Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Eskild Hein Trøen

Evaluating established B⁺-tree access methods in a parallel processing environment

Master's thesis in Computer Science
Supervisor: Svein Erik Bratsberg
May 2022

NTNU
Kunnskap for en bedre verden

Eskild Hein Trøen

Evaluating established B+-tree access methods in a parallel processing environment

Master's thesis in Computer Science
Supervisor: Svein Erik Bratsberg
May 2022

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU

Kunnskap for en bedre verden

Abstract

In order to keep achieving higher performance and throughput, computers nowadays trend towards employing an increasing number of processing units. The modern hardware architecture demands horizontal scalability by applications and systems wanting to operate most efficiently. Within the ability to scale horizontally lies the ability to operate concurrently, and in parallel. Database management systems (DBMSs) are complex systems with a rich history, always seeking to improve in ways that allows most efficient management of data, and handling of requests. For the purposes of speeding up the very common DBMS task of retrieving records in response to certain search conditions, additional auxiliary access structures, called indexes, are created and made use of. The B/B$^+$-tree is a commonly used database index structure, with origins dating all the way back to 1970, a time when most computers only had a single processing unit. There are many attempts at making this structure better fit for the modern hardware architecture, e.g. [1]–[5]. These attempts are varied in approaches and techniques applied.

This master's thesis makes its own attempt at parallelizing the already established access methods of the B$^+$-tree. The parallel B$^+$-tree implementation is sustained in-memory, makes use of a thread pool design pattern, and supports batch processing of operations grouped by type, as well as so-called single key operations. Futhermore, Bloom filters can optionally be enabled and essential variables, such as order, the number of threads, and the number of base B$^+$-trees, can be configured.

The results obtained when comparing the parallel B$^+$-tree implementation against a single threaded B$^+$-tree basline, show that opting to leverage parallel processing capabilities will only ever be worthwhile, if the degree of parallelism is sufficient, and the cost induced by task creation and managment becomes negligible compared to task execution time. This motivates the use of parallel batch processing, and application of cost-benefit analysis to determine when to execute in parallel.

Sammendrag

For å kunne fortsette å oppnå stadig større ytelse og gjennomstrømning, benytter datamaskiner i dag et økende antall prosesseringsenheter. Den moderne maskinvarearkitekturen krever horisontal skalerbarhet av applikasjoner og systemer som ønsker å operere mest mulig effektivt. Det å kunne skalere horisontalt innbefatter også evnen til å operere samtidig, og parallelt. Databasehåndteringssystemer (DBMSer) er komplekse systemer med en rik historie, samt en konstant søken etter forbedring på de områdene som muliggjør mest mulig effektiv håndtering av data, og behandling av forespørsler. Med det formålet å forbedre responstiden på den vanlige DBMS oppgaven som går ut på å få tak i poster som tilfredsstiller visse søkebetingelser, opprettes og benyttes supplerende behjelpelige tilgangsstrukturer kalt indekser. B/B$^+$-treet er en ofte brukt databaseindeksstruktur. Denne strukturen ble først introdusert i 1970, en tid da de fleste datamaskiner bare hadde en enkelt prosesseringsenhet. Det finnes mange forsøk på å tilpasse og modifisere B/B$^+$-treet slik at samspillet blir best mulig i møte med den moderne maskinvarearkitekturen, f.eks. [1]–[5]. Disse forsøkene er varierte både i tilnærming og teknikker benyttet.

Denne masteroppgaven gjør sitt eget forsøk på å parallellisere de allerede etablerte tilgangsmetodene til B$^+$-treet. Implementasjonen av det parallelle B$^+$-treet som presenteres opprettholdes i minnet, benytter "thread pool" designmønsteret, og støtter batchprosessering av operasjoner gruppert etter type, så vel som enkeltoperasjoner. Dessuten kan Bloom-filtre valgfritt benyttes og essensielle variabler, som den øvre grensen av en nodes antall barn, antall tråder, og antall vanlige B$^+$-tree, konfigureres.

Resultatene som ble oppnådd på bakgrunn av å sammenligne ytelsen til den parallelle implementasjonen av B$^+$-treet, og grunnlinjen etablert av det vanlige B$^+$-treet som kun benytter en enkelt tråd, viser at beslutningen om å velge å utnytte tilgjengelig parallell prosesseringskapabilitet, bare vil være verdt det dersom graden av parallellitet er tilstrekkelig, og kostnaden indusert av oppgaveoppretting og håndtering blir ubetydelig sammenlignet med oppgaveutførelsestiden. Dette motiverer bruken av parallell batchprosessering og bruk av nytte-kostnadsanalyse for å beslutte når parallell utførelse skal foretas.

Preface

This master's thesis was written during the spring of 2022 at the Department of Computer Science (IDI) at the Norwegian University of Science and Technology (NTNU). The dissertation is the final delivery in the 5 years Computer Science master's degree program, put forward for the university degree.

Acknowledgements

I would like to thank my supervisor Svein Erik Bratsberg for his assistance and guidance during the entire process of working with this thesis and its matters.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

In 1965, Moore's law [6] was introduced. It simply states that the number of transistors on a dense integrated circuit doubles about every two years. For those concerned with data technological advancements, and the ones wanting to get the most out of their computers, the law for a long time yielded a simplistic solution to problems only solvable by use of more processing power; wait for some time, and then exchange the central processing unit (CPU) for a more powerful CPU, once available. However, in the modern day and age Moore's law does not seem to keep up [7]. The decline has been known to come sooner or later, seeing as, after all, there is a hard physical limit on how small a integrated circuit's architecture can become. Emerging from the realisation of the decline, is the realisation that further performance improvements must largely be driven by horizontal scaling. Thus, the modern hardware architecture employes multiple processing units, and facilitates for high degree parallelism, in order to further increase computer performance.

A database management system (DBMS) must also adapt to the modern hardware architecture in the best ways possible. The management system will deal with a lot of different tasks to manage the data it stores. Although these tasks are diverse, the DBMS always tries to complete them efficiently. In this process a index is often involved. Many of the traditional index designs which the DBMS might rely on, were invented and first designed a long time ago. A time when parallel processing was not widespread at all. One of these indexes is the B^+-tree. Its properties as a single entity is well-known, but when it comes to using the structure in a parallel processing environment, less is known about the ideal adjustments to make, and the ideal techniques to employ. Of course, commercial DBMSs are among the entities utmost interested in index structure performance, and therefore they continuously try to improve on designs and implementations, but as is often the case with commercialization, there seems to be some secrecy involved in their works.

Motivated by the modern hardware architecture's demand on horizontally scalable designs and implementations, this master's thesis aims to study some of many potential adjustments that can be made to the classical B^+-tree index, in hopes of effectively utilizing available parallel processing capabilities.

1.1 Research goals

The following research goals, formed on the basis of the introductory background and motivation, have been established to set the master's thesis scope:

1. Implement a standard, memory-resident B^+-tree.
2. Design and implement a parallel B^+-tree. That is, a version of the B^+-tree with parallelized operations.
3. Carry out performance evaluation of both B^+-trees operations.
4. Compare the two, and in doing so identify bottlenecks and potential for improvement of the parallel implementation.
5. Evaluate the parallel implementation's suitability and potential in the modern computing environment.

1.2 Limitations

Some limitations constraining a project or thesis will always be present. In this regard it should be noted that the master's thesis has a timeframe of 20 weeks. This of course implies that the workload undertaken is confined by the timeframe. To avoid unneccessary diversions, keep the thesis consistent, and produce a thesis report that digs deep into certain aspects, the chapters concerning implementation and results (i.e. chapter 3 and chapter 4), thus focuses on a particular index structure, namely the B^+-tree. Even though, as is stated during the end of section 5.2, it would in its own right be interesting to examine and attempt parallelization of other classical index structures.

Furthermore, the specific results obtained pertain to the context in which they were achieved. What is meant by this is that the diversity of computer configurations, both in terms of hardware and software, will to the greatest extent affect the results obtained. Nevertheless, an effort has been made to implement the baseline B^+-tree and the parallel B^+-tree, such that their relative differences are preserved accross different architectures which have parallel operability. Among other things, this is done by use of the C++ programming language, which is CPU architecture dependently compiled.

1.3 Thesis structure

The remaining chapters that are to be presented, revolves around the following topics and themes:

- Chapter 2: Background

  Presents background information covering three key topics: index structures (what they are, their nature in general, and coverage of different types), parallel programming (the idea, some distinctions, how it works, and its potential), and concurrency mechanisms (the mechanisms allowing for coordination and cooperation during stages of parallelization requiring mutual exclusion). Additionally, some related work is presented at the end of this chapter.

  This chapter should establish common ground, as the reader is provided with general knowledge of the domain explored in this master's thesis. The chapter is foundational, in that it makes the thesis' undertaking more understandable, as well as makes the reader better equipped for reading the ensuing chapters which dives deeper into certain aspects of parallelization through hands-on implementation and evaluation.

- Chapter 3: Implementation

  Describes and details the thesis' programming work and development. A standard B^+-tree index has been implemented. Furthering this work, the standard B^+-tree implementation has been used as the core of a parallel B^+-tree design and implementation. The parallel implementation parallelizes all operations in two distinct modes, termed single key operation mode and batch operation mode.

- Chapter 4: Results and Discussion

  Evaluates performance of the parallel B^+-tree implementation in various configurations. Throughput measured in operations per second is used as the main performance evaluator, and comparison is done against representative single threaded B^+-tree baseline configurations. The results obtained are analyzed and discussed. Bottlenecks are identified and potential solutions proposed. Supporting the performance measurements and the behavior observed, the reader will find a profiling section, containing detailed profiling results of two specific, representative of low and high performance, parallel B^+-tree runs.

- Chapter 5: Conclusion and Future work

  Concludes the thesis' by summarizing its work and key findings. Furthermore, some future work originating from the thesis' entirety is put forward.

Some project work revolving around the same themes and topics as this master's thesis, was finished by this thesis' author during fall of 2021 [8]. This master's thesis is a continuation of the preceding body of work. Whereas the preceding project

work lays the groundwork for this master's thesis, first and foremost through literature study and practical domain exploration, the master's thesis expands and continues the work by presenting a complete parallel implementation, doing more structured analysis and evaluation of results, as well as improve and present the believed most relevant background knowledge and related work. The introductory parts of chapter 1, together with chapter 2, takes the most inspiration from the preceding project work, and it is thus fitting to accredit them as being loosely based on their respective, counterpart chapters found in [8].

Chapter 2

Background

This chapter serves to provide sufficent theoretical knowledge within the realm of index structures and parallel programming. Additionally, a selection of related work will be presented.

2.1 Index structures

What is a index, and what is its purpose? Well, data records will get stored in some form of primary organization, say unordered, ordered, or hashed. An index is an additional auxiliary access structure that has the main purpose of speeding up retrieval of records in response to certain search conditions [9, p. 601]. These secondary access paths will not affect the physical placement of the data records on disk, but will enable efficient access to relevant records based on the indexing fields used to create the index. Despite there being some overhead associated with the creation and usage of an index, intelligent construction of indexes based on identified common access patterns, will in general improve the performance of a DBMS. Indexing structures are varied and come in a lot of different configurations. For example there are single-field and multi-field indexes, single-level and multi-level indexes, primary key and secondary key indexes, clustered and unclustered indexes, and numerous other categories which one could describe indexes by.

Throughout the years a multitude of different index structures have been proposed. This section gives an overview over some of the most prominent ones that continue to see widespread use, both in DBMSs and file systems.

2.1.1 B-tree

The B-tree [10] is one of the most well-known index structures. It was invented in 1970 as a generalization of the binary search tree, thus allowing for nodes with more than two children. The nodes of a B-tree contains data records consisting of the index key and its associated value(s). The keys of an internal node are sorted in ascending order. Between the keys, pointers to nodes at the next level are present. These keys in conjunction with the pointers guide searches through the tree by acting as seperation values. For example if we are searching for the data record with key k, we follow a pointer in an internal node if $k_1 < k < k_2$, where k_1, k_2 are the keys on both sides of the pointer, and $k_2 > k_1$.

The order (branching factor) of the B-tree represents an upper bound on the allowable number of children an internal node can have. Conversely the degree of the tree represents a lower bound on the number of children an internal node can have. The only exception to the lower bound is the root which has at least two children if it is not a leaf. As formalized in [11], a B-tree of order m is a tree that satisfies the following properties:

i) Every node has at most m children.

ii) Every node, except for the root and the leaves, has at least $\lceil \frac{m}{2} \rceil$ children.

iii) The root has at least 2 children (unless it is a leaf).

iv) All leaves appear on the same level, and carry no information.

v) A nonleaf node with k children contains $k - 1$ keys.

The only point that might need elaboration is the one specifying that leaves carry no information. Leaves as defined in this source may by other authors be viewed as the internal nodes one level above. In essence what is considered information by this source is the information garnering the location of a record's final destination, as localized by the key, in the index structure. Thus, since the search terminates in a leaf node, if reached, there is no more information to be found here. Viewed in another way: no search path follows through a leaf. Figure 2.1 shows an example of the B-tree structure.

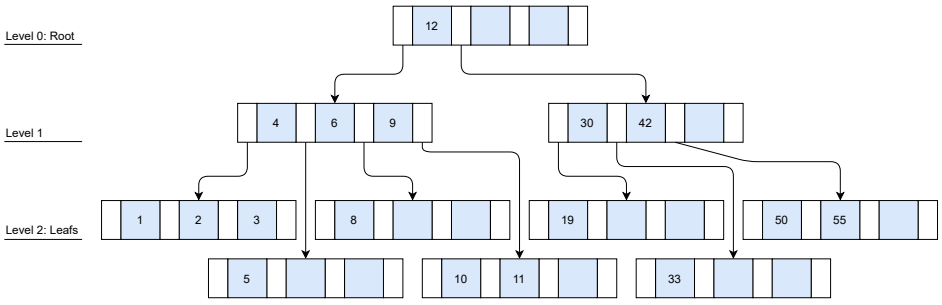


Figure 2.1: Example structure of a B-tree with order $m = 4$.

At last it should be mentioned that the structure has both average and worst case time complexity of $\mathcal{O}(\log n)$ for operations such as insert, delete and search, and with regards to space complexity the structure performs linearly $\mathcal{O}(n)$ for both average and worst case. The structure's maintained sorted order of keys makes sequential traversing easy and the B-tree fit for applications having access patters with reads and writes of large chunks of data with high locality. Accordingly the B-tree has commonly been used in general purpose databases and file systems.

2.1.2 B$^+$-tree

When it comes to implementing and using a dynamic multilevel index, such as the B-tree, most systems actually use a variation of the B-tree known as a B$^+$-tree [9, pp. 622–630]. This variation differentiates itself from the standard B-tree by using a different structure for internal nodes and leaf nodes.

In the B-tree every key occur once along with its associated value(s), whereas in the B$^+$-tree all keys and their value(s) are stored in the leaf nodes. In the B$^+$-tree internal nodes hold copies of the keys in the leafs to guide search in the same manner as the B-tree. Storing only keys in internal nodes has the benefit of reducing the I/O cost when operating on the index. This is because more of the nodes used for search can be present in main memory. Additionally the B$^+$-tree links the leaf nodes together, either as a doubly linked list or a singly linked list, to even more effectively support broad range queries. The average case insert, delete and search for the B$^+$-tree is the same as for the B-tree, namely $\mathcal{O}(\log n)$, and the average space complexity preserved alike, that is $\mathcal{O}(n)$. Figure 2.2 shows an example of the B$^+$-tree structure.

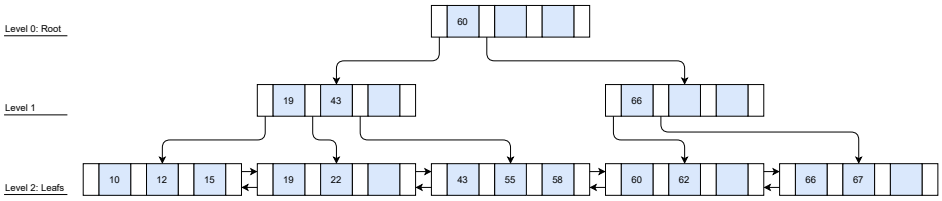


Figure 2.2: Example structure of a B$^+$-tree with order $m = 4$. In this case the leaf nodes are doubly linked.

2.1.3 Hash index

Hashing is a technique with many different areas of application. The concept of hashing bases itself on the usage of a hash function. That is, a function that takes input of arbitrary size and maps it to a domain of fixed-size values. In the case of an index we define the hash function, $h(k)$, where k ranges over the search key values to be indexed [9, pp. 633–634]. To create the index the data records are

iterated over and each record placed in the bucket given by the hash of its key. Buckets are made up of pages of fixed size, as such overflow chains are possible. This means that if a page becomes full, we link this page to a new page and continue inserting records at this location for a given hash mapping. The hash index is fairly lightweight since it only contains data pointers to storage locations for the hashes that it has digested during creation and subsequent inserts, in addition to the hash function itself.

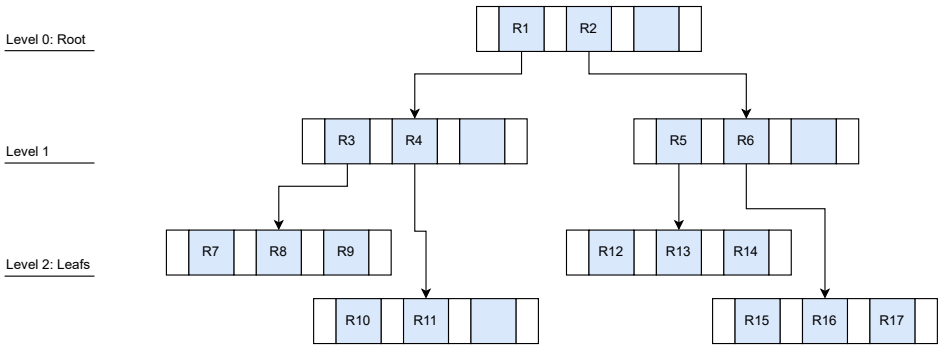
To retrieve a record given its key value k, we simply evaluate $h(k)$. $h(k)$ gives the location of the record's bucket. We retrieve the bucket and search through it until we find the record we are looking for. What should become apparent is that the hash index will perform best, in terms of I/O cost, for single key lookups in a hash index that is constructed on a key that gives limited amounts of overflow chains. With a proper hash function locality of data is not preserved making range queries highly inefficient. With regards to time complexity, insert, delete and search operates at the order of $\mathcal{O}(1)$ average case, and $\mathcal{O}(n)$ worst case. Space complexity is similar to that of the B-/B$^+$-tree, that is $\mathcal{O}(n)$.

