ISSN: 0362-5915. DOI: 10.1145/1292609.1292616. URL: https://doi.org/10.1145/1292609.1292616.

[13] Guy E Blelloch. "Prefix sums and their applications". In: *Synthesis of Parallel Algorithms* (1990). URL: http://shelf2.library.cmu.edu/Tech/23445461.pdf.

[14] Jan Bulánek, Michal Koucký, and Michael Saks. "Tight Lower Bounds for the Online Labeling Problem". In: *Proceedings of the Forty-Fourth Annual ACM Symposium on Theory of Computing*. STOC '12. New York, New York, USA: Association for Computing Machinery, 2012, pp. 1185–1198. ISBN: 9781450312455. DOI: 10.1145/2213977.2214083. URL: https://doi.org/10.1145/2213977.2214083.

[15] Dean De Leo and Peter Boncz. "Fast Concurrent Reads and Updates with PMAs". In: *Proceedings of the 2nd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*. GRADES-NDA'19. Amsterdam, Netherlands: Association for Computing Machinery, 2019. ISBN: 9781450367899. DOI: 10.1145/3327964.3328497. URL: https://doi.org/10.1145/3327964.3328497.

[16] Dean De Leo and Peter Boncz. "Packed Memory Arrays - Rewired". In: *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 2019, pp. 830–841. DOI: 10.1109/ICDE.2019.00079.

[17] Paul F. Dietz and Ju Zhang. "Lower bounds for monotonic list labeling". In: *SWAT 90*. Ed. by John R. Gilbert and Rolf Karlsson. Berlin, Heidelberg: Springer Berlin Heidelberg, 1990, pp. 173–180. ISBN: 978-3-540-47164-6. URL: https://link.springer.com/chapter/10.1007/3-540-52846-6_87.

[18] Marie Durand, Bruno Raffin, and François Faure. "A packed memory array to keep moving particles sorted". In: *VRIPHYS 2012-9th Workshop on Virtual Reality Interaction and Physical Simulation*. The Eurographics Association. 2012, pp. 69–77. URL: https://inria.hal.science/hal-00762593/.

[19] Thomas A. Henzinger, Christoph M. Kirsch, Hannes Payer, Ali Sezgin, and Ana Sokolova. "Quantitative Relaxation of Concurrent Data Structures". In: *SIGPLAN Not.* 48.1 (2013), pp. 317–328. ISSN: 0362-1340. DOI: 10.1145/2480359.2429109. URL: https://doi.org/10.1145/2480359.2429109.

[20] Scott Huddleston and Kurt Mehlhorn. "A new data structure for representing sorted lists". In: *Acta informatica* 17 (1982), pp. 157–184. DOI: 10.1007/BF00288968.

[21] A. Itai, A. G. Konheim, and M. Rodeh. "A Sparse Table Implementation of Priority Queues". In: *Proceedings of the 8th Colloquium on Automata, Languages and Programming*. Ed. by S. Even and O. Kariv. Vol. 115. LNCS. Springer, 1981, pp. 417–431. ISBN: 3-540-10843-2. URL: https://link.springer.com/content/pdf/10.1007/3-540-10843-2_34.pdf.

[22] Alon Itai and Irit Katriel. "Canonical density control". In: *Information Processing Letters* 104.6 (2007), pp. 200–204. ISSN: 0020-0190. DOI: https://doi.org/10.1016/j.ipl.2007.07.001. URL: https://www.sciencedirect.com/science/article/pii/S0020019007001755.

[23] Gonzalo Navarro. *Compact data structures: A practical approach*. Cambridge University Press, 2016. DOI: 10.1017/CBO9781316588284.

[24] Peter Sanders, Kurt Mehlhorn, Martin Dietzfelbinger, and Roman Dementiev. *Sequential and Parallel Algorithms and Data Structures*. Springer, 2019. DOI: 10.1007/978-3-030-25209-0. URL: https://link.springer.com/book/10.1007/978-3-030-25209-0.

[25] Felix Martin Schuhknecht, Jens Dittrich, and Ankur Sharma. "RUMA Has It: Rewired User-Space Memory Access is Possible!" In: *Proc. VLDB Endow.* 9.10 (2016), pp. 768–779. ISSN: 2150-8097. DOI: 10.14778/2977797.2977803. URL: https://doi.org/10.14778/2977797.2977803.

[26] Brian Wheatman, Randal Burns, Aydın Buluç, and Helen Xu. "CPMA: An Efficient Batch-Parallel Compressed Set Without Pointers". In: (2023). arXiv: 2305.05055.

[27] Brian Wheatman, Randal Burns, Aydın Buluç, and Helen Xu. "Optimizing Search Layouts in Packed Memory Arrays". In: *2023 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*, pp. 148–161. DOI: 10.1137/1.9781611977561.ch13. URL: https://epubs.siam.org/doi/abs/10.1137/1.9781611977561.ch13.

[28] Brian Wheatman and Helen Xu. "A Parallel Packed Memory Array to Store Dynamic Graphs". In: *2021 Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*, pp. 31–45. DOI: 10.1137/1.9781611976472.3. URL: https://epubs.siam.org/doi/abs/10.1137/1.9781611976472.3.

[29] Brian Wheatman and Helen Xu. "Packed Compressed Sparse Row: A Dynamic Graph Representation". In: *2018 IEEE High Performance extreme Computing Conference (HPEC)*. 2018, pp. 1–7. DOI: 10.1109/HPEC.2018.8547566.

[30] Dan E. Willard. "A density control algorithm for doing insertions and deletions in a sequentially ordered file in a good worst-case time". In: *Information and Computation* 97.2 (1992), pp. 150–204. ISSN: 0890-5401. DOI: https://doi.org/10.1016/0890-5401(92)90034-D. URL: https://www.sciencedirect.com/science/article/pii/089054019290034D.

[31] Dan E. Willard. "Good Worst-Case Algorithms for Inserting and Deleting Records in Dense Sequential Files". In: *SIGMOD Rec.* 15.2 (1986), pp. 251–260. ISSN: 0163-5808. DOI: 10.1145/16856.16879. URL: https://doi.org/10.1145/16856.16879.

[32] Dan E. Willard. "Maintaining Dense Sequential Files in a Dynamic Environment (Extended Abstract)". In: *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*. STOC '82. San Francisco, California, USA: Association for Computing Machinery, 1982, pp. 114–121. ISBN: 0897910702. DOI: 10.1145/800070.802183. URL: https://doi.org/10.1145/800070.802183.

A. Appendix

A.1. Full Parameter Tuning Results for BBPMA

Tables A.1 to A.9 show extended results of the parameter tuning results described in Section 6.2.1 for the uniform, zipf and dense normal inputs and the three density configurations.

A.2. Full Block Size Evaluation Results for GBPMA

Figure A.1 shows the results of the block size tuning experiment for the GBPMA described in Section 6.3.1 for all inputs.

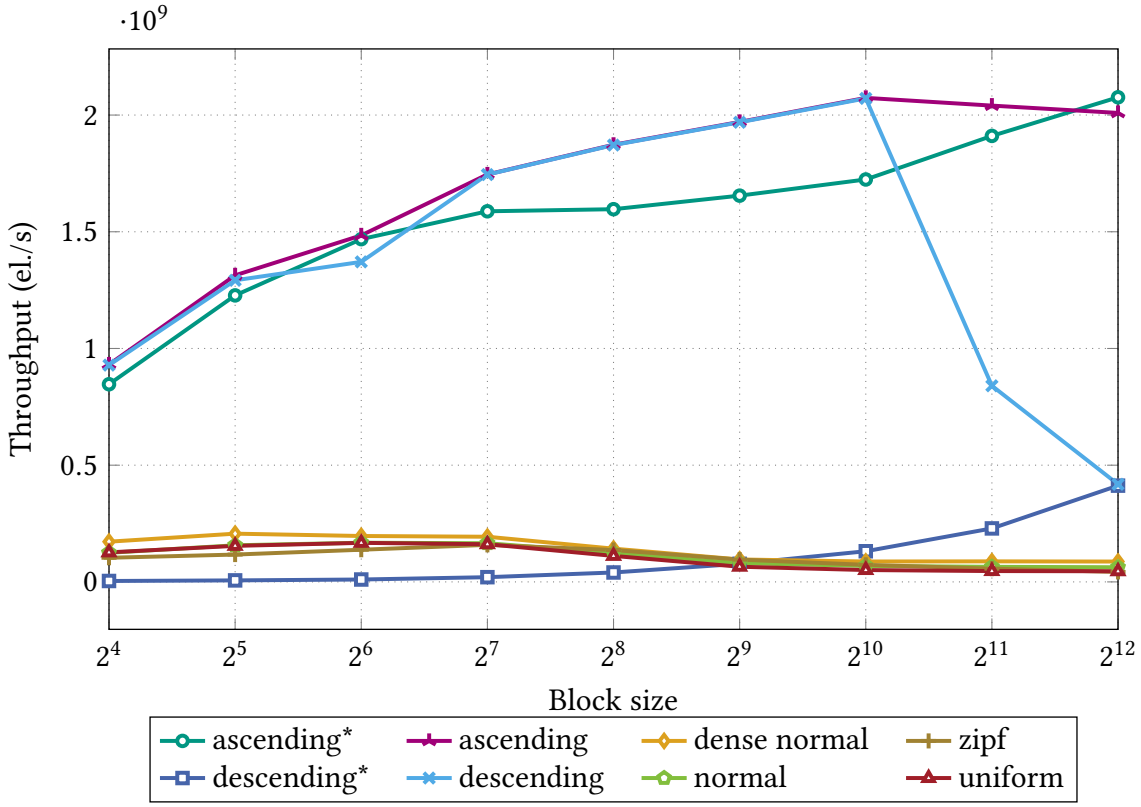


Figure A.1.: Insertion throughput in the GBPMA for different block sizes, using a growing factor of 1.8 and 64 threads. This plot shows the results for all inputs, of which some are included in Figure 6.9.