2.1.4 R-tree

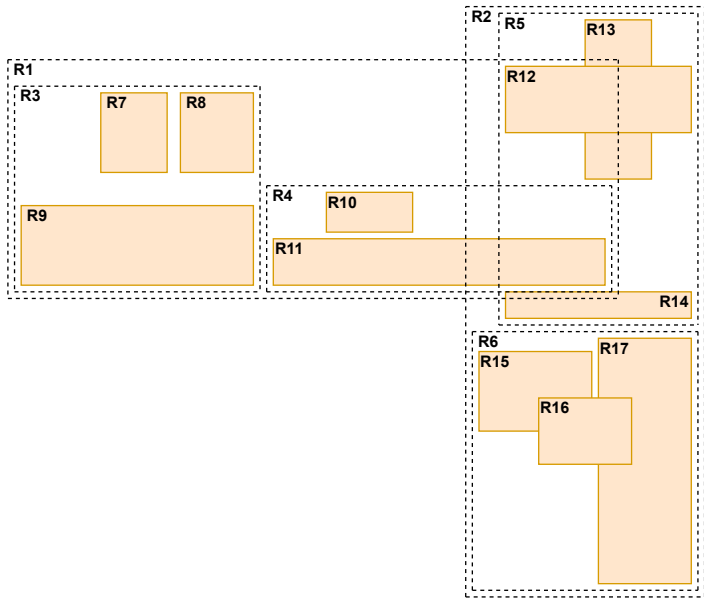
Multi-dimensional data brings about problems for well established index structures such as the B-tree. Geographical coordinates is one type of such data. A B-tree will not easily accommodate spatial access methods in the form of queries such as "Locate all hiking destinations that are between 5 and 10 km away from my current location", the R-tree [12], however, will.

The R-tree resembles the B-tree in many ways. Both the R-tree and B-tree are balanced search trees using pointers to navigate to the next level. The R-tree's structure however, consists of minimum bounding boxes (MBBs) which groups together nearby objects. In the case of two dimensions a MBB is called a minimum bounding rectangle (MBR). This is in fact where the tree gets its name from; "R" for rectangle. A MBR is defined by four coordinates: min_x, max_x, min_y, max_y.

The R-tree expresses the maximum extents of the objects that lies within it. Conversely this means the smallest rectangle that encompasses all required objects. It should be noted that the construction of an R-tree is non-deterministic, and its performance reliant on a good tree structure. The leaf nodes of the R-tree consists of MBBs containing single objects and a parent MBB encompasses all child MBBs. As such there is a guarantee that if a query does not intersect two MBBs, none of their contained objects can intersect. This is the property that makes the R-tree efficient when faced with queries concerning nearest neighbor searches, containment and intersection. An example R-tree is shown in Figure 2.3.



(a) R-tree.



(b) Rectangles.

Figure 2.3: An R-tree with order $m = 3$ constructed from 2D rectangles.

2.1.5 LSM-tree

Yet another tree-based index structure is the log-structured merge-tree (LSM-tree) [13]. It stores key-value pairs in a way that makes it excel under write-heavy workloads. The tree is structured into a set of components where one of these components is kept in-memory and all others on-disk. Append of new records are done to the in-memory component until it reaches its maximum size. Upon reaching maximum size the entries are flushed and merged into the next component through a rolling merge process. Updates can be done to the in-memory component in-place, whereas updated to on-disk components are performed out-

of-place. What makes the LSM-tree optimized for write-heavy workloads is the batching of records in main memory. When a certain batch is merged into the on-disk component, sequential writes are utilized to effectivly update the structure. The rolling merge process can be found illustrated in Figure 2.4.

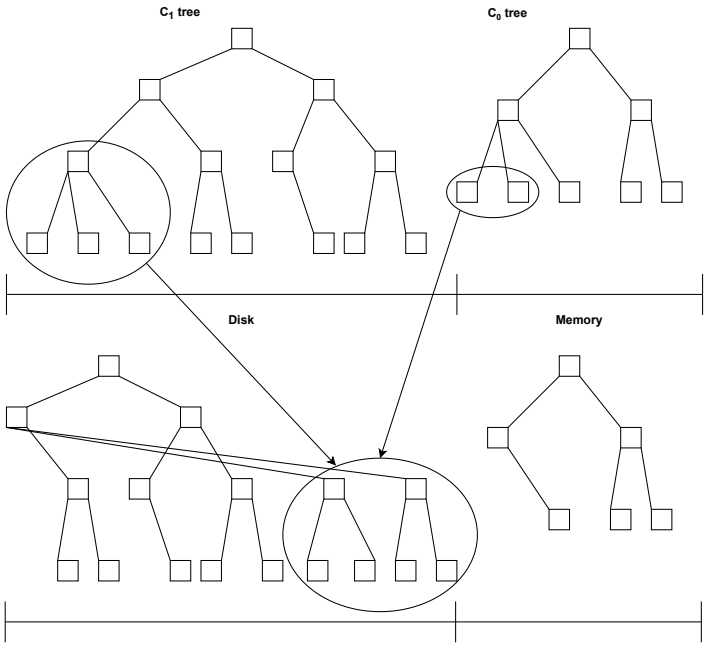


Figure 2.4: Depiction of the LSM-tree's rolling merge process. Figure made to look like the original presented in [13].

2.2 Parallel programming

This section describes the concept of parallel programming and takes a look at some important distinctions. Main sources for both subsection 2.2.2 and subsection 2.2.3 are [14] and [15].

2.2.1 The idea of parallel programming

The idea of parallel programming is in essence simple; distribute the workload among many workers, get the job done faster. The problems arise when trying to do this in an effective, maintainable and scalable way. Preventing simultaneous access to a shared resource, so-called mutual exclusion, becomes important. A program must be divided into independent task in a time-efficient manner and producers and consumers of such tasks must work together in harmony. The

saying “too many cooks spoil the broth” is often applicable in the context of parallel programming.

Although there are challenges to overcome when trying to parallelize a program, there are also rewards to be reaped by those who succeed. For example graphical processing units (GPUs) heavily rely on single instruction multiple data (SIMD) parallel processing (as well as single program multiple data (SPMD)) to quickly transform thousands or millions of verticies in the same way [16]. This has in turn made GPUs leading in applications such as video processing, artificial intelligence and gaming.

Also in the world of databases parallel programming has contributed to performance increase. An example is [17], which lists parallel execution as a way to optimize performance in an Oracle Database deployed on a system with all of the following characteristics:

- Symetric multiprocessors (SMPs), clusters, or massively parallel systems.
- Sufficient I/O bandwidth.
- Underutilized or intermittently used CPUs (for example, systems where CPU usage is typically less than 30 percent).
- Sufficient memory to support additional memory-intensive processes, such as sorts, hashing and I/O buffers.

When looking at index creation in major database systems (specifically Oracle Database and PostgreSQL), it holds true that there is support for index creation in parallel [18], [19]. However, none of the documentations listed make mention of general bottlenecks when creating indexes in parallel, or elaborates in extensive detail on inner workings of this operation.

2.2.2 Concurrency vs. parallelism

Concurrency and parallelism are terms decribing similar concepts, but they most certainly differ. Parallelism in the current context describes the process of running multiple copies of the same program simultaneously, but executing those programs on different data. For example, parallelism can be used to print the words of a document having two pages by starting two copies of the read-and-print program and giving each copy one of the pages. During execution the programs will not have to communicate with each other, but they are both running, executing the same task independently of each other, in parallel.

On the other hand we have concurrency. Concurrency in this context describes the process of running multiple copies of the same program simultaneously while communicating with each other. The executing programs will overlap in task execution and thus communication is neccessary. Using the same document printing example, we would be doing concurrent programming if we were to print the document by having each copy of the program start in either end of the document and print words until they met in the middle. The copies would print words at

the same time, but they would need access to a shared memory location as to know when to stop and not pass each other unknowingly. Figure 2.5 illustrates the difference.

With parallel programming the primary goal is most often to improve throughput through means of employing more workers, whereas concurrent programming is more concerned with handling the complexity of a potentially non-deterministic control flow. Concurrency is often simply viewed as running and completing tasks at the same time, but not necessarily running the tasks at the same instant. Concurrency in task execution can be achieved on a singlecore processor, but true parallelism requires a multicore processor.

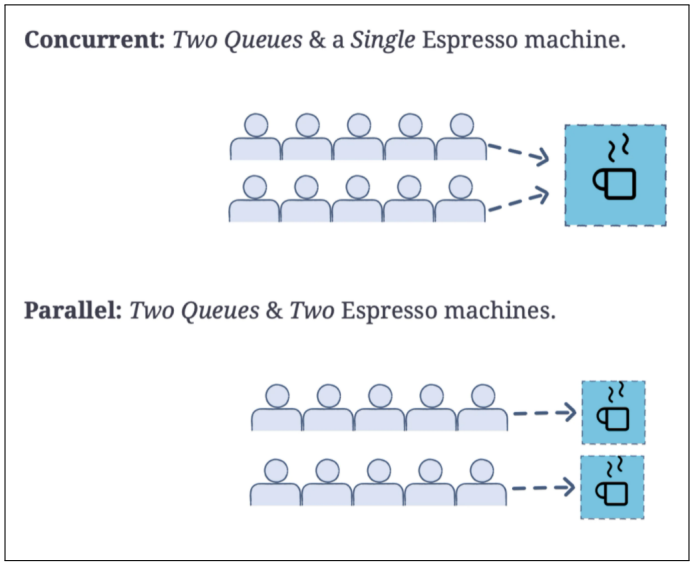


Figure 2.5: An illustration showing the difference between concurrent operation and parallel operation. Source: [14].

2.2.3 Multithreading vs. parallelism

In the same way that concurrency and parallelism often get mixed up, so do multithreading and parallelism. Multithreading is the technique that allows for concurrent execution of two or more parts of a program by effective utilization of the systems CPU(s). In other words, multithreading is the technique of dividing a program into threads. The threads might be run in parallel, but it is not neccessary. In certain programming scenarios it might make perfect sense to structure a program through means of creating threads. However, if one runs the program on a single core machine, then one is simply doing multithreading and not achieving parallelism.

2.2.4 Parallelism at different levels

Following up on the contents of subsection 2.2.3 is a brief summary of the mapping models that should be considered when creating a thread at the user-level. A programming language that has some form of a thread application programming interface (API), will at some point in time have to map those threads to kernel-level threads for execution [20, pp. 183–190]. Invoking a function at the kernel-level differentiates itself from the user-level in the way that an actual system call to the kernel will be issued upon invocation. The underlying operating system (OS) and hardware architecture will dictate this mapping. Its important to note that true parallelism is only achieved if the kernel utilizes multiple threads spread accross different cores during program execution.

The different mapping models between user- and kernel-level threads are illustrated in Figure 2.6. The alternatives are:

- *Many-to-one*: All user-level threads execute on one kernel-level thread.
- *One-to-one*: Each user-level thread execute on its own kernel-level thread.
- *Many-to-many*: For a given number, n, user-level threads, m kernel-level threads are allocated for execution of these.
- *Two-level*: Same as the many-to-many model, but certain user-level threads maps to single kernel-level threads, allowing for one-to-one execution in these instances.

As should be clear from examining these models is that all models are able to parallelize a single process's tasks, in the form of threads, on a multicore system architecture, except the many-to-one mapping model. In this model the kernel simply views the process to which the user-level threads belong as a single unit that occasionally makes system calls when the kernel-level thread is executed. Traditional UNIX implementation favors the mapping one-to-one as stated in [20, p. 188].

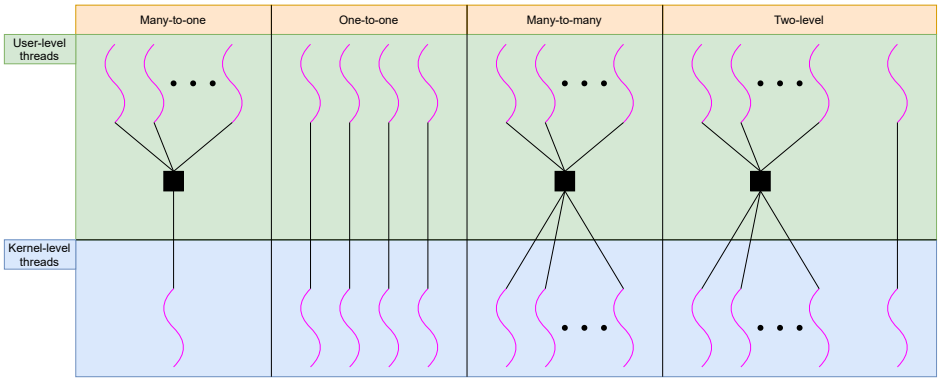


Figure 2.6: Different thread mapping models available between user- and kernel-level threads.

2.2.5 Speedup potential and fallpits

As one gains access to parallel processing resources, and starts to realise its potential, there are some fallpits and fundamental relationships one should be aware of. Amdahl's law [20, pp. 190–195] objectivly states that the speedup one can hope to achive when moving from a single core processor to a multicore processor and enabling parallelism, is dependent on the code that is inherently serial:

$$Speedup = \frac{time\ to\ execute\ program\ on\ a\ single\ processor}{time\ to\ execute\ program\ on\ N\ parallel\ processors} = \frac{1}{(1-f) + \frac{f}{N}}$$

Here f denotes the fraction of code that is infinitely parallelizable with no scheduling overhead. Thus, $(1 - f)$ denotes the inherently serial code fraction. Drawing a graph, as is done in Figure 2.7, illustrates that even a small portion of serial code will severly degrade the achivable performance. The law does not take into account additional communication, distribution and cache coherence costs, further degrading performance at a certain level of processors in use, if not carefully dealt with.

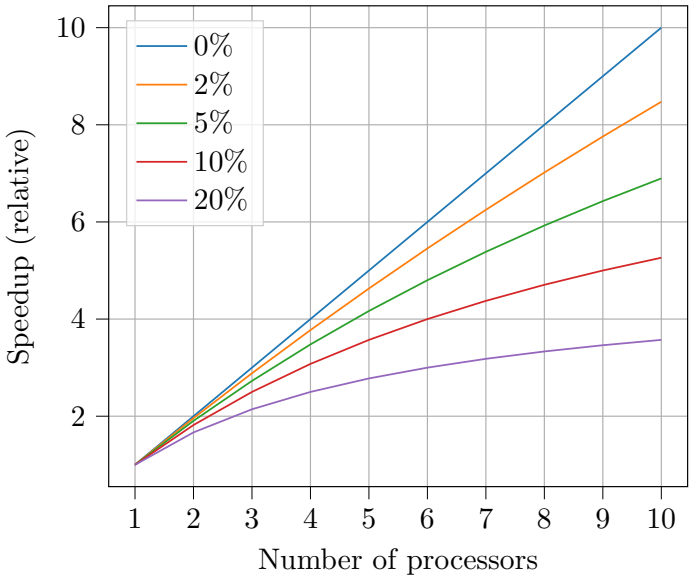


Figure 2.7: Amdahl's law.

The law on its own might give rise to some worries for those dealing with attempts at speedup by parallel execution. However, as with all findings and attempts at simplistically explaining reality, the law has undergone criticism. As eloquently pointed out in a blog post [21]; the applicability of Amdahl's law depends on how well the workload fits the programming model assumed by the law's model. How

the critical code sections are structured, and how the tasks are scheduled and executed on the architecture, will determine resulting performance even in presence of serial thread execution. For example in the producer-consumer scheme, a single producer will only become the cause of a bottleneck if it cannot feed the consumers quickly enough.

2.3 Concurrency mechanisms

In order to achieve concurrency in operation, and parallelism therein, it is common to make use of a handful of standardised cocurrency mechanisms. Such mechanisms could have hardware support, but this section describe the most common OS and programming language mechanisms, due to them being most relevant here. All the mechanisms to be described have in common that they can be used to achieve mutual exclusion and enable synchronization between threads. [20, pp. 244–270] is the main source used for the mechanisms described.

2.3.1 Semaphore

The semaphore was first introduced by Dijkstra in 1965 [22]. The idea is to use a integer value to communicate between processes or separate execution contexts. To do so, three atomic operations are permitted on the integer value: initialize, decrement and increment. Initializing the semaphore means setting the integer value to a value of 0 or more. The value chosen at this stage represents the number of threads allowed to immediatly continue after their decrement test have been issued on the semaphore. The decrement operation on its side is used to test entry to the guarded code section. If the integer value becomes negative by a thread issuing a decrement operation, the thread is blocked, otherwise the thread is allowed to continue on. The increment operation is the opposite of the decrement operation. When issued, the semaphore integer value is incremented, having the following consequence: if the resulting integer value is less than or equal to zero, then a thread that has been blocked by a decrement operation, if any, is unblocked.

The semaphore description given above is often called a counting semaphore. A more restrictive semaphore definition is that of the binary semaphore which makes the integer value only allowed to be 0 or 1. In any case, using a semaphore to guard resources means that the code section requiring access control is primitivly guarded without knowledge of which resources are available, only how many are free. Furthermore, semaphores can be classified as either strong or weak based on the policy used to select the next candidate from the blocked queue upon unblocking. If a policy (e.g. first-in-first-out (FIFO)) exist then the semaphore is strong. Otherwise, if a policy is not present, the semaphore is weak.

2.3.2 Mutex

The mutex, or mutual exclusion lock, is a similar concurrency mechanism to the binary semaphore. It distinguishes itself from the latter by requiring that the actor changing its value to 0 (locking the lock), must be the one to set its value back to 1 (unlocking the lock). In many programming languages implementation of the mutex, we find that the implementation contains two mutex variations; one allowing for unique access, and one allowing for shared access, to the resource they protect. This is in turn recognizable as the well-known access pattern scheme comprising use of unique writer locks and shared reader locks.

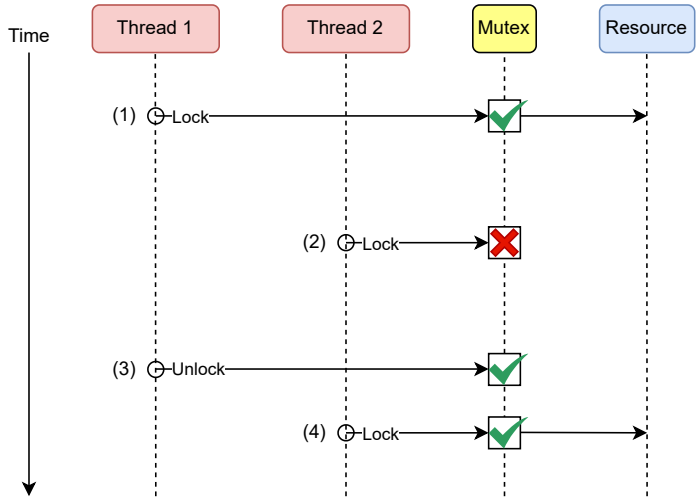


Figure 2.8: The mutex concurrency mechanism in action.

When a mutex blocks a thread from entering, the blocking mechanism is commonly implemented using the spinlock policy. With such policy, the blocked thread will execute a busy waiting loop that polls the availability of the lock. This means that resources are tied up in a busy wait state. Another option is to make use of a interrupt-based policy. Doing so requires an additional sleep-queue, but has the benefit of not constantly scheduling the thread for execution only to check state. Favorably, if blocking is known to last for a prolonged time period, the process managing the blocking thread should block the thread at the user-level as to not block the entire process.