Size		Throughput [$\cdot 10^6$ elem. /s]						Search time [μs]
Block	Segment	Batch insertion				Scan		
		$p = 1$	$p = 4$	$p = 16$	$p = 64$	short	long	
64	256	3.51	12.04	28.24	41.74	14.69	254.30	1.74
	512	3.38	11.59	28.24	45.52	14.90	256.60	1.71
	1024	3.39	11.66	29.05	47.24	14.90	258.09	1.71
	2048	3.38	11.77	29.63	49.26	15.13	256.57	1.72
	4096	3.42	11.94	29.10	49.19	15.06	256.95	1.72
128	256	**3.97**	14.34	35.71	58.66	16.51	289.18	1.42
	512	3.91	14.11	36.16	60.82	16.95	292.23	1.42
	1024	3.82	13.96	36.43	62.63	16.77	298.03	1.38
	2048	3.96	**14.41**	**37.14**	**63.08**	16.48	298.64	1.40
	4096	3.86	14.14	36.03	54.56	17.18	297.06	1.41
256	256	3.69	13.89	34.56	54.26	18.50	417.50	1.12
	512	3.66	13.78	34.79	55.21	18.62	419.10	1.06
	1024	3.61	13.68	34.59	55.29	19.34	424.57	1.10
	2048	3.63	13.73	34.41	50.78	18.94	423.02	1.08
	4096	3.62	13.71	33.16	50.05	19.12	418.72	1.07
512	256	3.44	13.19	31.84	49.76	21.81	565.31	1.02
	512	3.37	12.95	31.77	49.69	21.72	562.98	0.99
	1024	3.43	13.07	31.60	46.63	21.40	559.36	1.03
	2048	3.36	12.86	31.05	46.29	22.87	566.29	0.97
	4096	3.42	13.04	30.91	45.79	23.10	560.59	0.99
1024	256	2.61	10.23	30.84	49.33	28.32	672.76	0.87
	512	2.85	11.10	31.80	47.76	27.95	672.57	0.88
	1024	2.60	10.20	30.58	47.07	28.26	672.69	0.88
	2048	2.82	11.02	31.17	45.93	28.34	673.44	**0.87**
	4096	2.60	10.17	30.04	44.00	27.76	674.00	0.90
2048	256	1.97	7.77	27.29	51.19	**28.82**	842.94	1.11
	512	1.98	7.82	27.35	51.13	28.57	843.03	1.11
	1024	1.97	7.78	27.15	50.67	28.49	843.86	1.11
	2048	1.98	7.79	27.14	49.73	27.59	844.31	1.11
	4096	1.97	7.76	26.83	48.77	28.36	844.31	1.11
4096	256	1.27	5.00	19.16	47.76	23.57	977.49	1.57
	512	1.00	3.96	15.26	41.51	24.07	971.81	1.57
	1024	1.28	5.04	19.07	47.24	25.07	980.79	1.50
	2048	1.00	3.96	15.22	40.93	25.43	**983.11**	1.50
	4096	1.27	5.01	18.97	46.31	24.82	973.62	1.54

Table A.1.: Parameter tuning results for the uniform input and density configuration α ($\tau^h_{\max} = 0.9$, $g = 1.8$). The best value of each column is marked in bold. Scan and search results are for the variant with $p = 64$.