Figure 2.8 shows how a mutex can be used to guard a shared resource. At (1) Thread 1 wants access to the shared resource. It attempts to lock the mutex. The operation is successful, thus granting Thread 1 access to the resource. Moving on, at (2) Thread 2 also wants access to the resource. Thread 2 does not know that Thread 1 is already accessing the resource when it issues its lock command on the mutex. The mutex rejects Thread 2, putting it to sleep if the blocking policy is interrupt-based. When Thread 1 finishes its work with the resource it issues

unlock, which can be seen at (3). Upon the mutex being unlocked by Thread 1, Thread 2 is awakened. Thread 2 immediatly issues the lock command and gains access to the resource. This is seen at (4).

2.3.3 Monitor

Even though the concurrency mechanism described thus far provide sufficient access control if utilized correctly, certain cases require better maintainability. Scattering increment and decrement operations throughout code, or locking without appropriate naming conventions, is likely to lead to code that is both hard to read and maintain. This is what motivated the monitor's creation. It was first introduced by Hoare in 1974 [23]. Hoare's definition of the monitor is the one to be described here, although it should be mentioned that alternate models of the monitor exist, e.g. Mesa monitor using notify and broadcast operations [24].

In essence the monitor is an all-encompassing structure that can be applied such that it locks entire objects at the programmers discretion. The structure is characterised by three properties, of which the first two are easily implementable in object-oriented programming languages:

- The monitor's procedures are the only procedures that can be used to gain access to the local data variables.
- Any process wishing to gain access to the "object" guarded by the monitor, enters the monitor via one of the monitor's procedures.
- The monitor should function such that only one process is allowed to execute inside it at any time.

Placing a shared resource within a monitor guarantees mutual exclusion if the monitor abides by these points. Additionally, synchronization mechanisms are provided such that blocking, waiting, and transfer of access can happen. To do so condition variables are in use. An overall view of the monitor's structure is shown in Figure 2.9. The `cwait` and `csignal` operations are the once used to manage the condition variables.

2.3.4 Message passing

Message passing is the last concurrency mechanism described in this section. In the current context, message passing is used to provide communication and synchronization between threads and/or processes. As a general technique, message passing can be found implemented in multiple different environments, e.g. distributed system as well as centralized systems having uniprocessor or multiprocessor architecture. There are many different configurations enabling message passing, but all of them, in some way or another, needs to define a pair of primitive functions enabling `send` and `receive`. Furthermore, the participants in the

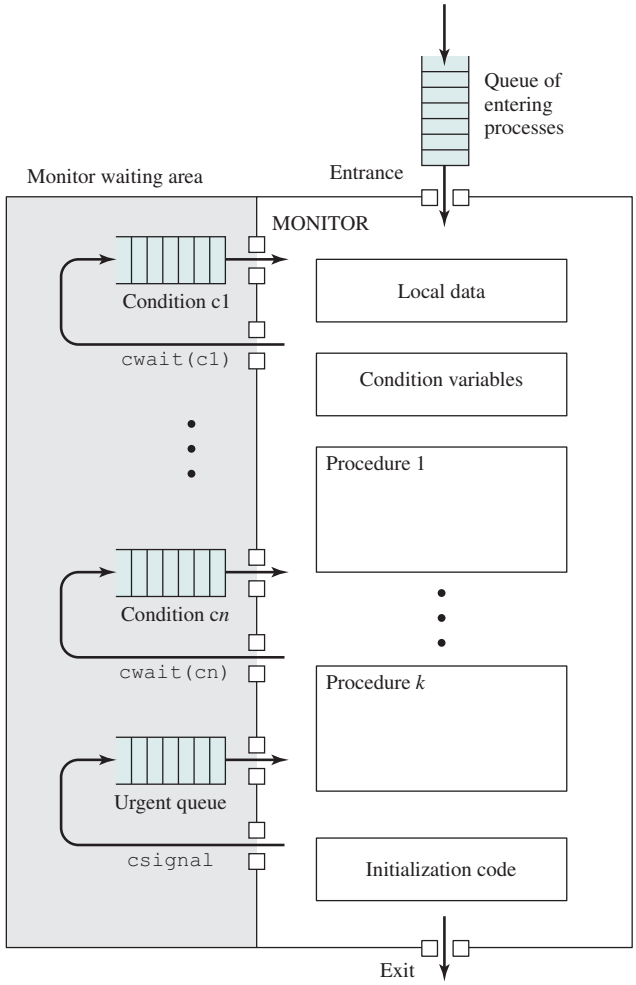


Figure 2.9: Structure of a monitor. Source: [20, p. 259].

message passing scheme has to agree upon a message format and the tasks to be coordinated. Since the entire concurrency mechanisms section mainly has concerned concurrency mechanisms with a focus on mutual exclusion, this is also the topic elaborated here, with message passing as the mechanism used to enforce it.

There are many ways to enforce mutual exclusion by use of message passing. One of the most basic ones, as described in [20, p. 268], uses a message box accessible by all participants. Figure 2.10 visualizes this mutual exclusion by message passing mechanism. The message box is initialized with a message functioning as a token that grants access to a protected code section. Only the process/thread which holds the message in its possesion can enter the protected code section.

Using blocking `receive` and nonblocking `send` on the message should make mutal exclusion enforceable. A thread wishing to enter the protected code section attempts to receive the message from the mailbox. If the mailbox is not empty, the message is delivered promptly, and thereby the thread granted access. If however, the mailbox is empty, the thread is blocked and must wait for the message to return to the mailbox. Upon exiting the critical section, the thread places the message back in the mailbox using the `send` operation. Correctly enforcing mutual exclusion this way presupposes that the message is delivered to only one thread, if requested by multiple `receive` operations concurrently. When the message has been delivered to one thread, all others should be blocked. Also, when the message returns to the mailbox and multiple contestants have previously been blocked, only one of them should be activated and given the message.

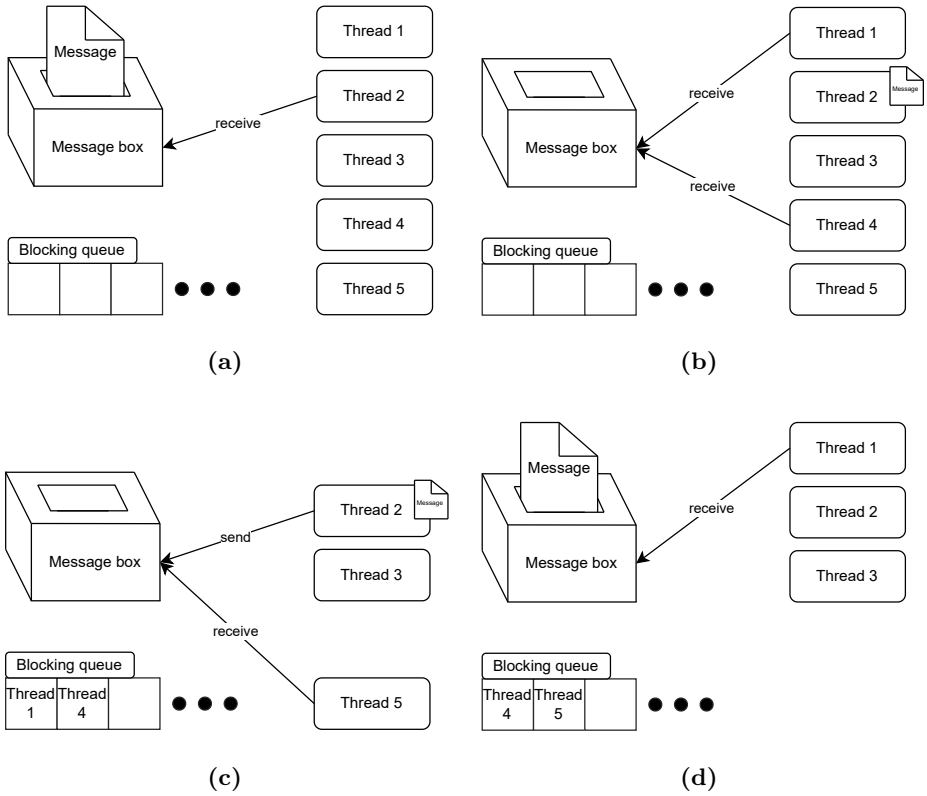


Figure 2.10: Message passing; passing a single message functioning as a token, in order to enforce mutual exclusion.

2.4 Related work

While a lot of core functionality of index structures are old, there have been many attempts at meeting shortcomings and adapting to a changing hardware environment. This section presents some notable related work that tackles such challenges.

2.4.1 PALM

PALM [1], is a technique that enables Parallel Architecture-Friendly Latch-Free Modifications to B^+ trees on many-core processors. What this means is that the technique aims to provide a scheme supporting multiple concurrent queries on in-memory B^+-trees. The paper presenting the technique argues that reliance on latches to avoid race-conditions and inconsistencies, in some cases forces serialization of queries, which in turn results in poor scaling. The authors continues on motivating their novel technique by pointing out how latch-based schemes can result in difficult to maintain, complicated code. PALM however, operates without use of latches in bulk synchronous fashion comprising four major stages. By processing queries in bulk, PALM remains latch free through ensuring that the tree is never modified until all searches have completed, and that a single node's modifications are the responsibility of only one thread.

Algorithm

A color coded visualization of the algorithm can be found in Figure 2.11. At a high level, the algorithm that processes batch queries are as follows:

- *Stage 1:*
  - Divide all queries among threads.
  - Locate the leaves reached by each query.
- *Stage 2:*
  - Redistribute work in order to eliminate any modification contention.
  - Ensure ordering of queries.
  - Carry out modification of leaves.
- *Stage 3:*
  - Move up the tree one step at a time, modifying internal nodes and redistributing work as necessesary, until root is reached.
- *Stage 4:*
  - Modify the root using a single thread if necessesary.

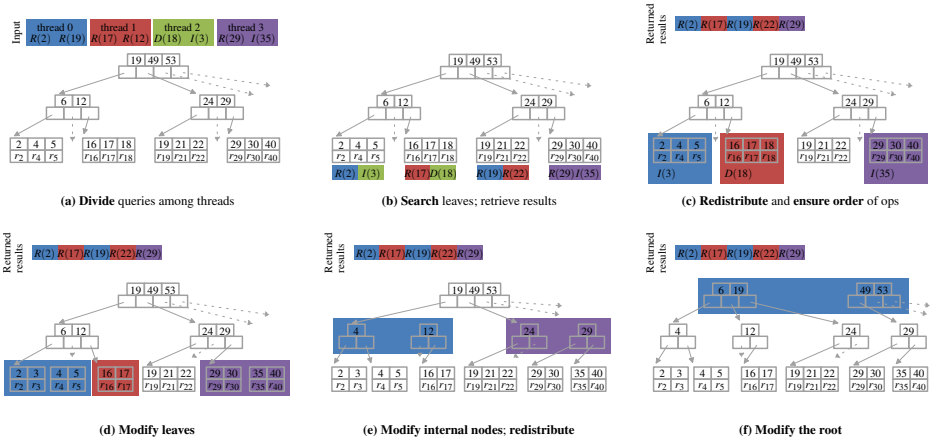


Figure 2.11: The PALM algorithm visualized. Source: [1].

Results

The paper reports the PALM scheme as having low response times when compared against buffering methods. Additionally, PALM performs 2.3×-19.1× faster than B-link trees [2] for a variety of configurations tested. Scalability, is also reportedly good with 10×-11.6× scaling reported at a range from 1 to 12 cores.

2.4.2 Bw-tree

The Bw-tree [3] was introduced in 2013 by researchers at Microsoft. The basis for its design is the recognition that multi-core CPUs mandate high concurrency, and that good multi-core processor performance depends on high cache hit ratios. Making the Bw-Tree latch-free and avoiding in-place updates deals with these issues. The paper implements the design introduced and evaluates it in relation to a pre-existing traditional B-tree architecture implemented in BerkeleyDB, and to a latch-free skip list. In the performance results section of the paper the Bw-tree is stated to achieve very high performance (backed by the researchers performance evaluation). Figure 2.12 illustrates the classical atomic record store architecture of the Bw-tree.

Compare and swap

In order to stay latch-free, state changes are carried out using the atomic compare and swap (CAS) instruction. Being able to avoid blocking in most cases avoids thread idle time and context switch costs. The only case where the Bw-tree blocks is when it needs to fetch a page from stable storage, which is rare when the system has a large main memory cache.

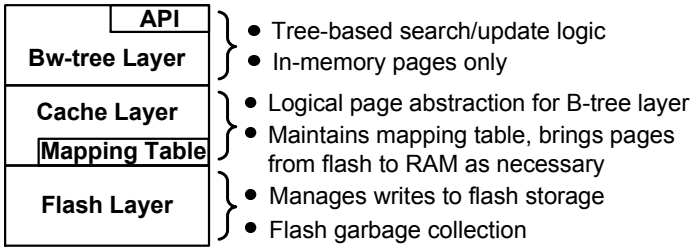


Figure 2.12: Architecture of the Bw-tree atomic record store. Source: [3].

An actual update on a node by the CAS instruction is called a delta update. It involves attaching the update to an exisiting page to reduce CPU cache invalidation.

Mapping table

The mapping table is what enables use of the CAS instruction during so-called delta updating. The mapping table maps logical pages to physical pages, i.e. it defines a mapping between either a page identifier and a flash offset, or the page identifier and a memory pointer. It severs the connection between physical location and inter-node links, having the result that delta updating can occur in main memory, whilst the physical location of a Bw-tree node can change every time a page is written to stable storage.

Open source version

The aforementioned Bw-tree's source code was not released, thus leading to other researchers taking it upon themselves to implement an open version of the design the researchers at Microsoft presented. In the paper "Building a Bw-Tree Takes More Than Just Buzz Words" [4], further improvements were made, and missing points in the Microsoft researchers original design document clarified. However, despite this open implementation being able to outperform the one presented by Microsoft, the researchers of this paper show that the Bw-tree still does not perform as well as other concurrent data structures that use locks.

2.4.3 Adaptive radix tree

One of the state-of-the-art high performance concurrent data structures compared against in the open Bw-tree paper [4], was the aptive radix tree (ART) [25]. ART is unsurprisingly a radix tree, also known as a space-optimized prefix tree. It is design for use in in-memory DBMSs. To improve cache efficiency the tree uses four different node layouts depending on the number of non-null child

pointers. The adaptive part of the radix tree comes from the structure adapting the reperesentation of each inner node locally using these layouts, wich optimizes global space utilization and access efficiency at the same time. An example of the adaptively sized nodes can be seen in Figure 2.13.

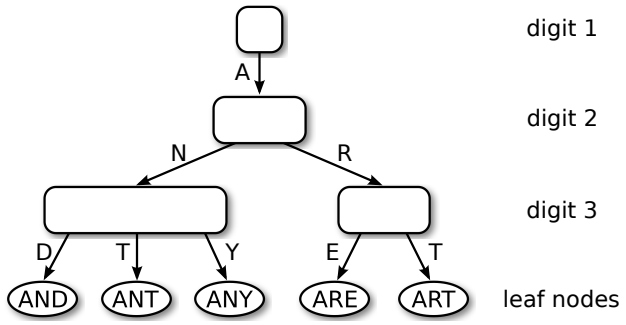


Figure 2.13: Adaptively sized nodes of the radix tree. Source: [25].

OLC and ROWEX

Although performance evaluation of ART show that it is both fast and space-efficient, it is often the case that lock-free data structures are difficult to implement and maintain. This motivated "The ART of practical synchronization" [26], which suggest a middle ground between scalable lock-free structures, and its difficulty, and fine-grained locking that does not scale particularly well on modern hardware, but has the benefit of consistency and ease of use. The paper does so through synchronizing ART with two different locking protocols (that use locking sparingly), namely optimistic lock coupling (OLC) and read-optimized write exclusion (ROWEX).

OLC is very simple, easily implementable and performs well when conflicts are not too frequent. On the other hand ROWEX is more complex, generally requiring changes to the data structure itself, but has the advantage that reads never block. Most importantly, both these locking mechanisms are not as complicated as truly lock-free mechanisms. Also, they perform consistent in that no indirection layer is present, something which may cause additional cache misses, as can be the case when using the Bw-tree.

Lastly, it is worth mentioning that the ART variant employing OLC as described by [26], was in fact among the state-of-the-art in-memory data structures (along with SkipList [27] and Masstree [5]) to beat the Bw-tree in performance on multi-core CPUs as found by [4]. Contrary to previous claims of lock-free indexes being superior to lock-based on such systems, the open Bw-tree paper [4] finds that the Bw-tree's indirection layer and delta records causes it to underperform the mentioned lock-based indexes by 1.5×-4.5×.

Chapter 3

Implementation

In this chapter the thesis' programming work and development is described. The main idea of the implementation is to approach the complex nature of parallel programming in a simplistic way. What this means is that the robust mechanisms of thread syncronization using locks is applied at a high level, and not highly micromanaged. This is done for two main reasons:

(1) Keeping the design simple allows development to progress rapidly.

(2) Trivially locking impose restriction on the degree of parallelism that can be obtained.

(1) is rather self-explanatory. (2) might not look favorable at first glance, but it has the implication that results obtained when comparing against a single threaded baseline are even more clear-cut if in favor of the parallel approach.

The two main components in the codebase is a fairly standard B^+-tree implementation, and a parallel B^+-tree implementation which uses the standard as part of its core. Both will be covered in detail during this chapter. The repository containing the codebase can be found at [28].

3.1 C++ programming language

The programming language chosen for the implementation was C++. C++ is a mature programming language that extends the C programming language first and foremost through use of classes. The language has a lengthy history and huge areas of application. Backwards compatibility has been of utmost importance to the language creators, resulting in a relatively huge standard compared to a lot of other programming languages. This is evidently apparent by the last freely available C++20 standard working draft [29], spanning 1857 PDF pages. The

language offers abstractions at the programmers discretion following a object-oriented programming paradigm, as well as low-level access if needed. Figure 3.1 briefly shows the main lines in development of C++ throughout the years, up until the start of 2020. So-called technical specifications (TSes) were first introduced in 2012 leading to a slightly different development process, whereby adjustments and modifications are allowed to occur within their own domain before being merged into the standard.

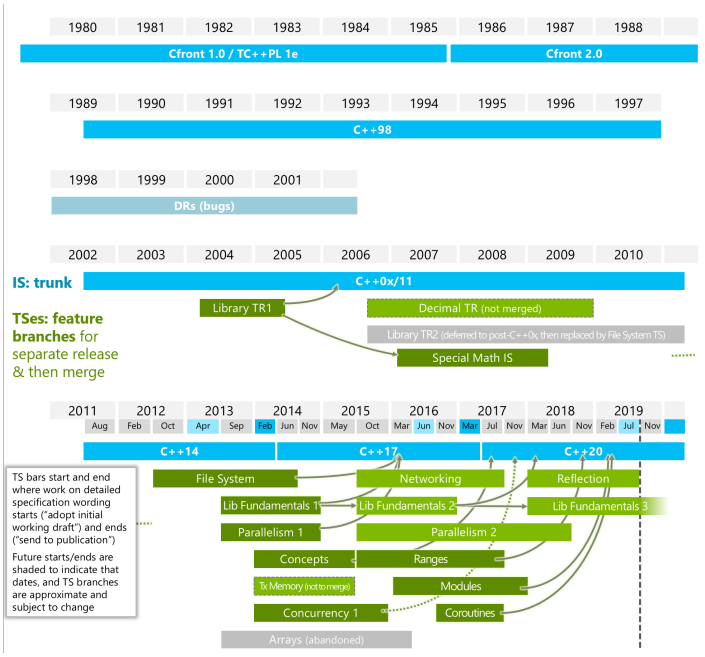


Figure 3.1: Timeline of C++ development and releases. Source [30].

3.1.1 Language of choice rational

The rational for choosing C++ is multifaceted. The language offers high performance if managed correctly. It does so by means of allowing for manual memory managment, and staying close to the hardware through CPU architecture dependent compilation. Lastly, the language also offers some comfort in its object-oriented approach which is familiar to the programmer.

Another compelling argument is that a related piece of work attempting to optimize the B^+-tree for the modern hardware architecture, in a somewhat similar fashion to what is done in this thesis, obtained unsatisfactory results due to a large portion of overhead being incurred by Java's garbage collector and the virtual machine's mapping of threads [31]. Among others, this body of work proposes using C/C++ to achieve better results.

3.1.2 Parallelism

The standard library of C++ offers a multitude of tools for working with concurrency and parallelism. Most importantly the `std::thread` [32] class allows for creating separate execution contexts that, when provided access to the OS' native threading API, can be executed in parallel as kernel-level threads if the underlying hardware architecture supports it. For example the Portable Operating System Interface (POSIX) Threads execution model, known as pthreads, defines a set of routines grouped by categories that does this. As it is a Unix family standard, documentation is readily available. For example the FreeBSD manual page can be found here [33].

Other than the bare-bones `std::thread` class. The C++ standard library also provides tools such as atomic types, condition variables, mutexes and futures [34]. All of which aims to prevent data races, synchronize memory access and alleviate the task of working asynchronously. There existis a host of locking wrapper classes and the like, leaving the level of detail managment very much up to the developer.

Of course thread creation incurs overhead. As such some time can be saved, in applications relying heavily on dispatching tasks via a multithreaded workflow, by creating a thread pool. That is, a design pattern that allows for task execution on the same set of threads, while only paying for the cost of creating the threads once. Although it is often common to want a thread pool, the C++ standard library does not explicitly declare this design pattern. The reason seemingly being that the type of work one might want to carry out can differ largely, and as such one agreed upon set of specifications for the thread pool are hard to establish. Additionaly, not providing tools for specific tasks at a certain high level of detail seems to fit with the C++ nature that cater to giving the programmer a lot of freedom. Nevertheless, a third-party C++ implementation was used in the programming work as detailed in subsection 3.2.3.

3.2 Components of the parallel B$^+$-tree

This section covers the individual central components making up the parallel B$^+$-tree. To aid the development process and keep focus on developing core functionality from scratch, i.e. the B$^+$-tree and parallel B$^+$-tree, two third-party open source libraries were used. The functionality and intended purpose of each library in the programming work of this thesis will be described here. Additionally, as is expected, the creators of the specific library implementations chosen are referenced. An architectural overview of the implemented parallel B$^+$-tree can be found in Figure 3.2.

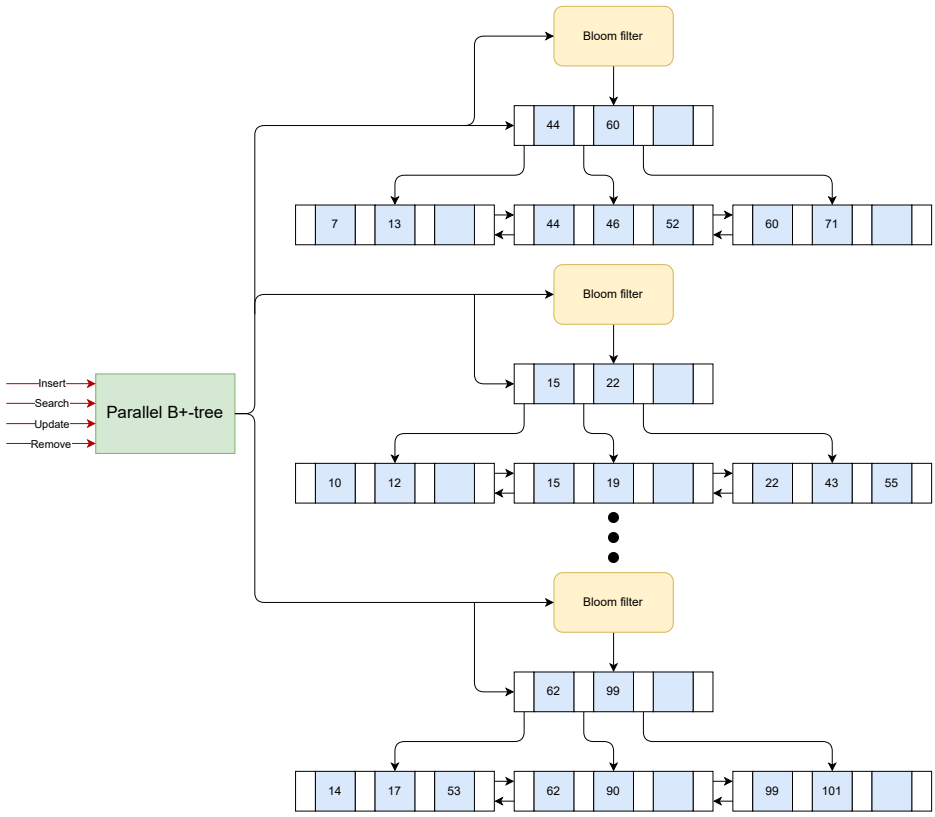


Figure 3.2: High level architectural overview of the parallel B^+-tree implementation.

3.2.1 B^+-tree

The generic B^+-tree has already been described in subsection 2.1.2. The implementation of this component was done from scratch in order to retain control over how operations were implemented. The implementation stays close to the nature of the aforementioned B^+-tree description, using [35, pp. 344–364] as a guideline when implementing the operations. In particular, when implementing `remove` (one of the more complex operations), the source used as a guideline showed itself greatly beneficial. The tree is a memory-resident, clustered index with doubly linked leaf nodes. Both keys and values are integers. A more rigourous block-and-posts format storing arbitrary object could be implemented later by using C++ templates. Also, to not bite off more than one can chew, no key compression or bulk-loading has been implemented as of now.

Having the B^+-tree be memory-resident simplifies the implementation as there is no need to deal with I/O management. Seeing as main memory have become larger and larger by time, in-memory databases have seen more and more use.

Thus, this simplifying choice might not be too far from the truth in certain use cases.

3.2.2 Bloom filter

Bloom filters are applied to each plain B$^+$-subtree of the parallel B$^+$-tree. They are intended to work as auxiliary structures enabling efficient reads, removes and updates by providing lookahead functionality for the keys in question.

A Bloom filter is a probabilistic data structure that tests whether a key is present in a tree by using k hash functions on the key. The hash funtions maps to a bit-array of m bits. If all the evaluated m bits are 1 we conclude that the element is present in the tree, and thus must be examined, otherwise the key is absolutely not in the tree. Figure 3.3 illustrates the data structure and its workings.

False positives are possible, but by doing a rough estimate on the amount of keys stored, the false positive probability can be controlled and made rather small. In fact, less that 10 bits per element are required for a 1% false positive probability, independent of the size or number of elements in the set [36]. Having a modifiable false positive probability is a nice property, but what is more important is that false negatives are not possible. Thus, consulting a tree's associated filter will in any case mark it as relevant if the key is present in the tree.

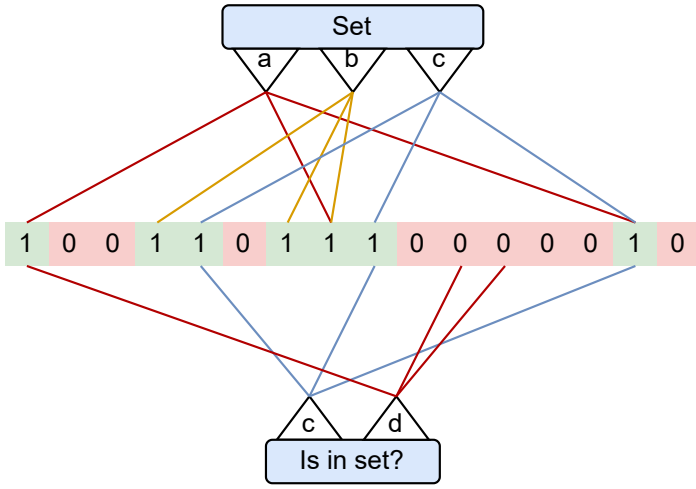


Figure 3.3: Bloom filter constructed on the set $\{a, b, c\}$ using $k = 3$ and $m = 16$. Querying the filter for c returns possible true because all bit-array positions are 1. Meanwhile, querying for d returns certain false since two bit-array positions are 0. Note that one bit-array position equal to 0 is enough to make sure the query argument is not a member of the set.

Usage of the Bloom filters are an experimental addition to the design and can

be enabled or disable upon parallel B$^+$-tree creation. As is known from the standard LSM-tree's reliance on this structure as well; operations that can or will remove keys from a certain tree (i.e. update and remove) will asymptotically degrade the performance of the filters [37]. This is due to not being able to remove an already hashed key from a filter, because the filter does not contain explicit information about all keys hashed. In turn this makes operations of such nature "second-class citizens" when enabled. This is an interesting aspect that is explored and discussed in more elaborate detail in the the performance evaluations of chapter 4.

The Bloom filter implementation used as a component of this thesis' parallel B$^+$-tree, is the C++ Bloom Filter Library created by Arash Partow [38]. It is a single header implementation with no external dependencies. A number of parameters can be passed during construction, most notably `projected_element_count` and `false_positive_probability`, which in turn makes finding a number of hash functions yielding the minimum amount of storage bits required to remain consistent with the user's arguments possible.

3.2.3 Thread pool

The thread pool is the workhorse of the parallel B$^+$-tree. It is intended to make working with asynchronus task execution easier, and reduce the amount of overhead required to dispatch tasks to the threads. As aforementioned, one of the main selling points of the thread pool's design pattern is avoiding the thread creation cost. A generic thread pool only composes a few central components as illustrated in Figure 3.4. The task submitters pushes tasks into a task queue, the threads in the thread pool pops tasks from said queue, executes them and then returns to the thread pool becoming available for new task assignments. Tasks submitted by the task submitters might be so-called fire-and-forget, meaning that they have no return value of relevance to the submitter, or they could have a return value that the task submitter can obtain from a shared state.

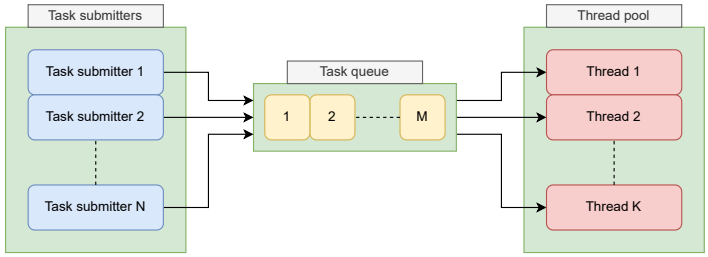


Figure 3.4: The main components of a thread pool design pattern.

In the parallel B$^+$-tree the main executing thread responsible for the parallel B$^+$-tree object is the only task submitter. Tasks are submitted to the task queue with required arguments for method execution, always working on the basis of an appointed B$^+$-subtree. The threads in the thread pool lock the tree which

they access during operation using appropriate read and write locks in the form of `std::unique_lock` [39] and `std::shared_lock` [40], respectivly. The mutex ownership wrappers are applied over the `std::shared_mutex` [41] associated with the tree, ensuring that only one thread at a time is allowed to modify a tree, but multiple threads can read from the tree if no unique write lock has been applied to the mutex.

The C++ thread pool implementation used for this thesis' purposes is written by Barak Shoshany [42]. Much like the Bloom filter implementation chosen, this implementation is also written as a self-contained class with no external dependencies. Upon construction of a new thread pool the number of worker threads can be specified. As is expected, providing a higher number than the number of available kernel-level threads is undesirable. There are two methods for submitting tasks to the thread pool: `submit` and `push_task`. Using the first returns a `std::future` [43] on the return value of the task submitted. If there is no return value for the task submitted, the `std::future` encapsulate a simple boolean data type which resolves to true when the task is completed. The latter method avoids the overhead of generating a `std::future` and is practical for fire-and-forget tasks. The last method from the thread pool class used in this thesis' programming work is the `wait_for_tasks` method, which blocks the caller until all tasks in the task queue have been completed.

3.3 Parallel B$^+$-tree operations

The methods implemented and evaluated on the `ParallelBplustree` class are `insert`, `search`, `update` and `remove` (delete is a reserved C++ keyword, hence remove). To look at how task creation cost affects performance, all operations have two implementations. That is, one implementation takes a single key (and possibly a single value: `insert` and `update`), and one takes a vector of keys (and possibly a vector of values: same operations as mentioned in the single case). This section describes the main lines and central ideas involved in implementing the operations through words an code snippets. But, before that, in order to understand how the `ParallelBplustree` works, a brief description of the parameters passed to the constructor during initialization of a new `ParallelBplustree` is given:

- `order`: Order of tree i.e. order of all managed B$^+$-trees
- `numThreads`: Number of threads to use in the thread pool.
- `numTrees`: Number of B$^+$-trees to create and use.
- `useBloomFilters`: Boolean determining if Bloom filters should be applied at a per B$^+$-tree basis.

3.3.1 Single key operations

Single key operations as denoted here refers to the implemented versions of `insert`, `search`, `update` and `remove` that operates on a single key at a time i.e. they are all callable with a `const int key` argument. All such single key operations, except `insert`, are from the perspective of the main executing thread, designed and implemented with the idea of quickly passing a coordination-task to the thread pool. The idea being that blocking the main executing thread should be the callers responsibility, which in turn should yield high throughput if creating a coordination-task can be done cheaply compared to the desired operation. Once the coordination-task has been popped from the thread pool's task queue, more tasks are created by the thread assigned the coordination-task. This thread is also responsible for relaying information back to caller of the operation's entry point in scenarios requiring it (e.g. `search`). The reason as to why there is no associated coordination-task during `insert`, has to do with `insert` being the only operation where it is entirely certain that only one tree must be accessed.

Insert

The single key `insert` operation naturally takes a key and value to be inserted into the data structure. This operation, when packed and dispatched as a `threadInsert` task, firstly checks if Bloom filters are used by the instance. If not, then one of the B$^+$-trees is chosen (pseudo-)randomly from a uniform distribution, a write lock on the tree is acquired, and the key-value pair inserted. If however, Bloom filters are used, the filters are consulted first. This is done in an attempt to preserve locality of data, i.e. store a key and all its associated values in one tree. If the key is found in a Bloom filter, the associated tree is where the key-value pair is inserted after aquiring a write lock. Otherwise, insert is done in the same manner as for when Bloom filters are not in use. Listing 3.1 shows a small excerpt from the `threadInsert` method when `useBloomFilters` is false. The filters are of course also updated during insert using the same locking mechanism as for the trees.

```
const int treeIndex = distr(gen);
std::unique_lock<std::shared_mutex> treeWriteLock(*treeLocks[
    ↪ treeIndex]);
trees[treeIndex]->insert(key, value)};
```

Listing 3.1: Insert of key-value pair in `threadInsert` task of parallel B$^+$-tree `insert` operation when `useBloomFilters` is false.

Search

The `search` operation takes a `const int key` argument, and returns the hefty data type `std::future<std::vector<std::future<const std::vector<int> *>>>`. That is, a value that can be waited on by the receiver. A pointer to the associated `std::promise` [44] of the `std::future` return value, is passed along with the key to the `threadSearchCoordinator` task, before the `search` operation returns. In the coordination-task, relevant B$^+$-trees that need searching are found, and tasks for searching them created. Values of type `std::future` for each search-task are pushed to a result vector. This result vector is used to set the value of the promise, making the intermediate vector-result available to the caller of `search` through its future's shared state. The caller can then wait on each individual tree search if wanted.

The idea is once again that having a quick return from the main entry point of `search`, should allow the caller to queue multiple searches quickly. Then its up to the parallel processing capabilities to finish the workload in a timely manner. Meanwhile, the caller can do other stuff or wait on all task to complete using, for example, the `waitForWorkToFinish` method on `ParallelBplustree`. The `search` operation described can be seen in its entirety in Listing 3.2. The promise created in the method is evidently allocated using heap memory. This memory is freed at the end of the coordination-task without disturbing the return value of `search`, since a underlying shared state exists.

```
std::future<std::vector<std::future<const std::vector<int> *>>>
    ↪ ParallelBplustree::search (const int key) {
  std::promise<std::vector<std::future<const std::vector<int> *>>>
      ↪ *prom = new std::promise<std::vector<std::future<const
      ↪ std::vector<int> *>>>;
  std::future<std::vector<std::future<const std::vector<int> *>>>
      ↪ fut = prom->get_future();
  threadPool.push_task([=, this]) () mutable {
      ↪ threadSearchCoordinator(key, prom); });
  return fut;
}
```

Listing 3.2: Single key `search` method as implemented in `ParallelBplustree`.

Update

Of the single key operations, the `update` operation involves the most complex coordination-task of the lot. In order to save some time in certain cases and make the `threadUpdateCoordinator` method not to long, all calls to `update` will insert the key with the values provided, if the key is not present in the tree at time of

update. This should not limit the possibilites from an end-user's perspective, as it can always be verified beforehand if a key is present in the tree by using the `search` operation.

The `update` operation should, along with a `const int key` argument, be provided with a argument that is a reference to a vector of values. Executing a update-coordination-task means finding a suitable B$^+$-tree for the update (or insert as previously mentioned) and removing all stale-to-be key-value pairs that might reside in other B$^+$-trees. Hopefully, when using Bloom filters, only one update will be issued and no removes. On the other hand, having disabled Bloom filters, will always result in one B$^+$-tree receiving the update and all other trees getting a remove call on the key. Listing 3.3 shows the gist of updating by a code snippet from `threadUpdateCoordinator` when `useBloomFilters` is false. Write locks are used and acquired by the threads carrying out `threadUpdate` and `threadRemove` tasks before accessing the B$^+$-tree on which to perform said operation.

```
int treeToUpdateOrInsert = distr(gen);
threadPool.push_task([=, &values, this] { threadUpdate(key, values,
    ↪ treeToUpdateOrInsert); });
for (int i = 0; i < numTrees; i++) {
  if (i != treeToUpdateOrInsert) {
    threadPool.push_task([=, this] { threadRemove(key, i); });
  }
}
```

Listing 3.3: Task of `threadUpdateCoordinator` during single key `update` when `useBloomFilters` is false.

It should be clear that insert-if-not-found during `update` is beneficial since then the `threadUpdateCoordinator` does not have to issue search tasks on all trees, block and wait, and then reconciling trees by issuing `threadUpdate` and `threadRemove` operations. Instead the `threadUpdateCoordinator` never blocks. However, an implementational implication of doing it like this is that the actual `threadUpdate` operation assumes update, and thus does not update the associated Bloom filter if insert occurs. Some steps could be taken to mitigate this, for example changing and using the return type of `update` on the `Bplustree` class, but as of now this has not been done.