Size		Throughput [$\cdot 10^6$ elem. /s]						Search time [μs]
Block	Segment	Batch insertion				Scan		
		$p = 1$	$p = 4$	$p = 16$	$p = 64$	short	long	
64	256	3.49	11.96	28.03	41.99	14.55	254.77	1.73
	512	3.40	11.67	28.27	44.97	15.05	256.06	1.72
	1024	3.34	11.56	29.14	47.53	14.66	256.35	1.73
	2048	3.39	11.81	29.53	49.12	14.75	256.36	1.70
	4096	3.40	11.87	29.28	49.16	15.04	257.41	1.69
128	256	**4.08**	**14.66**	35.93	58.37	16.52	291.78	1.41
	512	3.84	14.02	36.21	60.69	17.03	299.61	1.42
	1024	3.95	14.32	36.71	62.78	17.26	294.18	1.40
	2048	3.83	14.00	**36.80**	**62.87**	16.74	297.27	1.41
	4096	3.97	14.41	35.82	54.65	17.01	295.76	1.41
256	256	3.69	13.87	34.56	54.28	18.64	423.22	1.06
	512	3.62	13.66	34.63	54.94	18.66	425.24	1.13
	1024	3.63	13.70	34.65	55.51	19.10	430.84	1.08
	2048	3.59	13.57	34.29	50.70	19.49	423.87	1.06
	4096	3.63	13.74	33.47	50.24	19.45	420.50	1.08
512	256	3.38	13.02	31.89	49.51	21.64	555.11	0.99
	512	3.43	13.12	31.77	49.78	21.70	561.13	1.01
	1024	3.37	12.91	31.54	46.67	21.32	566.42	1.05
	2048	3.42	13.03	31.03	46.26	22.21	562.43	1.05
	4096	3.36	12.88	30.92	45.88	23.45	555.18	1.03
1024	256	2.83	11.03	31.76	49.56	28.22	672.61	0.88
	512	2.61	10.22	30.79	47.81	28.17	673.57	0.88
	1024	2.83	10.98	31.56	47.16	27.98	673.03	**0.88**
	2048	2.60	10.19	30.26	46.03	28.54	673.42	0.88
	4096	2.83	11.03	31.04	42.48	22.61	665.64	1.02
2048	256	1.98	7.81	27.39	51.16	27.99	843.44	1.11
	512	1.97	7.77	27.27	51.02	28.42	843.38	1.11
	1024	1.98	7.81	27.25	50.74	**28.75**	844.60	1.11
	2048	1.96	7.75	27.00	49.83	27.97	845.30	1.12
	4096	1.98	7.55	25.45	39.88	22.07	823.06	1.23
4096	256	1.00	3.95	15.26	41.68	24.45	974.71	1.52
	512	1.28	4.99	19.01	47.64	24.33	973.65	1.50
	1024	1.00	3.95	15.24	41.15	24.74	977.28	1.53
	2048	1.30	5.00	18.97	46.79	25.04	**978.03**	1.56
	4096	1.00	3.90	14.64	36.26	24.35	947.01	1.82

Table A.2.: Parameter tuning results for the uniform input and density configuration β ($\tau^h_{\max} = 0.9$, $g = 1.2$). The best value of each column is marked in bold. Scan and search results are for the variant with $p = 64$.

Size		Throughput [$\cdot 10^6$ elem. /s]						Search time [μs]
Block	Segment	Batch insertion				Scan		
		$p = 1$	$p = 4$	$p = 16$	$p = 64$	short	long	
64	256	3.50	12.00	27.66	41.62	14.62	254.35	1.70
	512	3.35	11.54	28.10	44.97	14.77	256.42	1.71
	1024	3.37	11.61	29.03	47.09	14.89	256.08	1.72
	2048	3.37	11.69	29.36	48.72	14.84	257.35	1.73
	4096	3.42	11.91	29.34	49.24	14.82	256.39	1.70
128	256	3.94	14.27	35.63	57.89	16.23	288.90	1.37
	512	**3.98**	**14.36**	36.15	60.46	16.53	297.46	1.42
	1024	3.82	13.96	36.40	62.25	16.73	296.13	1.39
	2048	3.95	14.35	**36.93**	**62.93**	16.62	305.60	1.44
	4096	3.84	14.13	35.88	54.39	16.73	297.16	1.41
256	256	3.68	13.84	34.44	54.11	18.34	415.71	1.07
	512	3.66	13.75	34.60	54.96	19.47	419.88	1.07
	1024	3.62	13.65	34.57	55.40	18.86	420.20	1.16
	2048	3.61	13.67	34.33	50.71	19.65	422.29	1.05
	4096	3.61	13.70	33.32	50.18	19.28	422.27	1.07
512	256	3.44	13.21	31.83	49.68	22.54	570.71	0.94
	512	3.34	12.82	31.79	49.84	21.50	556.66	1.02
	1024	3.42	13.06	31.63	46.71	21.41	572.20	1.11
	2048	3.35	12.85	31.10	46.19	22.21	563.75	1.02
	4096	3.42	13.03	30.90	45.97	23.34	563.34	1.00
1024	256	2.61	10.22	30.87	49.50	28.11	672.38	0.87
	512	2.85	11.07	31.80	47.85	28.20	672.70	0.88
	1024	2.60	10.17	30.60	47.05	28.62	673.00	**0.87**
	2048	2.82	11.02	31.16	45.90	28.57	673.01	0.88
	4096	2.60	10.17	30.04	44.04	27.54	674.10	0.88
2048	256	1.97	7.78	27.24	51.19	**28.64**	840.93	1.11
	512	1.98	7.81	27.37	50.90	28.25	844.29	1.11
	1024	1.96	7.68	26.97	50.65	27.33	846.06	1.12
	2048	1.98	7.78	27.07	49.85	28.52	843.38	1.12
	4096	1.97	7.74	26.82	48.61	27.89	843.69	1.11
4096	256	1.28	5.01	19.13	47.88	25.36	**988.35**	1.51
	512	1.00	3.96	15.16	41.48	24.63	979.69	1.47
	1024	1.27	4.98	18.98	47.01	24.80	970.89	1.56
	2048	1.00	3.96	15.19	40.79	25.30	982.16	1.57
	4096	1.29	5.06	19.08	46.03	24.73	975.35	1.57

Table A.3.: Parameter tuning results for the uniform input and density configuration γ ($\tau^h_{\max} = 0.7$, $g = 1.8$). The best value of each column is marked in bold. Scan and search results are for the variant with $p = 64$.