Remove

Removing a key from the parallel B$^+$-tree is in many ways the inverse operation of `insert`. If Bloom filters are in use, the coordination-task created by `remove` will consult the Bloom filters to identify B$^+$-trees where the key to remove might reside. Since false negatives are not a possibility, all relevant trees will be operated on with a `threadRemove` task. If however, Bloom filters are not in use, a

`threadRemove` task must be issued for every B$^+$-tree of the parallel tree. The specific `remove` implementation uses the same promise and future return mechanism as described during `search`. This allows for nuanced waiting on the single key `remove` operation at the callers discretion. Listing 3.4 shows a snippet from the `threadRemoveCoordinator` method when `useBloomFilters` is true. The vector named `result` shown in this snippet is later moved and set as the value of the promise having a shared state with the callers return value.

```
for (int i = 0; i < numTrees; i++) {
  std::shared_lock<std::shared_mutex> treeFilterReadLock(*
      ↪ treeFilterLocks[i]);
  if (treeFilters[i]->contains(key)) {
    treeFilterReadLock.unlock();
    result.push_back(
      threadPool.submit(
        [=, this] { return threadRemove(key, i); }
      )
    );
  }
}
```

Listing 3.4: Task of `threadRemoveCoordinator` during single key `remove` when `useBloomFilters` is true.

3.3.2 Batch operations

The batch operations are the counterpart versions of the single key operations found in subsection 3.3.1. They extend parallel processing applicability by carrying out multiple simple operations at once when called upon. These operations came to be after doing some initial single key operation tests during development. Those tests seemed to indicated that in order to leverage parallel processing capabilities meaningfully, the workload of tasks created by the thread pool design pattern has to be costly to the degree that the cost of task creation becomes negligible. Only relying on batch operations in a real system could be regarded as lazy evaluation, whereby the processing resources should be effectively utilized when requested.

For the sake of clarity; the batch operations are `insert`, `search`, `update` and `remove`. What all methods have in common is that the main executing thread carries out the equivalent coordination-task discussed during single key operations. Additionally all methods except `search` are void, i.e. they have no return value. The methods partition the batch workload according to the number of threads (`numThreads`) or number of trees (`numTrees`) used by the parallel B$^+$-tree instance, depending on the value of `useBloomFilters`.

Insert

The batch `insert` operation takes a reference to a vector of keys, and a reference to a vector of the keys associated values. If Bloom filters are used, the workload is split by the number of threads in an attempt to utilize all parallel processing capability. The difference from the single key operation then manifests itself mainly through creation of a (likely) severly reduced number of thread pool tasks, but uses the same `threadInsert` backbone as the comparable single key `insert` operation. There is however a `threadInsert` wrapper method that gets provided with iterators, which in turn are pointers to appropriate partition start- and endpoints of the reference arguments. This is done to cut down on argument copies compared to the single key `insert` operation that packages each key-value pair separately in a `threadInsert` task.

When Bloom filters are not used the batch `insert` operation should be able to perform really well since the batch can simply be split by the number of trees. One thread accessing one tree allows for acquiring the write lock once before the actual `threadInsert` of the batch begins. This lock can be held until the operation finishes. The one obvious drawback is the general "not using Bloom filters"-drawback, i.e. not having locality in data means all other operations becomes more demanding in the parallel B^+-tree. Listing 3.5 shows the `threadInsert` wrapper method.

Search

For the batch `search` operation the variable to divide the workload by is always the number of trees, i.e. one task is pushed to the thread pool for each B^+-tree managed. When Bloom filters are in use, the main executing thread checks for each key in the keys vector reference provided; which trees it might belong to. Any relevant trees are marked for search on the key in question. After looping through all keys the `threadSearch` tasks are pushed to the thread pool and a result vector returned to the caller. This result vector will, when the pool finishes all tasks, contain all search results using a mapping of equivalence between a search key index in the keys vector and the result vector index. Not using Bloom filters means that the batch `search` operation will immediately create a `threadSearch` task for each tree on all keys.

The `threadSearch` method packaged as tasks during batch `search` can be seen in Listing 3.6. As shown by the code snippet, read locking is only applied once before all searches on a tree. Additionally, the result of a search is assigned directly in the result vector previously returned to the caller of the batch operation. This last point is worth elaborating on since, in general, thread safety cannot be guaranteed working with a vector. Looking at the code snippet it should be apparent that values have to already exist in the result vector to avoid a segmentation fault at runtime. This is taken care of at the start of the batch `search` operation where result is initialized with a allocator providing `nullptr` values on each potential B^+-tree search result. This way, locking on indexes in the result vector (which

```
void ParallelBplustree::threadInsert(
  std::vector<int>::iterator keysSplitBegin,
  std::vector<int>::iterator keysSplitEnd,
  std::vector<int>::iterator valuesSplitBegin,
  const int treeIndex
) {
  if (treeIndex > -1) {
    std::unique_lock<std::shared_mutex> treeWriteLock(*treeLocks[
        ↪ treeIndex]);
    for (
      std::vector<int>::iterator keysSplitIt = keysSplitBegin;
      keysSplitIt != keysSplitEnd;
      keysSplitIt++, valuesSplitBegin++
    ) {
      trees[treeIndex]->insert(*keysSplitIt, *valuesSplitBegin);
    }
  }
  else {
    for (
      std::vector<int>::iterator keysSplitIt = keysSplitBegin;
      keysSplitIt != keysSplitEnd;
      keysSplitIt++, valuesSplitBegin++
    ) {
      threadInsert(*keysSplitIt, *valuesSplitBegin);
    }
  }
}
```

Listing 3.5: The `threadInsert` wrapper method pushed as partitioned tasks to the thread pool during batch `insert`. Note the acquisition of a write lock once for all of a partition's inserts when `useBloomFilters` is false i.e. `treeIndex` > -1.

would need to be done if empty initializing and using the vector container's `push_back` method) becomes unnecessary, since at any time the only thread to access a B$^+$-tree specific search result is the one thread tasked with the `threadSearch` method on that tree.

Update

Batch updating follows the natural design pattern derived during implementation of the aforementioned batch operations. The main executing thread when calling the `update` operation starts of by finding a suitable B$^+$-tree to receive each update. Then, potential B$^+$-trees that require remove on each of the keys

```
void ParallelBplustree::threadSearch(
  const std::vector<int> *batchKeys,
  const int treeIndex,
  std::vector<std::vector<const std::vector<int> *>> &result,
  const std::vector<int> keysPos
) {
  if (useBloomFilters) {
    if (keysPos.size() > 0) {
      std::shared_lock<std::shared_mutex> treeReadLock(*treeLocks[
          ↪ treeIndex]);
      for (int i = 0; i < keysPos.size(); i++) {
        result[keysPos[i]][treeIndex] = trees[treeIndex]->search((*
            ↪ batchKeys)[keysPos[i]]);
      }
    }
  }
  else {
    std::shared_lock<std::shared_mutex> treeReadLock(*treeLocks[
        ↪ treeIndex]);
    for (int i = 0; i < batchKeys->size(); i++) {
      result[i][treeIndex] = trees[treeIndex]->search((*batchKeys)[
          ↪ i]);
    }
  }
}
```

Listing 3.6: The `threadSearch` method pushed as partitioned tasks to the thread pool during batch `search`.

to receive updates are identified and marked. Using Bloom filters should cut down on the number of single operations needed in the end. If these filters are not used the update workload is distributed evenly accross trees, and each update will lead to `numTrees - 1` removes. For each tree a task on the method `threadUpdateThenDelete` is pushed to the thread pool. All updates will precede the removes on a tree, and write locking is applied once before all updates and removes on a tree are carried out. The same insert-if-not-found mechanism discussed during the single key update version is also used here. Listing 3.7 shows the `threadUpdateThenDelete` method.

Remove

The batch `remove` operation passes a pointer to the keys vector it got passed by reference, to the `threadRemove` task, for each tree when Bloom filters are not in use. Otherwise, similar to other operations the Bloom filters are first tested

```
void ParallelBplustree::threadUpdateThenDelete(
  std::vector<int> updateKeys,
  std::vector<int> updateIndexOfValues,
  const std::vector<std::vector<int>> *updateBatchValues,
  std::vector<int> deleteKeys,
  const int treeIndex
) {
  std::unique_lock<std::shared_mutex> treeWriteLock(*treeLocks[
      ↪ treeIndex]);
  for (int i = 0; i < updateKeys.size(); i++) {
    trees[treeIndex]->update(updateKeys[i], (*updateBatchValues)[
        ↪ updateIndexOfValues[i]], true);
  }
  for (int i = 0; i < deleteKeys.size(); i++) {
    trees[treeIndex]->remove(deleteKeys[i]);
  }
}
```

Listing 3.7: The `threadUpdateThenDelete` method pushed as partitioned tasks to the thread pool during batch `update`. All taken by value vector parameters are provided as move-constructed arguments during task creation in `update` to avoid unnecessary copies.

for each key, resulting in a mapping between trees and the keys that needs be removed from them. These arguments are then provided to the `threadRemove` tasks. Write locking is in any case only done once for each tree operated on as show in Listing 3.8.

```
void ParallelBplustree::threadRemove(std::vector<int> keys, const
    ↪ int treeIndex) {
  std::unique_lock<std::shared_mutex> treeWriteLock(*treeLocks[
      ↪ treeIndex]);
  for (int key : keys) {
    trees[treeIndex]->remove(key);
  }
}
```

Listing 3.8: The `threadRemove` method pushed as partitioned tasks to the thread pool during batch `remove` when `useBloomFilters` is true.

Chapter 4

Results and Discussion

This chapter presents the performance results obtained for all operations found in section 3.3. There exists a number of variables that can be tuned to create different configurations, which of course will generate different results. The results presented here are those best belived to showcase strengths and weaknesses of the implementation. Reasoning for choice of specific variable values will be given in section 4.2 and on presentation of the results. All parallel operations are compared and illustrated against their equivalent single threaded B^+-tree implementation. The B^+-tree implementation will be referred to as baseline. The main performance evaluator is throughput measured in operations per second.

4.1 Setup

All tests and performance evaluations were executed on a computer with the specifications show in Listing 4.1. The codebase was compiled with the following compiler and options:

- g++ (Ubuntu 9.3.0-17ubuntu1 20.04) 9.3.0
- -std=c++2a
- -O3
- -pthread

Using an Intel processor, the computers supports hyper-threading which is Intel's proprietary version of simultaneous multithreading [45]. With hyper-threading enabled the CPU exposes two execution contexts per physical core, i.e. threads. As stated by the aforementioned Intel source: "[...] Intel Hyper-Threading Technology improves CPU throughput (by up to 30% in server applications)". Accord-

ingly, as can also be understood from the output of `lscpu` seen in Listing 4.1, the computer used for testing supports at most 32 kernel-level threads.

```
$ cat /var/run/motd.dynamics
[...]  Ubuntu 20.04.3 LTS (GNU/Linux 5.4.0-96-generic x86_64) [...]
$ lscpu
Architecture:            x86_64
CPU op-mode(s):          32-bit, 64-bit
Byte Order:              Little Endian
Address sizes:           46 bits physical, 48 bits virtual
CPU(s):                  32
On-line CPU(s) list:     0-31
Thread(s) per core:      2
Core(s) per socket:      8
Socket(s):               2
NUMA node(s):            2
Vendor ID:               GenuineIntel
CPU family:              6
Model:                   63
Model name:              Intel(R) Xeon(R) CPU E5-2640 v3 @ 2.60GHz
Stepping:                2
CPU MHz:                 1199.204
CPU max MHz:             3400.0000
CPU min MHz:             1200.0000
BogoMIPS:                5194.34
Virtualization:          VT-x
L1d cache:               512 KiB
L1i cache:               512 KiB
L2 cache:                4 MiB
L3 cache:                40 MiB
$ free -h --si
               total        used        free      shared  buff/cache   available
Mem:           128G        5.1G        117G        13M         6.4G         122G
Swap:          130G          0B        130G
```

Listing 4.1: Specifications of the computer used for testing and evaluating performance.

4.2 Commonalities

The interface for running the tests and controlling variable values is a command line program. Listing A.1 found in appendix A shows all the available flags and options.

The variables, whose impact on the two operation modes are studied, are:

- `order`
- `numThreads`
- `numTrees`

All bar charts that present results examining how the value of `order` impacts an operation's throughput, uses `numThreads` = `numTrees` = `std::thread::hardware_concurrency()` = 32. A starting point is needed, and the reason for chosing these values are as follows: since the main executing thread blocks, it can probably contribute in the thread pool while suspended. Accordingly, `numThreads` = `std::thread::hardware_concurrency()`. If parallel execution was perfectly timed, locking would become obsolete and only one thread would modify a critical structure at any one point in time. As such, `numTrees` >= `numThreads` if all threads are to be used all the time, when locking on trees.

Moving on, the results obtained when examining `order` are used to determine its value for each operation when examining the other variables. The same way, the results obtained when examining `numThreads`, in conjunction with those from observing `order`, are used to determine its value for runs observing `numTrees`. The values chosen are, for each operation, given at the start of the section linked to the bar charts, and reasoned in their appropriate subsections. This methodology implies that the culmination of the parallel implementations' throughput performance, yields highest measurements at the last studied variable, namely `numTrees`. Regarding the bar charts, it should be mentioned that they are all presented using a logarithmic y-axis.

To hone in on the specific variables and study the central aspects of the implementations, some command line arguments have been set to the same appropriate values throughout testing. These are:

- `--op`/`--tree-size` 5000000
- `--op-distr-high`/`--build-distr-high` 5000000
- `--op-distr-low`/`--build-distr-low` 1 (default i.e. not explicitly provided)

The value provided to `--op` will ensure that 5000000 operations are performed for the test carried out, whereas the value provided to `--tree-size`, ensures that the tree to perform the test on has received 5000000 inserts during tree build, if the test is not to evaluate the performance of `insert`. The value of 5000000 is sort of arbitrarily chosen, but it strikes a nice balance between running time and sufficent workload.

All keys and values are, for both build and test, drawn from a uniform integer distribution provided with a Mersenne Twister pseudo-random generator having a state size of 19937 bits. In use this is the `std::uniform_int_distribution` [46] provided with the `std::mersenne_twister_engine` [47]. Since the distribution is uniform and the keys drawn pseudo-randomly, all tests end up with

a somewhat large portion of duplicate keys drawn. Other distribution would also be interesting to put to the test, as well as evaluating sequentially unique keys, in order to see how the implementation adapts to other recognizable real-life access patterns. Nevertheless, the uniform integer distribution should be regarded as foundationally sufficent to obtain performance evaluation of valid character. The values provided to `--op-distr-high`/`--build-distr-high` and `--op-distr-low`/`--build-distr-low` as seen above ensures that the entire uniform range is used during all test runs.

4.3 Insert performance

The insert performance results are presented in Figure 4.1. An accompanying breakdown of the parallel B$^+$-trees main executing threads time expenditures can be found in Table 4.1.

Secondary variables and their value:

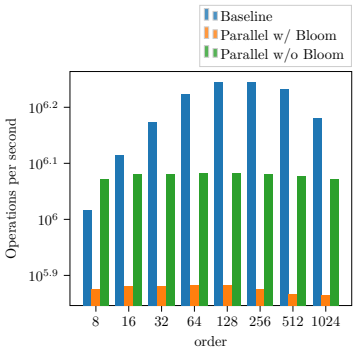
- Figure 4.1a, 4.1b: `numThreads` = 32, `numTrees` = 32
- Figure 4.1c, 4.1d: `order` = 128, `numTrees` = 32
- Figure 4.1e, 4.1f: `order` = 128, `numThreads` = 16

4.3.1 `order` impact on insert throughput

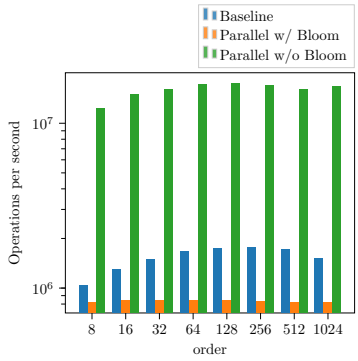
Figure 4.1a and Figure 4.1b shows how the order of the B$^+$-tree(s) affect the insert throughput. The baseline shows steady increase in performance up until the 128 and 256 `order` marks, from there a slight decline is observed.

Regarding the parallel instances during single key operation mode, both show leveled performance. The leveled performance is explainable by taking a look at the time expenditures show in Table 4.1. For the bar charts in question the majority of operation time is spent pushing tasks to the thread pool. This is clearly seen by the pushing-waiting ratio being above 1. When running without Bloom filters this ratio is extremely high, ranging from around 602 to 4143. As such, there is indication that pushing more meaningful tasks to the pool, i.e. tasks containing multiple operations, should be beneficial. Of course, this bottleneck means that in such configuration the insert throughput is largely limited by single threaded performance, with an additional cost of dividing each insert operation into a two-step process. Having such a high ratio means that the value of `order` becomes insignificant, since any potential speed up will be eaten away by the pushing time.

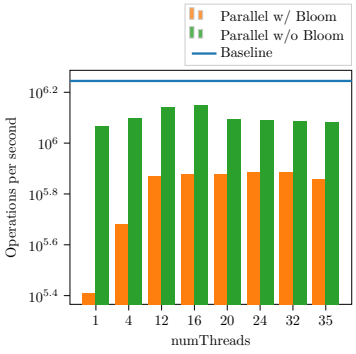
For the single key operations, the without Bloom filters variant unsurprisingly lies above its counterpart throughout the range of values. This is due to the with Bloom filters variant having to manage the Bloom filters, which involves additional locking, reading and insert overhead.



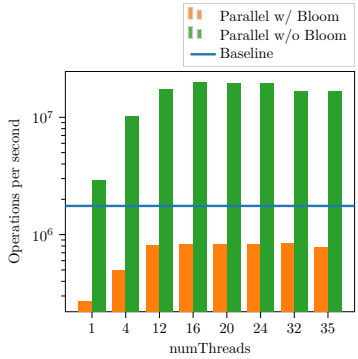
(a) `order` impact on throughput - Single key operation.



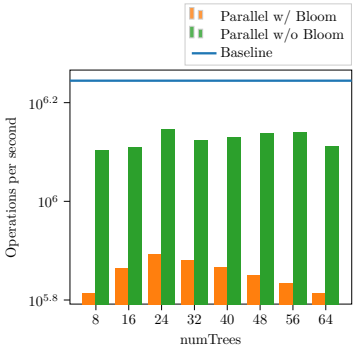
(b) `order` impact on throughput - Batch operation.



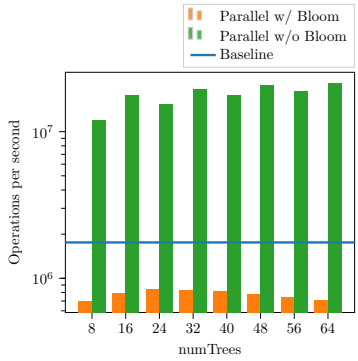
(c) `numThreads` impact on throughput - Single key operation.



(d) `numThreads` impact on throughput - Batch operation.



(e) `numTrees` impact on throughput - Single key operation.



(f) `numTrees` impact on throughput - Batch operation.