Size		Throughput [$\cdot 10^6$ elem. /s]						Search time [μs]
Block	Segment	Batch insertion				Scan		
		$p = 1$	$p = 4$	$p = 16$	$p = 64$	short	long	
64	256	7.56	16.38	34.56	76.18	18.76	252.84	1.22
	512	7.13	15.96	34.39	76.96	18.72	253.92	1.25
	1024	6.74	15.41	33.31	73.28	18.58	254.99	1.25
	2048	6.19	14.78	31.45	68.10	18.94	255.51	1.27
	4096	5.80	14.02	30.18	62.58	18.72	256.25	1.26
128	256	**8.66**	**18.18**	**44.05**	100.03	21.89	297.79	0.99
	512	8.49	17.83	43.65	**100.08**	21.87	297.76	0.99
	1024	8.01	17.26	42.53	97.80	21.81	296.79	1.00
	2048	7.93	17.18	40.62	92.92	22.09	298.99	0.98
	4096	7.40	16.39	38.90	71.60	22.77	298.51	0.97
256	256	8.18	16.48	41.10	89.33	26.16	430.21	0.82
	512	8.06	16.52	41.57	90.05	25.73	429.52	0.82
	1024	8.02	16.06	40.18	84.42	25.54	431.20	0.81
	2048	7.81	15.83	39.57	75.59	25.74	427.62	0.81
	4096	7.71	15.40	37.46	73.05	26.23	430.87	0.80
512	256	7.41	14.59	34.97	71.07	29.49	580.94	0.77
	512	7.16	14.28	34.55	68.86	29.92	577.98	0.76
	1024	7.33	14.37	34.50	67.56	30.49	575.76	0.77
	2048	7.07	14.09	33.81	65.27	30.51	581.86	**0.76**
	4096	7.19	14.14	32.96	62.66	30.77	568.08	0.76
1024	256	5.05	10.53	26.59	56.12	33.15	678.71	0.90
	512	5.55	10.79	26.16	55.49	33.62	679.03	0.90
	1024	5.03	10.49	26.31	55.42	33.95	679.13	0.91
	2048	5.51	10.75	25.85	54.39	33.12	678.65	0.90
	4096	5.01	10.43	25.91	53.46	33.68	679.28	0.89
2048	256	3.52	7.77	19.52	43.93	33.68	846.91	1.13
	512	3.61	7.79	19.63	44.00	33.68	847.17	1.14
	1024	3.51	7.76	19.45	43.65	**34.05**	846.77	1.14
	2048	3.61	7.78	19.55	43.26	33.19	848.62	1.14
	4096	3.51	7.76	19.32	42.24	33.43	846.54	1.13
4096	256	2.26	5.07	13.31	34.41	27.95	972.51	1.63
	512	1.81	4.43	12.92	34.25	28.06	969.82	1.63
	1024	2.26	5.07	13.26	34.19	28.85	**974.77**	1.62
	2048	1.81	4.43	12.88	33.96	28.26	972.53	1.62
	4096	2.25	5.05	13.19	33.61	28.53	971.97	1.62

Table A.4.: Parameter tuning results for the zipf input and density configuration α ($\tau^h_{\max} = 0.9$, $g = 1.8$). The best value of each column is marked in bold. Scan and search results are for the variant with $p = 64$.

Size		Throughput [$\cdot 10^6$ elem. /s]						Search time [μs]
Block	Segment	Batch insertion				Scan		
		$p = 1$	$p = 4$	$p = 16$	$p = 64$	short	long	
64	256	7.52	16.36	35.05	77.31	18.52	253.30	1.26
	512	7.25	16.10	34.50	77.04	19.02	253.93	1.24
	1024	6.68	15.43	33.22	73.03	18.94	255.23	1.24
	2048	6.18	14.66	31.21	68.40	18.31	255.52	1.26
	4096	5.75	14.07	30.23	62.51	19.21	256.07	1.25
128	256	**8.93**	**18.59**	**43.85**	**99.98**	21.89	296.04	1.01
	512	8.36	17.73	43.70	99.78	22.09	298.15	0.98
	1024	8.31	17.67	42.57	98.18	22.50	297.21	0.97
	2048	7.68	16.78	40.86	92.31	21.36	300.15	0.97
	4096	7.60	16.58	38.35	71.58	22.77	300.31	0.98
256	256	8.07	16.70	41.63	89.93	25.61	429.72	0.80
	512	8.08	16.25	41.01	89.95	25.80	431.93	0.79
	1024	7.95	16.20	40.65	84.50	25.69	432.24	0.80
	2048	7.85	15.58	39.25	75.24	22.60	425.79	0.90
	4096	7.65	15.54	38.19	73.03	25.59	428.97	0.79
512	256	7.21	14.40	34.74	71.06	30.25	575.52	0.76
	512	7.36	14.47	34.89	68.84	30.39	575.78	0.77
	1024	7.13	14.19	34.41	67.74	30.14	572.63	0.76
	2048	7.26	13.85	31.72	57.38	25.80	557.08	0.87
	4096	7.01	13.88	32.71	63.35	31.05	579.94	**0.76**
1024	256	5.53	10.77	26.25	55.72	32.48	678.14	0.89
	512	5.04	10.52	26.52	55.90	33.97	679.61	0.90
	1024	5.51	10.73	25.95	55.17	34.08	679.59	0.89
	2048	4.99	10.37	25.27	51.06	32.10	654.14	1.02
	4096	5.51	10.72	25.63	52.82	**34.24**	678.86	0.89
2048	256	3.61	7.79	19.67	44.07	32.93	846.48	1.13
	512	3.51	7.77	19.50	43.81	33.06	847.28	1.14
	1024	3.61	7.79	19.62	43.83	33.50	848.19	1.14
	2048	3.48	7.73	18.96	42.19	28.86	816.10	1.26
	4096	3.61	7.78	19.44	42.71	33.85	848.26	1.14
4096	256	1.81	4.43	12.90	34.31	29.07	**981.70**	1.60
	512	2.27	5.05	13.27	34.33	28.54	976.05	1.63
	1024	1.81	4.43	12.89	34.15	27.97	972.85	1.61
	2048	2.25	5.02	13.25	34.03	28.69	973.29	1.61
	4096	1.81	4.43	12.85	33.30	28.99	980.66	1.58

Table A.5.: Parameter tuning results for the zipf input and density configuration β ($\tau^h_{\max} = 0.9$, $g = 1.2$). The best value of each column is marked in bold. Scan and search results are for the variant with $p = 64$.

Size		Throughput [$\cdot 10^6$ elem. /s]						Search time [μs]
Block	Segment	Batch insertion				Scan		
		$p = 1$	$p = 4$	$p = 16$	$p = 64$	short	long	
64	256	7.17	15.12	30.37	58.54	16.62	251.38	1.68
	512	6.79	14.76	30.61	59.34	16.83	252.02	1.68
	1024	6.41	14.44	29.82	59.09	16.84	252.13	1.62
	2048	5.97	13.94	29.15	56.37	17.01	253.06	1.60
	4096	5.77	14.08	29.83	62.68	18.91	256.10	1.25
128	256	8.33	17.22	39.64	82.58	19.55	293.55	1.31
	512	**8.38**	**17.38**	39.59	85.04	19.38	293.51	1.31
	1024	7.84	16.78	40.22	84.20	19.79	294.92	1.28
	2048	7.81	17.07	40.57	**92.94**	22.08	301.65	0.96
	4096	7.24	15.84	36.97	65.18	21.35	292.89	1.16
256	256	8.06	16.29	39.43	82.53	22.04	427.09	1.02
	512	7.99	16.28	**41.22**	84.56	24.36	424.66	0.92
	1024	8.02	15.98	40.14	84.53	25.88	429.78	0.79
	2048	7.75	15.82	39.47	75.72	27.00	428.04	0.79
	4096	7.67	15.29	37.16	73.03	26.82	434.38	0.77
512	256	7.41	14.40	35.04	69.19	27.47	584.09	0.89
	512	7.11	14.16	34.60	69.11	30.30	580.98	**0.75**
	1024	7.31	14.39	34.44	67.48	31.52	579.91	0.76
	2048	7.04	13.99	33.59	65.03	31.56	584.13	0.78
	4096	7.18	14.14	33.06	62.75	30.96	567.07	0.79
1024	256	5.03	10.52	26.60	56.12	33.17	679.18	0.89
	512	5.48	10.79	26.18	55.43	**34.07**	680.03	0.88
	1024	5.03	10.48	26.33	55.47	33.51	663.87	0.90
	2048	5.49	10.73	25.82	54.20	33.00	678.44	0.90
	4096	5.01	10.44	25.90	53.44	33.47	679.82	0.91
2048	256	3.51	7.77	19.51	43.87	33.19	846.10	1.14
	512	3.61	7.79	19.63	43.99	32.75	849.40	1.13
	1024	3.49	7.69	19.28	43.55	32.82	847.07	1.14
	2048	3.61	7.77	19.52	43.50	33.91	848.20	1.14
	4096	3.51	7.74	19.28	42.15	33.59	848.30	1.14
4096	256	2.26	5.05	13.28	34.42	27.78	972.81	1.59
	512	1.81	4.43	12.91	34.25	27.44	974.99	1.63
	1024	2.25	5.04	13.26	34.27	28.20	977.36	1.63
	2048	1.81	4.43	12.89	33.88	28.68	972.56	1.62
	4096	2.27	5.08	13.25	33.69	28.87	**981.46**	1.62

Table A.6.: Parameter tuning results for the zipf input and density configuration γ ($\tau^h_{\max} = 0.7$, $g = 1.8$). The best value of each column is marked in bold. Scan and search results are for the variant with $p = 64$.