Figure 4.1: Insert performance of baseline and parallel B^+-tree implementations.

Table 4.1: Time consumption grouped by stage during parallel B$^+$-tree insert operation. *Pushing* refers to time spent pushing tasks to the thread pool, i.e. time spent accessing the public `insert` operation's entry point. *Waiting* refers to time spent waiting for work to finish using the `waitForWorkToFinish` method. Low combined *Pushing* and *Waiting* time yields high throughput.

Variable	Figure reference	Bloom filter	Variable value	Pushing (ms)	Waiting (ms)	Pushing-Waiting ratio
`order`	Figure 4.1a (Single key)	w/	8	4893	1752	2.792808
			16	4468	2109	2.11854
			32	4497	2079	2.163059
			64	3951	2587	1.527252
			128	4614	1919	2.404377
			256	4352	2289	1.901267
			512	4356	2444	1.782324
			1024	4602	2211	2.081411
		w/o	8	4219	7	602.714286
			16	4141	4	1035.25
			32	4139	6	689.833333
			64	4127	5	825.4
			128	4121	2	2060.5
			256	4143	1	4143.0
			512	4172	2	2086.0
			1024	4226	2	2113.0
	Figure 4.1b (Batch)	w/	8	0	6061	0.0
			16	0	5889	0.0
			32	0	5898	0.0
			64	0	5888	0.0
			128	0	5934	0.0
			256	0	6046	0.0
			512	0	6076	0.0
			1024	0	6055	0.0
		w/o	8	0	408	0.0
			16	0	335	0.0
			32	0	310	0.0
			64	0	290	0.0
			128	0	287	0.0
			256	0	294	0.0
			512	0	321	0.0
			1024	0	300	0.0
`numThreads`	Figure 4.1c (Single key)	w/	1	563	18998	0.029635
			4	1188	9235	0.128641
			12	2579	4195	0.614779
			16	3456	3192	1.082707
			20	3966	2685	1.477095
			24	4485	2049	2.188873
			32	4562	1937	2.355188
			35	3241	3708	0.874056
		w/o	1	1147	3133	0.366103
			4	3844	145	26.510345
			12	3608	8	451.0
			16	3557	7	508.142857
			20	4025	6	670.833333
			24	4064	2	2032.0
			32	4107	2	2053.5
			35	4156	3	1385.333333
	Figure 4.1d (Batch)	w/	1	0	18332	0.0
			4	0	10068	0.0
			12	0	6111	0.0
			16	0	6013	0.0
			20	0	6079	0.0
			24	0	6042	0.0
			32	0	5981	0.0

			35	0	6360	0.0
		w/o	1	0	1724	0.0
			4	0	494	0.0
			12	0	290	0.0
			16	0	252	0.0
			20	0	254	0.0
			24	0	257	0.0
			32	0	300	0.0
			35	0	301	0.0
numTrees	Figure 4.1e (Single key)	w/	8	2587	5099	0.507354
			16	2971	3850	0.771688
			24	3281	3108	1.055663
			32	3596	2974	1.209146
			40	3308	3491	0.947579
			48	2991	4079	0.733268
			56	2548	4770	0.534172
			64	2605	5073	0.513503
		w/o	8	3939	1	3939.0
			16	3876	3	1292.0
			24	3565	3	1188.333333
			32	3755	5	751.0
			40	3705	4	926.25
			48	3693	1	3693.0
			56	3624	6	604.0
			64	3868	5	773.6
	Figure 4.1f (Batch)	w/	8	0	7210	0.0
			16	0	6330	0.0
			24	0	5931	0.0
			32	0	6011	0.0
			40	0	6114	0.0
			48	0	6417	0.0
			56	0	6710	0.0
			64	0	7049	0.0
		w/o	8	0	416	0.0
			16	0	281	0.0
			24	0	327	0.0
			32	0	255	0.0
			40	0	282	0.0
			48	0	240	0.0
			56	0	264	0.0
			64	0	233	0.0

When taking a look at the batch operation mode, a performance boost is observed for both variations of the parallel B$^+$-tree. The without filters variant hugly benefits from dividing the inserts by `numTrees` and only locking once. Pushing time for both becomes 0 ms, but the full parallel processing capabilities are only realized in the without variant since it has no lock collisions. Further, as can be seen in Figure 4.1b, there is about 10 times throughput improvement in the without variant compared to its single key operation mode (note the y-axis change from 10^6 to 10^7). As such, the baseline is vastly outperformed. The with variant's improvement is completely overshadowed by the without's improvement. But, once again looking at the relevant parts of Table 4.1 reveals that in batch operation mode the with variant consistently hits running times right around 6000 ms and below, compared to 6500 ms in the single key operation mode. Locking and Bloom filter managment evidently becomes the bottleneck of this configuration.

Spending time on the actual insert tasks, as opposed to pushing tasks to the thread pool, shows that the parallel variants are susceptible to the value of `order` as well. Similar to the baseline, best performance is observed around the 128 `order` mark.

The reason as to why 128 is a good choice for the value of `order` during `insert`, most likely has to do with the cache layout and sizes of the computer used for evaluating the operations performance. The integer data type in C++ is 4 bytes, and the fill factor reached during these tests seems to allign nicely with stacking multiple data structure containing the keys into the closest caches when `order` is around 128.

The results obtained and explained in this subsection led to `order` having 128 as value for the remaining insert performance tests to be described.

4.3.2 `numThreads` impact on insert throughput

The impact of adjusting `numThreads` for the parallel B^+-tree variants can be seen in Figure 4.1c and Figure 4.1d. The blue line labeled "Baseline" shows the `order` = 128 performance for the baseline B^+-tree found in Figure 4.1a and Figure 4.1b.

First describing the single key operation mode and the with Bloom filters variant. A performance increase is observed until `numThreads` surpasses the available number of kernel-level threads (32). The increase is most notable when moving from 1, to 4, to 12 threads in the thread pool. Beyond this, performance increase is severly reduced due to locking incurring overhead of such character that additional parallel processing capabilities added, are simply wasted away waiting for locks to become available. Of course, moving from 32 to 35 threads is more of a sanity check than anything else. Performance at this stage is expected to drop, because a value beyond the maximum number of supported kernel-level threads necessarily incurs context switching overhead.

Taking a look at the single key operation mode and the without Bloom filter variant some interesting results are observed. Perfomance increase is found until `numThreads` = 16, but then at `numThreads` = 20 a relatively larger dip is seen. From here on, there is slight decrease in performance as `numThreads` continue to increase. Remember that the without Bloom filters variant still uses locks, but no filter locks. It seems the saturation point for when such simple locking becomes to much, is `numThreads` = 16 for in single key operation mode. Studying relevant parts of Table 4.1 we see that there is however improvements in waiting time beyond 16 threads, but the additional pushing time makes the throughput turn out worse. This is most likely due to the task queue of the thread pool being a bottleneck. As more threads are actively popping tasks from the queue, the main executing thread will more frequently have to wait to push new tasks. Thus, also using multiple thread pools when the computer used supports a great number of threads, might show further increase in performance.

In batch operation mode, the with Bloom filters variant shows similar development to its counterpart single key operation mode. The insert throughput is slightly better, something that is clear from looking at the total time expenditures in Table 4.1. The without variant outperforms the baseline and shows bigger leaps in performance when increasing the numer of threads, compared to its counterpart

single key operation mode. However, while 16 threads remains the peak, the "major" dip is now observed moving from 24 to 32 threads. As the time expenditures table confirms, it is the waiting time that grows. This is interesting since, in the single key operation mode, splitting the workload by `numTrees` showed improved waiting time when utilizing a greater number of threads. A possible explanation for this behavior is that with 5000000 keys and the great performance increase, finding, accessing and moving the keys to be inserted between the caches of cores located on two sockets, turns out more expensive than keeping it all close to one, and letting each thread of the thread pool complete 1 or 2 tasks.

Based on the results of this subsection `numThreads` has value 16 for the last insert performance test group to be described. A last point as to why 16 might be a reasonable value is that the computer used for performance evaluation after all, in reality have 16 cores (8 cores on 2 sockets). Thus, using anything beyond 16 threads also becomes a test of hyper-threading performance. So, to not introduce unnecessary noise 16 is the value of choice, although the parallel B^+-tree as aforementioned, showed increased performance until matching the number of kernel-level threads.

4.3.3 `numTrees` impact on insert throughput

The results that show how the value of `numTrees` impact the insert throughput can be found in Figure 4.1e and Figure 4.1f. Regarding the with Bloom filters variant, development is similar in both single key and batch operation mode. Peak insert throughput is in both cases observed at `numTrees` = 24, but the batch operation mode outperforms its counterpart by 60424 ops. Comparing the two throughout the range shows higher throughput in favor of batch operation mode, with values including and in between 42929-72042 ops. Table 4.1 shows that the pushing-waiting ratio during single key operation mode at the peak is relatively high compared to most other values in the range. In both cases for the with Bloom filters variant, a small reduction in throughput is observed after the peak for each increment in `numTrees`. It is interesting that 24 is the value in both cases, where the overhead of Bloom filter managment hits its sweet spot when `numThreads` = 16 and `order` = 128 on this computer.

When looking at the without Bloom filters variant, the results are a bit inconclusive during single key operation mode, but during batch operation mode an interesting pattern emerges. Insert throughput spikes during times when `numTrees` is a multiple of 16. As previously mentioned, 16 is the number of cores the machine has, so these results makes sense, i.e. the underlying supporting architecture impacts the insert throughput when performance is high. Lastly, we see the highest insert throughput measurements during insert performance tests combining all discoveries made, with peaks over 20 Mops. In fact, the results does not top out on the range of `numTrees` values tested on. A top will necessarily exist since batch operation mode in the end becomes a form of single key operation mode (in the extreme case where each insert operation gets its own tree). Also, although the baseline is clearly outperformed by this parallel configuration, the number of

B^+-trees will make the other operations more complex and time consuming, as we will see.

4.4 Search performance

The search performance results are presented in Figure 4.2. Table 4.2 shows the breakdown of the parallel B^+-trees main executing threads time expenditures.

Secondary variables and their value:

- Figure 4.2a, 4.2b: `numThreads` = 32, `numTrees` = 32
- Figure 4.2c, 4.2d: `order` = 1024, `numTrees` = 32
- Figure 4.2e, 4.2f: `order` = 1024, `numThreads` = 32

For all search tests, about 63% of the keys searched for are found in the tree. This is due to the way in which keys are drawn, both during build and search as explained during the last part of section 4.2.

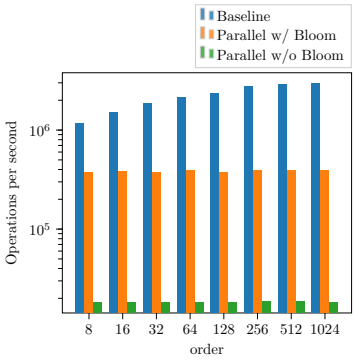
Note that, even though none of the performance runs displayed in Figure 4.2 beats their comparable baseline, there are in fact some sligthly more extreme configurations on the computer used during testing that will, e.g. batching, ensuring `order` = `numThreads` = `numTrees` = 4 yielded 1.3 Mops, which beats a baseline `order` = 4 having around 740 Kops. This indicates that the in-memory data structure can benefit from reduced tree height when compared against the baseline's height during search.

4.4.1 `order` impact on search throughput

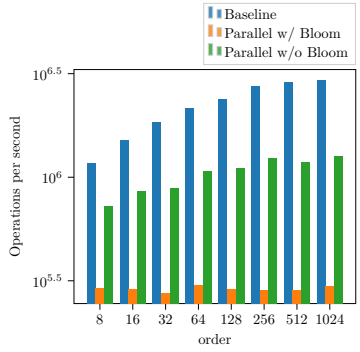
How different values of `order` affects the implementations search throughput can be seen in Figure 4.2a and Figure 4.2b. Both in single key and batch operation mode the baseline outperforms the parallel implementations for all values of `order` tested. At most the baseline peaks at close to 3 Mops.

The with Bloom filters variant is the best candidate of the two variations when looking at single key operation mode. Albeit, its performance is fairly leveled throughout the range. Table 4.2 shows that its pushing-waiting ratio ranges from 8-14 in this case, while a great increase in pushing-waiting ratio, making the value lie between 169-285, is found during its batch operation mode. Tasking the main executing thread with correctly identifying the trees to search is likely the reason for this increase. The silver lining of the latter results is however that waiting times mostly lie below 100 ms, which shows the parallel potiential when fully realized.

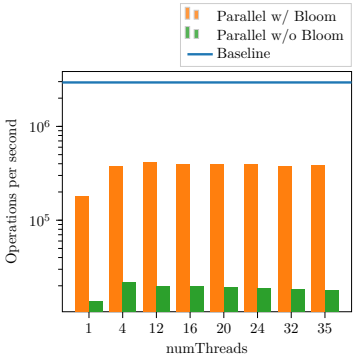
The without variant on its side becomes the prime candidate when looking at batch operation mode. Here the pushing-waiting ratio sinks a bit below 1 as seen



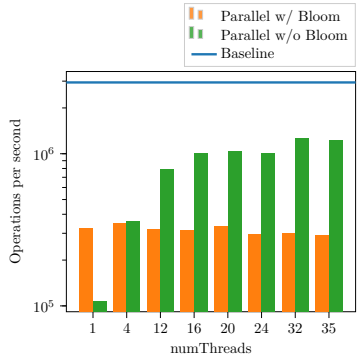
(a) `order` impact on throughput - Single key operation.



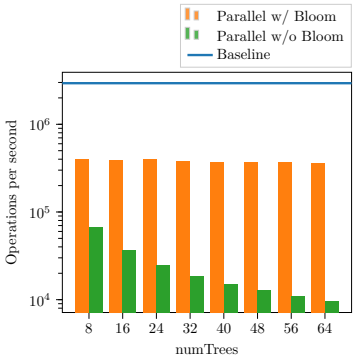
(b) `order` impact on throughput - Batch operation.



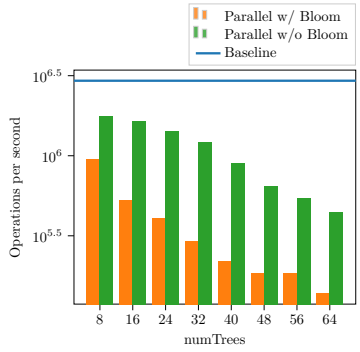
(c) `numThreads` impact on throughput - Single key operation.



(d) `numThreads` impact on throughput - Batch operation.



(e) `numTrees` impact on throughput - Single key operation.



(f) `numTrees` impact on throughput - Batch operation.

Figure 4.2: Search performance of baseline and parallel B^+-tree implementations.

Table 4.2: Time consumption grouped by stage during parallel B$^+$-tree search operation. *Pushing* refers to time spent pushing tasks to the thread pool, i.e. time spent accessing the public `search` operation's entry point. *Waiting* refers to time spent waiting for work to finish using the `waitForWorkToFinish` method. Low combined *Pushing* and *Waiting* time yields high throughput.

Variable	Figure reference	Bloom filter	Variable value	Pushing (ms)	Waiting (ms)	Pushing-Waiting ratio
order	Figure 4.2a (Single key)	w/	8	11973	1364	8.777859
			16	11970	991	12.078708
			32	11854	1436	8.254875
			64	11982	882	13.585034
			128	11925	1431	8.333333
			256	11856	847	13.997639
			512	11936	878	13.594533
			1024	11860	809	14.660074
		w/o	8	134480	137917	0.975079
			16	135399	137161	0.987154
			32	135100	136606	0.988976
			64	136842	135797	1.007695
			128	135971	135432	1.00398
			256	135487	135348	1.001027
			512	135222	135305	0.999387
			1024	136079	135015	1.007881
	Figure 4.2b (Batch)	w/	8	17157	101	169.871287
			16	17363	78	222.602564
			32	18033	66	273.227273
			64	16536	58	285.103448
			128	17382	66	263.363636
			256	17454	64	272.71875
			512	17520	64	273.75
			1024	16829	60	280.483333
		w/o	8	1586	5303	0.299076
			16	1954	3881	0.503478
			32	1606	4021	0.399403
			64	1572	3124	0.503201
			128	1923	2592	0.741898
			256	1822	2229	0.817407
			512	1892	2343	0.807512
			1024	1576	2378	0.662742
numThreads	Figure 4.2c (Single key)	w/	1	2201	25115	0.087637
			4	5045	8107	0.622302
			12	11234	780	14.402564
			16	11362	1086	10.462247
			20	11593	973	11.914697
			24	11758	795	14.789937
			32	11868	1389	8.544276
			35	11915	1004	11.86753
		w/o	1	3828	357730	0.010701
			4	14323	212245	0.067483
			12	47648	202665	0.235107
			16	63843	190188	0.335684
			20	81242	178706	0.454613
			24	100727	162380	0.620317
			32	134849	133759	1.008149
			35	151020	123820	1.219674
	Figure 4.2d (Batch)	w/	1	14162	1321	10.720666
			4	14073	182	77.324176
			12	15598	55	283.6
			16	15730	46	341.956522
			20	14921	50	298.42
			24	16899	50	337.98
			32	16536	60	275.6

			35	16989	53	320.54717
		w/o	1	1346	44962	0.029936
			4	1390	12383	0.112251
			12	1729	4529	0.381762
			16	1374	3561	0.385847
			20	1391	3438	0.404596
			24	1517	3413	0.444477
			32	1824	2140	0.852336
			35	1845	2229	0.827725
`numTrees`	Figure 4.2e (Single key)	w/	8	11656	806	14.461538
			16	11616	1300	8.935385
			24	11650	798	14.598997
			32	11857	1398	8.481402
			40	12222	1235	9.896356
			48	12165	1293	9.408353
			56	12320	1278	9.640063
			64	12474	1520	8.206579
		w/o	8	70260	4394	15.989986
			16	128206	9735	13.169594
			24	131809	71900	1.833227
			32	135478	134880	1.004434
			40	138322	195783	0.706507
			48	137288	258101	0.531916
			56	139716	320189	0.436355
			64	140106	385536	0.363406
	Figure 4.2f (Batch)	w/	8	5038	252	19.992063
			16	9330	93	100.322581
			24	12273	65	188.815385
			32	16998	53	320.716981
			40	22681	48	472.520833
			48	27128	37	733.189189
			56	27221	35	777.742857
			64	36213	28	1293.321429
		w/o	8	610	2205	0.276644
			16	905	2155	0.419954
			24	1244	2283	0.544897
			32	1587	2504	0.633786
			40	2194	3360	0.652976
			48	2294	5511	0.416258
			56	2694	6499	0.414525
			64	2939	8388	0.350381

in Table 4.2. Development throughout the range is similar to that of the baseline. Although, its pushing-wating ratio is close to 1 during single key operation mode, the implementation is in this case plagued by both pushing and waiting times of high absolute values. This is because the `threadSearchCoordinator` pushes an additional `numTrees` tasks to the pool for each single key task push by the main executing thread. Which most likeley causes a thread pool access bottleneck.

In any case, the most promising looking value for `order` when trying to achive high search thoughput seems to be 1024, so this is the value chosen moving forward. Interestingly, this contrasts the 128 value obtained when examining insert performance in section 4.3. This is possibly due to more surrounding data structures (e.g. `std::promise` and vector return types) being used in comparison to the `insert` operations.

4.4.2 `numThreads` impact on search throughput

The search throughput impact of adjusting `numThreads` can be found in Figure 4.2c and Figure 4.2d.

In single key operation mode both variants make it clear that adding additional parallel processing resources is useless if the bottleneck is caused by troubles in effectively distributing the tasks quickly enough. Table 4.2 shows how waiting time grows from 2201 ms to 11915 ms, and from 3828 ms to 151020 ms, for the with and without Bloom filters variants respectively. Remember that for any run the number of tasks to be pushed is 5000000. So, it is clear in single key operation mode that adding threads to the thread pool severly reduces task distribution time. Simultaneously, the reduction in waiting time is not observed, or not enough, to counteract the pushing time effects for values of `numThreads` surpassing 12 and 4, for with and without Bloom filters respectively.

Batch operation mode shows more promising results in utilizing parallel processing capabilities for the without Bloom filters variant. Somewhat sporadic leaps in performance until the maximum number of available kernel-level threads are in use is observed. The with variant on its side shows a more similar pattern to that of single key operation mode, not really benefiting much from the switch to batching. Centralizing the with variants task distribution on the main executing thread in hopes of speed up, does not seem worth it when comparing to increased lock contention as a result of increased `numThreads`, with a middleman coordination task pushed to the pool, as done during comparable parts of the `insert` operations. Batch mode does on the plus side for both variants show time expenditures that develop more predictibly compared to single key operation mode. That is, small increments in pushing times, and small decrements in waiting times for the majority of `numThreads` values.

4.4.3 `numTrees` impact on search throughput

The results regarding `numTrees` impact on search throughput can be found in Figure 4.2e and Figure 4.2f.

The without Bloom filters variant shows similar development during both single key and batch operation mode. Although, the numbers are much better during batch operation mode. The relationship between `numTrees` and search throughput is in this case inversely proportional. The same relationship holds when looking at the with Bloom filters variant. However, the relationship is then much more pronounced during batch operation mode. Even though this relationship is expected, it stands in stark contrast to the relationship derived between `numTrees` and insert throughput, at least during batch operation mode without use of Bloom filters, which shows proportionality. This trade-off would need assessment if configuring the implementation for use in a real system, i.e. are inserts or searches most important? Taking a look at Table 4.2 confirms that it is the pushing time that grows the most during batch operation mode for both variants, whereas dur-

ing single operation mode it is mainly waiting time increase that is responsible for degrading performance of the without variant, while the with variant stays fairly leveled here.

4.5 Update performance

In Figure 4.3 all update performance results are presented. The accompanying time expenditures table, which shows the breakdown of the parallel B$^+$-trees main executing threads, is Table 4.3.

Secondary variables and their value:

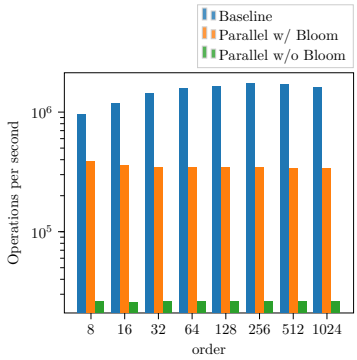
- Figure 4.3a, 4.3b: `numThreads` = 32, `numTrees` = 32
- Figure 4.3c, 4.3d: `order` = 512, `numTrees` = 32
- Figure 4.3e, 4.3f: `order` = 512, `numThreads` = 12

4.5.1 `order` impact on update throughput

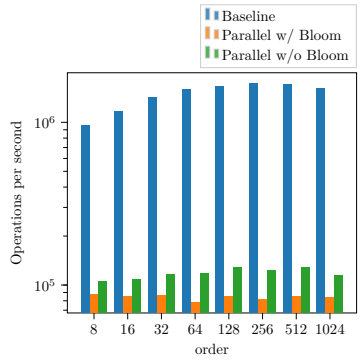
The results concerning how `order` affects the B$^+$-tree(s) update performance are presented in Figure 4.3a and Figure 4.3b. In single key operation mode both parallel variants do not display any significant variation in update throughput, in response to the value of `order` varying. Table 4.3 shows that their pushing-waiting ratios stays fairly stable over all values tested. Taking a look at the absolute times, the with variant spends almost all its time pushing and close to no time waiting. The without variant on its side have pushing-waiting ratios right around 1 for all values, but compared to the with variant, it uses vastly more time in absolute terms. Since single key updating by use of a coordination-task is more complex than the other comparable parallel B$^+$-tree operations (complex in the sense that a great number of modifiable (sub)tasks might be added to the thread pool's task queue), the payoff when using Bloom filters becomes clear in single key operation mode.

In batch operation mode the without variant performs better than the with variant. Albeit, peak performance does not surpass the results obtained for the parallel B$^+$-trees in single key operation mode. For both variants the update throughput does not move in any one direction, when moving from start to finish on the range of values put to the test. Time expenditures for both however show tendency towards reduced waiting time when the value of `order` increases. This makes sense as the impact of varying `order` should mostly affect the actual operations carried out involving the B$^+$-trees.

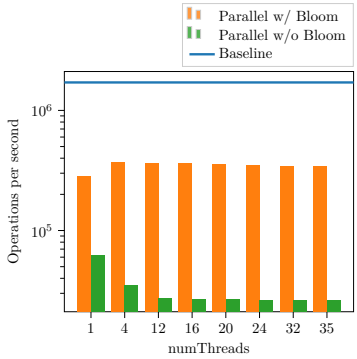
The baseline outperforms both variations throughput on the entire range of values tested. Since the single key operation mode variants do not seem too affected by the value of `order`, and the batch operation mode results are somewhat inconclusive, the choice of `order` to use when generating the other update performance



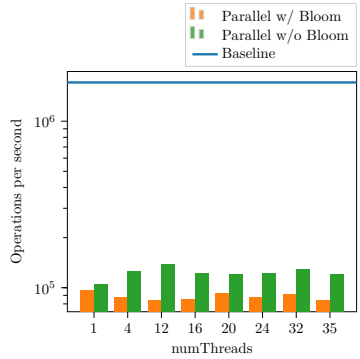
(a) `order` impact on throughput - Single key operation.



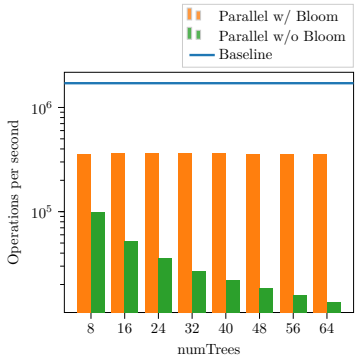
(b) `order` impact on throughput - Batch operation.



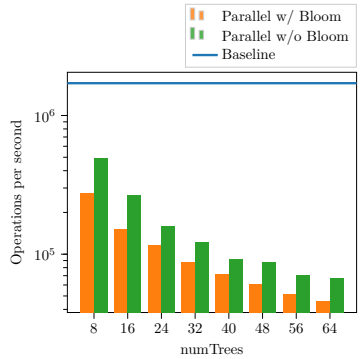
(c) `numThreads` impact on throughput - Single key operation.



(d) `numThreads` impact on throughput - Batch operation.



(e) `numTrees` impact on throughput - Single key operation.



(f) `numTrees` impact on throughput - Batch operation.

Figure 4.3: Update performance of baseline and parallel B$^+$-tree implementations.

Table 4.3: Time consumption grouped by stage during parallel B$^+$-tree update operation. *Pushing* refers to time spent pushing tasks to the thread pool, i.e. time spent accessing the public `update` operation's entry point. *Waiting* refers to time spent waiting for work to finish using the `waitForWorkToFinish` method. Low combined *Pushing* and *Waiting* time yields high throughput.

Variable	Figure reference	Bloom filter	Variable value	Pushing (ms)	Waiting (ms)	Pushing-Waiting ratio
order	Figure 4.3a (Single key)	w/	8	12940	32	404.375
			16	14035	2	7017.5
			32	14590	12	1215.833333
			64	14544	1	14544.0
			128	14543	2	7271.5
			256	14558	2	7279.0
			512	14663	2	7331.5
			1024	14751	3	4917.0
		w/o	8	92596	97780	0.946983
			16	94979	98727	0.962037
			32	93644	99141	0.944554
			64	95324	96502	0.987793
			128	95383	96002	0.993552
			256	95683	96451	0.992037
			512	95792	95682	1.00115
			1024	95131	96312	0.987738
	Figure 4.3b (Batch)	w/	8	56106	1024	54.791016
			16	57340	973	58.931141
			32	57026	810	70.402469
			64	63507	102	622.617647
			128	57936	465	124.593548
			256	60453	387	156.209302
			512	58068	223	260.394619
			1024	59523	195	305.246154
		w/o	8	45465	2179	20.865076
			16	44412	1408	31.542614
			32	41966	1228	34.174267
			64	41359	1052	39.314639
			128	38211	913	41.852136
			256	40024	581	68.888124
			512	38118	681	55.973568
			1024	42915	460	93.293478
numThreads	Figure 4.3c (Single key)	w/	1	1035	16623	0.062263
			4	2697	10767	0.250488
			12	13759	2	6879.5
			16	13739	1	13739.0
			20	14011	8	1751.375
			24	14328	9	1592.0
			32	14619	5	2923.8
			35	14719	2	7359.5
		w/o	1	1311	79465	0.016498
			4	8750	133418	0.065583
			12	32734	150422	0.217614
			16	42675	144502	0.295325
			20	54416	134463	0.404691
			24	66843	124223	0.538089
			32	96182	94729	1.015338
			35	106236	86182	1.232694
	Figure 4.3d (Batch)	w/	1	50511	1422	35.521097
			4	57392	153	375.111111
			12	58980	359	164.289694
			16	58795	13	4522.692308
			20	54190	271	199.9631
			24	57139	350	163.254286
			32	54935	245	224.22449

			35	59626	275	216.821818
		w/o	1	28859	19114	1.509836
			4	39367	700	56.238571
			12	35778	570	62.768421
			16	40309	864	46.653935
			20	40738	707	57.620934
			24	39965	919	43.487486
			32	38192	737	51.820896
			35	40656	835	48.68982
numTrees	Figure 4.3e (Single key)	w/	8	13903	1	13903.0
			16	13728	2	6864.0
			24	13721	2	6860.5
			32	13723	3	4574.333333
			40	13803	5	2760.6
			48	14011	3	4670.333333
			56	14060	4	3515.0
			64	11234	2841	3.954241
		w/o	8	29228	21673	1.34859
			16	31141	63696	0.4889
			24	32216	105946	0.304079
			32	33171	152427	0.217619
			40	32302	195594	0.165148
			48	33087	239800	0.137977
			56	34319	283477	0.121064
			64	34407	330215	0.104196
	Figure 4.3f (Batch)	w/	8	17555	751	23.375499
			16	32735	490	66.806122
			24	42502	510	83.337255
			32	57369	404	142.002475
			40	69765	362	192.720994
			48	83037	202	411.074257
			56	96778	296	326.952703
			64	109449	334	327.691617
		w/o	8	8974	1287	6.972805
			16	18109	672	26.947917
			24	30873	706	43.729462
			32	40160	776	51.752577
			40	53806	817	65.858017
			48	56793	125	454.344
			56	70597	718	98.324513
			64	74241	416	178.463942

results is 512. 512 produces reasonable throughput results, both for the baseline and the two batch mode operation variants.

4.5.2 `numThreads` impact on update throughput

Figure 4.3c and Figure 4.3d presents the update throughput results obtained when varying the number of threads used in the thread pool. In single key operation mode the with variant only shows increased performance during the first step from 1 to 4 threads. Moving beyond 4 threads, performance declines sligthly with each step. Table 4.3 shows that at 12 threads and onwards the main executing thread's waiting time is in reality non-existent. This indicates that dispatching the update during the coordination-task, and completing it, takes less time than pushing a subsequent update task. The additional parallel processing capabilities are not utilized, while they incur additional cost reflected in the pushing times. In the end this leads to the declining development observed. The without variant shows

more dramatic decline in performance until 12 threads, and growing pushing-waiting ratios throughout the range. From 12 threads and onwards any reduction in waiting time is merely cancelled out by increase in pushing time, thus yielding similar throughput measurments.

In batch operation mode a `numThreads` value of 12 and 32 marks the peaks for the without variant. Although, multiple other `numThreads` values produce results fairly close by. Taking a look at the time expenditures reveales that batching the way it is done, makes the main executing thread spend almost all its time at the pushing stage for both variants. The waiting times are all really similar for the without variant when using more than 1 thread in the thread pool. The with variant on its side have one `numThreads` value that stands out when examining the waiting times. That is, `numThreads` = 16 which yields 13 ms spent waiting for this particular test run. The reason as to why 16 stands out, most likely has to do with the specifications of the computer used, as also discussed during subsection 4.3.2 where `numThreads` = 16 showed itself as a good value of choice.

Since it seems batch operation mode has the most to gain from adjusting `numThreads`, the peak of the without variant in this operation mode makes `numThreads` have value 12, for the `numTrees` update performance results to be presented next.

4.5.3 `numTrees` impact on update throughput

Lastly, the update performance results regarding how the value of `numTrees` impact throughput are presented in Figure 4.3e and Figure 4.3f. The same inversely proportional relationship between `numTrees` and update throughput is found for all variants, except the with Bloom filters single key operation mode variant. This is the same relationship that was found when examining `numTrees` impact on search throughput in subsection 4.4.3. In batch operation mode the relationship is due to increased pushing times, stemming from more partitions and tasks created on the main executing thread. While for the without Bloom filters single key operation mode variant, the relationship is due to increased waiting times, stemming from the coordination-task creating more partitions and tasks. The with variant in single key operation mode remains level with minimal decline due to the bottleneck of this particular design seemingly being the pushing stage. However, when looking at the time expenditures one data point stands out. That is, `numTrees` = 64 forces a shift in distribution of time between stages, yielding a comparativly low pushing-waiting ratio of 4. This shift might be due to the amount of storage required for the data structure reaching a value which demands the computer manages the data in a different way.

4.6 Remove performance

Figure 4.4 presentes the remove performance results. The accompanying time expenditures table, which shows the breakdown of the parallel B^+-trees main executing threads, is Table 4.4.

Secondary variables and their value:

- Figure 4.4a, 4.4b: `numThreads` = 32, `numTrees` = 32
- Figure 4.4c, 4.4d: `order` = 1024, `numTrees` = 32
- Figure 4.4e, 4.4f: `order` = 1024, `numThreads` = 32

4.6.1 `order` impact on remove throughput

The impact of `order` on remove throughput can be seen in Figure 4.4a and Figure 4.4b. In single key operation mode both parallel variants are outperformed by the baseline. Both variants stay level throughout the range of values, and the with Bloom filters variant performs much better than the without variant.

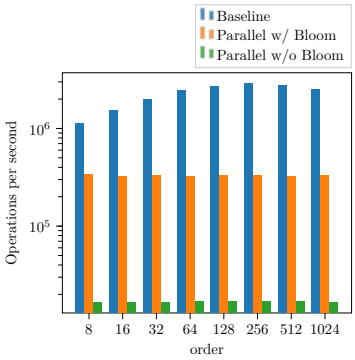
The way better results are found when looking at batch operation mode. Here the without variant beats all other competitors for all values of `order`. Wheras the baseline peaks at `order` = 256, the without variant continues to show performance increase until the last value tested, namely `order` = 1024. At this point performance is about 4.5 Mops compared to the baseline's 2.5 Mops. Expectedly, remove performance should increases up until a certain point when `order` increases as long as the bottleneck of performance is the actual remove operation. This is due to more simple removes being carried out, and less merges and redistributions occurring. Table 4.4 shows that the batch operation mode without variant has pushing times of 0 ms, stemming from the simplicity of splitting workload by `numTrees` and optimized compiling. The throughput performance of the batch operation mode with variant on its side is way worse, but comparable to its single key operation mode counterpart.

Due to the high performer being the batch operation mode without variant, and it peaking at `order` = 1024, 1024 is chosen as the fixed value of `order` for the other remove performance tests carried out.

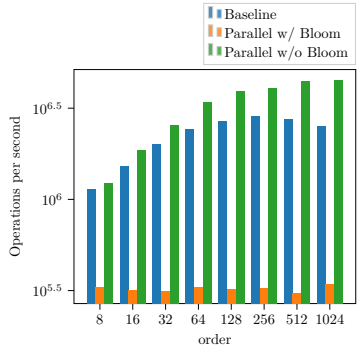
4.6.2 `numThreads` impact on remove throughput

The results concerning `numThreads` affect on remove throughput are presented in Figure 4.4c and Figure 4.4d.

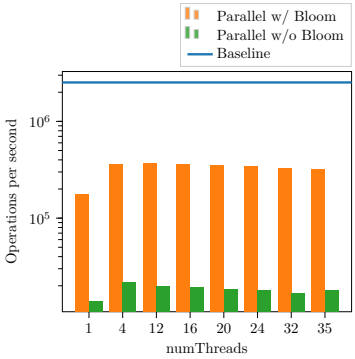
In single key operation mode the with variant reaches its 371 Kops peak at 12 threads. Meanwhile, the peak of the without variant is just 22 Kops with 4 threads in use. As confirmed by looking at time expenditures, increasing the number of threads leads to longer pushing times due to more contention and managment of



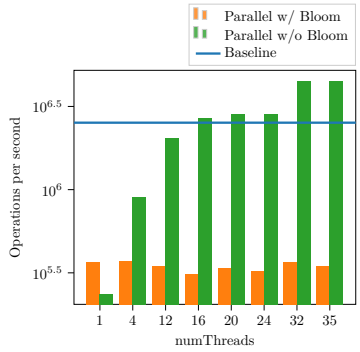
(a) `order` impact on throughput - Single key operation.



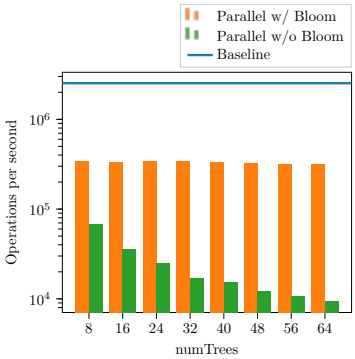
(b) `order` impact on throughput - Batch operation.



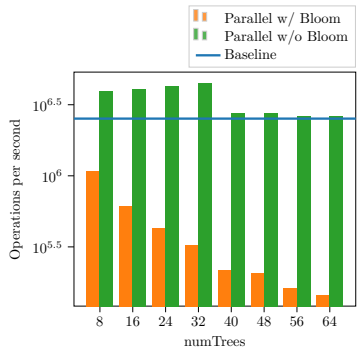
(c) `numThreads` impact on throughput - Single key operation.



(d) `numThreads` impact on throughput - Batch operation.



(e) `numTrees` impact on throughput - Single key operation.



(f) `numTrees` impact on throughput - Batch operation.

Figure 4.4: Remove performance of baseline and parallel B^+-tree implementations.

Table 4.4: Time consumption grouped by stage during parallel B$^+$-tree remove operation. *Pushing* refers to time spent pushing tasks to the thread pool, i.e. time spent accessing the public `remove` operation's entry point. *Waiting* refers to time spent waiting for work to finish using the `waitForWorkToFinish` method. Low combined *Pushing* and *Waiting* time yields high throughput.