Size		Throughput [$\cdot 10^6$ elem. /s]						Search time [μs]
Block	Segment	Batch insertion				Scan		
		$p = 1$	$p = 4$	$p = 16$	$p = 64$	short	long	
64	256	5.65	18.75	46.08	69.00	17.38	251.45	1.36
	512	5.60	18.82	47.86	73.48	17.35	251.46	1.38
	1024	5.74	19.72	50.26	75.27	17.40	252.65	1.36
	2048	5.68	19.43	49.89	74.82	17.49	251.95	1.36
	4096	5.77	19.97	50.81	72.35	17.38	251.77	1.41
128	256	6.30	22.82	59.65	96.84	20.40	282.54	1.08
	512	6.37	22.96	62.11	100.23	21.09	283.12	1.04
	1024	6.22	22.78	63.23	99.11	20.67	283.52	1.06
	2048	**6.56**	**23.96**	64.46	96.53	20.65	283.01	1.04
	4096	6.31	23.20	63.63	89.93	20.52	288.24	1.03
256	256	6.18	23.29	63.28	100.77	23.62	416.50	0.95
	512	5.98	22.64	63.21	99.90	24.29	418.14	0.85
	1024	6.16	23.55	**64.63**	97.94	23.33	412.74	0.86
	2048	6.00	22.94	63.49	92.56	23.62	415.28	0.86
	4096	6.21	23.71	63.20	89.77	23.74	416.37	0.89
512	256	5.58	21.64	64.28	**103.09**	26.61	561.69	0.89
	512	5.36	20.90	63.41	100.71	26.25	565.43	0.95
	1024	5.61	21.79	64.47	97.92	26.72	565.49	0.93
	2048	5.38	20.94	62.36	94.47	27.18	563.32	0.93
	4096	5.57	21.66	62.76	90.32	26.78	569.01	0.92
1024	256	3.49	13.61	48.11	100.95	28.99	667.03	0.86
	512	4.12	16.02	55.04	102.74	29.26	666.94	0.86
	1024	3.49	13.61	47.95	97.79	29.53	666.91	0.86
	2048	4.10	15.97	54.35	98.20	29.12	666.71	**0.84**
	4096	3.49	13.63	47.55	92.86	**29.55**	667.25	0.87
2048	256	2.34	9.01	33.44	88.78	29.08	840.30	1.08
	512	2.44	9.49	35.27	91.16	29.04	838.43	1.08
	1024	2.33	9.00	33.44	87.85	27.50	839.62	1.08
	2048	2.44	9.49	35.08	88.86	28.24	841.97	1.07
	4096	2.33	9.01	32.95	84.50	27.81	840.24	1.06
4096	256	1.49	5.62	21.43	59.32	24.44	974.16	1.46
	512	1.11	4.31	16.20	46.40	24.10	978.36	1.46
	1024	1.48	5.78	21.66	59.75	24.40	**981.60**	1.53
	2048	1.11	4.32	16.18	46.13	24.42	981.48	1.46
	4096	1.47	5.72	21.19	58.07	24.88	979.47	1.43

Table A.7.: Parameter tuning results for the dense normal input and density configuration α ($\tau^h_{\max} = 0.9$, $g = 1.8$). The best value of each column is marked in bold. Scan and search results are for the variant with $p = 64$.

Size		Throughput [$\cdot 10^6$ elem. /s]						Search time [μs]
Block	Segment	Batch insertion				Scan		
		$p = 1$	$p = 4$	$p = 16$	$p = 64$	short	long	
64	256	5.54	18.54	45.87	68.06	17.23	251.10	1.35
	512	5.68	19.20	48.41	73.59	17.47	251.67	1.39
	1024	5.61	19.40	49.47	74.83	17.67	251.97	1.39
	2048	5.72	19.60	49.79	74.46	17.43	251.75	1.36
	4096	5.66	19.80	50.67	72.82	17.19	252.15	1.42
128	256	6.54	23.42	60.34	97.88	20.62	285.13	1.08
	512	6.26	22.66	61.72	100.13	20.39	289.08	1.07
	1024	**6.55**	23.70	63.64	99.84	20.32	282.36	1.07
	2048	6.27	22.98	63.63	97.18	20.65	283.07	1.05
	4096	6.53	**23.88**	63.35	89.53	20.58	279.98	1.02
256	256	5.96	22.43	62.57	100.27	23.14	420.70	0.90
	512	6.16	23.31	64.13	100.77	22.94	420.94	0.84
	1024	5.97	22.76	63.63	97.66	23.54	423.92	0.85
	2048	6.16	23.49	64.06	92.68	24.01	419.52	0.86
	4096	6.01	23.03	62.54	90.34	23.61	416.87	0.92
512	256	5.37	20.83	62.79	102.57	27.15	564.74	0.86
	512	5.56	21.65	**64.42**	100.77	27.62	560.09	0.85
	1024	5.38	20.93	63.24	97.34	26.62	559.91	0.96
	2048	5.58	21.71	63.68	94.91	27.54	555.33	0.90
	4096	5.38	20.96	61.61	89.81	27.35	567.97	0.90
1024	256	4.09	15.92	54.66	**103.66**	28.59	667.66	0.87
	512	3.49	13.62	48.21	98.91	28.62	666.28	0.86
	1024	4.09	15.99	54.76	100.54	29.68	666.53	0.85
	2048	3.50	13.42	46.84	95.12	29.13	668.00	**0.84**
	4096	4.10	16.02	53.97	95.61	**29.75**	666.38	0.88
2048	256	2.44	9.50	35.27	91.46	29.04	840.31	1.07
	512	2.33	9.01	33.43	88.57	27.77	839.60	1.08
	1024	2.44	9.49	35.24	90.69	27.53	840.41	1.06
	2048	2.33	9.00	33.33	87.21	26.99	840.15	1.08
	4096	2.44	9.48	34.81	85.95	28.83	839.53	1.08
4096	256	1.11	4.31	16.22	46.52	24.54	981.05	1.46
	512	1.49	5.72	21.53	59.57	23.83	975.97	1.47
	1024	1.11	4.31	16.22	46.35	24.79	982.48	1.47
	2048	1.46	5.57	19.81	47.18	18.70	941.63	1.73
	4096	1.11	4.31	16.16	45.60	24.70	**986.30**	1.50

Table A.8.: Parameter tuning results for the dense normal input and density configuration β ($\tau^h_{\max} = 0.9$, $g = 1.2$). The best value of each column is marked in bold. Scan and search results are for the variant with $p = 64$.

Size		Throughput [$\cdot 10^6$ elem. /s]						Search time [μs]
Block	Segment	Batch insertion				Scan		
		$p = 1$	$p = 4$	$p = 16$	$p = 64$	short	long	
64	256	5.55	18.65	45.41	67.00	17.56	249.78	1.44
	512	5.52	18.94	47.33	73.37	17.60	250.46	1.45
	1024	5.68	19.56	49.46	74.43	17.40	249.84	1.41
	2048	5.60	19.52	49.64	74.63	17.05	249.49	1.47
	4096	5.73	19.74	49.86	72.67	17.92	249.18	1.39
128	256	6.23	22.53	59.52	97.04	20.53	282.26	1.10
	512	**6.48**	23.63	62.50	100.50	20.29	284.92	1.12
	1024	6.21	22.83	62.95	99.56	21.27	284.60	1.10
	2048	6.46	**23.73**	63.51	97.09	19.77	285.19	1.12
	4096	6.25	23.13	63.37	88.15	20.28	277.97	1.11
256	256	6.15	23.20	62.89	99.80	23.62	418.86	0.91
	512	5.97	22.61	62.87	99.75	22.92	414.68	0.89
	1024	6.17	23.47	64.19	98.38	23.59	413.60	0.88
	2048	5.96	22.77	62.81	91.53	23.14	413.60	0.95
	4096	6.19	23.57	63.51	87.80	23.84	415.84	0.89
512	256	5.57	21.65	63.63	**101.63**	26.19	560.19	0.94
	512	5.33	20.84	63.03	101.42	26.62	560.07	0.95
	1024	5.54	21.61	**64.25**	96.48	26.35	554.54	0.90
	2048	5.34	20.82	62.15	93.37	26.45	559.59	0.91
	4096	5.56	21.61	62.50	90.74	26.10	559.88	0.95
1024	256	3.47	13.53	48.05	99.95	28.10	666.06	0.87
	512	4.04	15.88	54.32	99.90	27.01	665.06	**0.85**
	1024	3.49	13.61	48.02	97.04	27.99	665.60	0.85
	2048	4.09	15.85	53.86	97.45	**28.56**	666.61	0.86
	4096	3.49	13.64	47.39	90.46	28.40	666.98	0.87
2048	256	2.33	9.02	33.62	88.11	26.57	838.24	1.08
	512	2.43	9.48	35.23	90.79	26.99	839.23	1.09
	1024	2.32	8.97	33.38	87.60	27.16	840.06	1.09
	2048	2.44	9.49	35.02	88.38	27.31	839.41	1.07
	4096	2.33	9.01	32.91	84.73	28.08	839.37	1.08
4096	256	1.48	5.74	21.37	59.51	23.89	984.58	1.50
	512	1.11	4.31	16.18	46.44	23.75	980.52	1.45
	1024	1.46	5.73	21.40	58.93	24.07	978.01	1.45
	2048	1.11	4.31	16.17	46.04	24.20	974.43	1.50
	4096	1.50	5.81	21.43	57.85	25.03	**987.53**	1.49

Table A.9.: Parameter tuning results for the dense normal input and density configuration γ ($\tau^h_{\max} = 0.7$, $g = 1.8$). The best value of each column is marked in bold. Scan and search results are for the variant with $p = 64$.