Variable	Figure reference	Bloom filter	Variable value	Pushing (ms)	Waiting (ms)	Pushing-Waiting ratio
order	Figure 4.4a (Single key)	w/	8	13523	1247	10.844427
			16	13937	1463	9.526316
			32	13775	1338	10.295217
			64	14166	1420	9.976056
			128	13982	1203	11.62261
			256	13591	1367	9.942209
			512	14032	1429	9.819454
			1024	13974	1158	12.067358
		w/o	8	132408	165198	0.801511
			16	133498	165042	0.808873
			32	133992	164065	0.816701
			64	132657	162088	0.818426
			128	133148	163661	0.81356
			256	133336	162260	0.821743
			512	133039	161359	0.824491
			1024	135251	164302	0.823185
	Figure 4.4b (Batch)	w/	8	14833	202	73.430693
			16	15560	131	118.778626
			32	15805	93	169.946237
			64	15114	93	162.516129
			128	15461	77	200.792208
			256	15198	61	249.147541
			512	16215	72	225.208333
			1024	14565	64	227.578125
		w/o	8	0	4091	0.0
			16	0	2683	0.0
			32	0	1945	0.0
			64	0	1459	0.0
			128	0	1274	0.0
			256	0	1222	0.0
			512	0	1124	0.0
			1024	0	1110	0.0
numThreads	Figure 4.4c (Single key)	w/	1	2135	26018	0.082059
			4	4587	9404	0.487771
			12	12606	854	14.761124
			16	12810	943	13.584305
			20	12918	1161	11.126615
			24	13370	1125	11.884444
			32	14008	1331	10.524418
			35	14279	1417	10.076923
		w/o	1	4015	353707	0.011351
			4	13390	216183	0.061938
			12	46070	208996	0.220435
			16	63462	197041	0.322075
			20	80143	189990	0.421827
			24	98759	182156	0.542167
			32	134037	161731	0.828765
			35	147421	129694	1.136683
	Figure 4.4d (Batch)	w/	1	12850	731	17.578659
			4	13153	211	62.336493
			12	14400	106	135.849057
			16	15919	95	167.568421
			20	14648	88	166.454545
			24	15340	91	168.571429
			32	13481	75	179.746667

			35	14395	60	239.916667
		w/o	1	0	21105	0.0
			4	0	5516	0.0
			12	0	2460	0.0
			16	0	1866	0.0
			20	0	1752	0.0
			24	0	1754	0.0
			32	0	1111	0.0
			35	0	1118	0.0
numTrees	Figure 4.4e (Single key)	w/	8	13271	1371	9.679796
			16	13637	1370	9.954015
			24	13603	1259	10.804607
			32	13710	1014	13.52071
			40	13976	1035	13.503382
			48	14382	1200	11.985
			56	14568	1257	11.589499
			64	14800	1247	11.868484
		w/o	8	68663	6207	11.062188
			16	122758	18291	6.711388
			24	129173	70503	1.832163
			32	133219	162723	0.818686
			40	131272	195735	0.670662
			48	134322	276359	0.486042
			56	136765	335857	0.407212
			64	138555	398137	0.348008
	Figure 4.4f (Batch)	w/	8	4476	180	24.866667
			16	7892	222	35.54955
			24	11553	87	132.793103
			32	15254	63	242.126984
			40	23021	62	371.306452
			48	24039	60	400.65
			56	30664	37	828.756757
			64	34698	43	806.930233
		w/o	8	0	1261	0.0
			16	0	1235	0.0
			24	0	1164	0.0
			32	0	1107	0.0
			40	0	1811	0.0
			48	0	1820	0.0
			56	0	1910	0.0
			64	0	1886	0.0

the thread pool. The with variants peak marks the point were further reduction in waiting times does not occur, while its pushing times continues to increase. The without variant on its side sees reductions in waiting times and increases in pushing times on the entire range of values. Even performing marginally better at a value of 35 than 32. Why that is, is not exactly clear. It might just be a particularly "good" run. Since performance at this point is so low, and the difference is minuscule, it is not deemed worthwhile investigating.

In batch operation mode the without variant displays a much more dependent pattern on the number of threads in use. Performance increases sharply until about 16 threads, and then sharply again moving from 24 to 32 threads. From 16 threads and onwards the baseline is beaten. As aforementioned, 16 is the computers number of cores, and with hyper-threading 32 threads are supported. Thus, the development is in line with what the computers underlying architecture supports. Unfortunately, the with variant does not perform as well or predictably. Once again, as also discussed during previous operations, this is most likely due to placing a to high demand on the main executing thread. The time expendi-

tures in this case shows that waiting times are really low, but pushing times are consistently plagued by the overhead associated with workload distribution in a fashion that attempts to avoid waiting for locks at the time of actually operating on the B$^+$-trees.

In hopes of achiving the best remove throughput, these results suggests that `numThreads` should have a value of 32 when generating the `numTrees` results.

4.6.3 `numTrees` impact on remove throughput

Figure 4.4e and Figure 4.4f shows how the value of `numTrees` impact remove throughput for the parallel variants.

In single key operation mode the with variant remains level, only declining slightly as `numTrees` increases. Since the Bloom filters have a low false positive probability, unneccessary tasks created by the coordination-task will be few even when the number of trees are high. Thus, the value of `numTrees` does not strongly affect this variant. The without variant however shows declining performance as `numTrees` increases. This is due to the number of B$^+$-tree remove calls being issued, being directly dependent on `numTrees`.

In batch operation mode the with variant displays similar development to that of the single key operation mode without variant. Although, its performance in absolute numbers is much better. The batch operation mode without variant beats the baseline with some margin, and increases performance until `numTrees` = 32. From there on, performance drops to right around the baseline's level. Of course, since the value of `numThreads` is 32, and workload in this case is split by `numTrees`, it makes sense that performance drops at above 32 trees since then some threads must carry out more than one task.

4.7 Profiling

To better understanding exactly where time gets spent during a portion of the operation modes, some configurations where selected for additional profiling. These are, one low performing search operation test, and one high performing insert operation test. Both these runs serve as examples helping illustrate and highlight when and why the parallel implementation scores better or worse.

Due to full access rights needed to perform profiling of the multithreaded code the correct way, a different computer to the one used during primary testing and performance evaluation was used for profiling. The two computers have the same x86_64 architecture, with similar performance development on the testable range of threads. Full relevant specifications of the computer used for profiling can be found in Listing 4.2. Once again, having an Intel processor, the profiling computer supports at most 4 kernel-level threads.

```
$ system_profiler -detailLevel mini SPSoftwareDataType
SPHardwareDataType
  Software:

    System Software Overview:

      System Version:  macOS 11.6.2 (20G314)
      Kernel Version:  Darwin 20.6.0

  Hardware:

    Hardware Overview:

      Model Name:  MacBook Pro
      Model Identifier:  MacBookPro11,1
      Processor Name:  Dual-Core Intel Core i5
      Processor Speed:  2,4 GHz
      Number of Processors:  1
      Total Number of Cores:  2
      L2 Cache (per Core):  256 KB
      L3 Cache:  3 MB
      Hyper-Threading Technology:  Enabled
      Memory:  4 GB
      System Firmware Version:  432.60.3.0.0
      SMC Version (system):  2.16f68
```

Listing 4.2: Specifications of the computer used for profiling.

4.7.1 Vizualisation tool and workflow

In order to convey information in an orderly fashion to the reader, two separate tools from the main profiling tool have been used. Those are, Brendan Gregg's flame graph vizualization tool [48] and Google's unofficial pprof tool [49].

Profiling runs were carried out using the Instruments performance analyzer [50], who is bundled with Xcode, and its Time Profiler template. From the raw profiles collected, the call tree's were deep copied and transformed to pprof readable files using the instrumentsToPprof tool [51]. Excerpts of top entry reports were generated using pprof, and using pprofutils [52], files of the folded text format needed to feed the flame graphs were generated.

Flame graph

To understand what the flame graphs actually display, here is a short quote from Brendan Gregg's own website [48]:

> The x-axis shows the stack profile population, sorted alphabetically (it is not the passage of time), and the y-axis shows stack depth, counting from zero at the bottom. Each rectangle represents a stack frame. The wider a frame is is, the more often it was present in the stacks. The top edge shows what is on-CPU, and beneath it is its ancestry. The colors are usually not significant, picked randomly to differentiate frames.

pprof reports

The pprof top entry reports show much the same as the flame graphs, but in a more detail specific way. This means that the flame graphs are best used to quickly overview the situation, and the graph reports allows for studying the profiling in more detail. Natively, the flame graphs are interactive SVGs, but in this report they are reproduces as images. Thus, the pprof graph reports are handy in their own right.

4.7.2 Search performance run

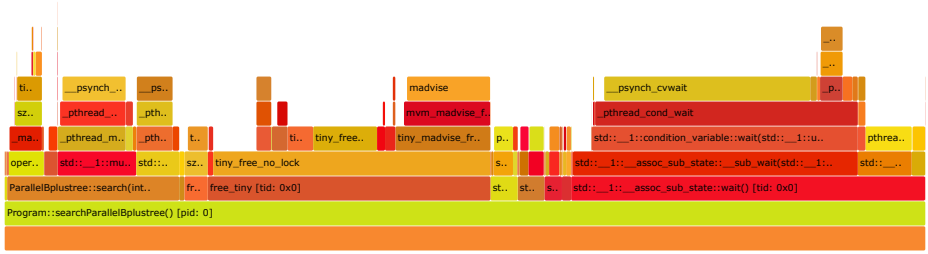
The parallel B^+-tree variant selected for profiling here, was the low performing without Bloom filters single key operation mode variant:

```
$ ./optimized --test search --bloom-disable --threads 2
```

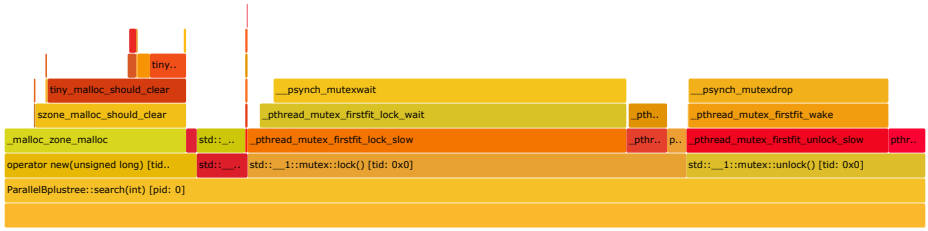
Main executing thread

The main executing thread's profiled results are presented in Figure 4.5 and Figure 4.6. A first observation is that during this run, this thread spends its majority of time waiting. The top 10 nodes sorted by cumulative weight presented in Figure 4.6b shows that `std::__1::__assoc_sub_stat::wait` clocks in at 38.45%. Waiting should of course make up most of this threads time if the number of operations pushed makes the thread pool's task queue grow. The other main portions seen in Figure 4.5a is the `free_tiny` calls and the `search` operation itself. Freeing a lot of memory is due to `search` having a return type, and the main executing thread collecting these results for simple statistics calculations before termination.

The `ParallelBplustree::search` operation makes up 18.69% cumulatively as seen in Figure 4.6b. Note that its flat weight is 0, which is easily verifiable by taking a look at Figure 4.5b. This flame graph also verifies that locking takes up most of the main executing threads time during the task pushing stage in single key operation mode. Specifically, its the `__psynch_mutex(wait/drop)` seen in Figure 4.6a holding high flat weights during this portions of program execution. The only other stack frame of notable size during the `search` operation, is that of `operator new`, which is caused by a `std::promise` heap allocation, which



(a) Wide view rooted at `Program::searchParallelBplustree`.



(b) Root locked at `ParallelBplustree::search`.

Figure 4.5: Flame graph displaying relevant portions of the main executing thread's stack trace during the search test run.

purpose is to in turn relay the search result back to the caller. The actual push on the thread pool's task queue (implemented by use of `std::deque` [53]) can only be found in 5.4% of the samples, i.e. 5.4% cumulative weigth. In the flame graph with root locked on the `search` operation this is the frame colored red, squezed between `operator new` and `std::__1::mutex::lock`. Ideally, we would of course like to trade some of the lock managment time, reflected in the mutex lock/unlock operations of these results, against time spent on actually pushing task to the thread pool's task queue.

Worker thread

The results obtained for the worker thread examined are presented in Figure 4.7 and Figure 4.8. As can be seen, the total time found in the reports are larger than that of the main executing thread. This is only due to the flame graph and reports in this instance being rooted at the `thread_pool::worker` function. Thus, results from the tree build stage can also be found lurking in these reports. However, it is easy enough to distinguish between the two.

Luckily, most of the worker's time during search performance evaluation is spent inside the actual `search` operation on B^+-trees. The cumulative report found in Figure 4.8b places the figure at 33.65% for `Bplustree::search`, and the

```
Type: cpu
Showing nodes accounting for 5140ms, 76.72% of 6700ms total
Dropped 31 nodes (cum <= 33.50ms)
Showing top 10 nodes out of 54
      flat  flat%   sum%        cum   cum%
    1506ms 22.48% 22.48%     1506ms 22.48%  __psynch_cvwait
     668ms  9.97% 32.45%      668ms  9.97%  __psynch_mutexwait
     645ms  9.63% 42.07%      645ms  9.63%  pthread_mutex_lock
     624ms  9.31% 51.39%      624ms  9.31%  madvise
     541ms  8.07% 59.46%      541ms  8.07%  tiny_free_list_remove_ptr
     319ms  4.76% 64.22%     2022ms 30.18%  tiny_free_no_lock
     261ms  3.90% 68.12%      261ms  3.90%  __psynch_mutexdrop
     206ms  3.07% 71.19%      206ms  3.07%  tiny_free_list_add_ptr
     192ms  2.87% 74.06%      192ms  2.87%  pthread_mutex_unlock
     178ms  2.66% 76.72%     2118ms 31.61%  std::__1::__assoc_sub_state::
    ↪ __sub_wait
```

(a) Entries sorted by flat/own weight.

```
Type: cpu
Showing nodes accounting for 2089ms, 31.18% of 6700ms total
Dropped 31 nodes (cum <= 33.50ms)
Showing top 10 nodes out of 54
      flat  flat%   sum%        cum   cum%
         0     0%     0%     6700ms   100%  Program::searchParallelBplustree [pid:
    ↪  0]
         0     0%     0%     2576ms 38.45%  std::__1::__assoc_sub_state::wait [tid
    ↪ : 0x0]
     178ms  2.66%  2.66%     2118ms 31.61%  std::__1::__assoc_sub_state::
    ↪ __sub_wait
         0     0%  2.66%     2056ms 30.69%  free_tiny [tid: 0x0]
     319ms  4.76%  7.42%     2022ms 30.18%  tiny_free_no_lock
       9ms  0.13%  7.55%     1936ms 28.90%  std::__1::condition_variable::wait
      67ms  1.00%  8.55%     1913ms 28.55%  _pthread_cond_wait
    1506ms 22.48% 31.03%     1506ms 22.48%  __psynch_cvwait
         0     0% 31.03%     1252ms 18.69%  ParallelBplustree::search [tid: 0x0]
      10ms  0.15% 31.18%     1181ms 17.63%  std::__1::mutex::lock
```

(b) Entries sorted by cumulative weight.

Figure 4.6: Top 10 nodes found in the reports on the main executing thread's stack trace during the search test run.

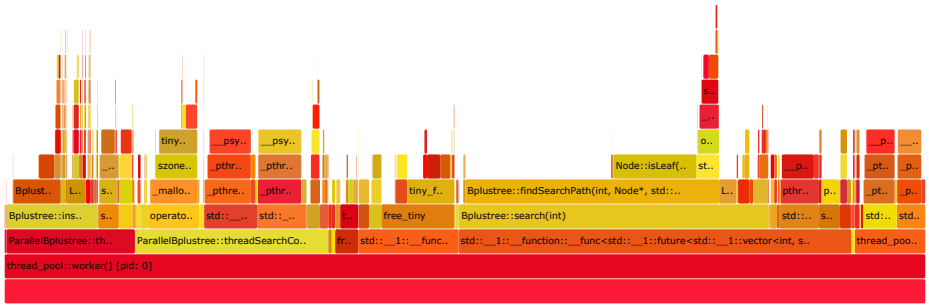


Figure 4.7: Flame graph displaying relevant portions of a worker thread's stack trace during the search test run.

top flat weight, seen in Figure 4.8a, is that of `Bplustree::findSearchPath` with 18.59%. Time spent elsewhere mainly concerns the `ParallelBplustree::threadSearchCoordinator` function, wherein time expenditures are spread out similar to what was found during examination of the main executing thread in Figure 4.5. That is, whereas the `ParallelBplustree::threadSearchCoordinator` function accounts for 21.01% cumulatively, the push operation on the thread pool's task queue concerning specific B$^+$-tree searches, has a measly cumulative weight at 0.52%. Otherwise, time at this point is mainly spent dealing with lock managments. Since Bloom filters are disabled, the samples collected that identify locking at this stage are first and foremost connected to the thread pool's task queue.

4.7.3 Insert performance run

The high performing parallel B$^+$-tree variant profiled, was the without Bloom filters batch operation mode variant:

```
$ ./optimized --test insert --batch --bloom-disable --threads 2
```

Main executing thread

The flame graph presenting the main executing thread during this run can be found in Figure 4.9. It clearly shows that the main executing thread spends close to all its time waiting for the thread pool to finish the insert operations. Taking a look at the reports found in Figure 4.10; Figure 4.10b places the cumulative percentage at 97.50% for the `ParallelBplustree::waitForWorkToFinish` method. At the same time, all flat weights registering more than 0 ms found in Figure 4.10a are associated with this waiting. In fact, no stack trace contains `ParallelBplustree::insert`, i.e. the pushing stage, at the sampling rate recorded at. This is not to surprising as the primary results presented during section 4.3 contains time

```
Type: cpu
Showing nodes accounting for 10453ms, 66.76% of 15657ms total
Dropped 72 nodes (cum <= 78.28ms)
Showing top 10 nodes out of 62
      flat  flat%   sum%        cum   cum%
    2910ms 18.59% 18.59%     5128ms 32.75%  Bplustree::findSearchPath
    1678ms 10.72% 29.30%     1678ms 10.72%  Node::isLeaf const
    1257ms  8.03% 37.33%     1257ms  8.03%  __psynch_mutexdrop
    1248ms  7.97% 45.30%     1248ms  7.97%  __psynch_mutexwait
     740ms  4.73% 50.03%      740ms  4.73%  __psynch_cvbroad
     701ms  4.48% 54.51%     1133ms  7.24%  tiny_malloc_should_clear
     573ms  3.66% 58.17%      573ms  3.66%  pthread_mutex_lock
     491ms  3.14% 61.30%      491ms  3.14%  tiny_free_list_add_ptr
     448ms  2.86% 64.16%     1370ms  8.75%  tiny_free_no_lock
     407ms  2.60% 66.76%     1717ms 10.97%  free_tiny
```

(a) Entries sorted by flat/own weight.

```
Type: cpu
Showing nodes accounting for 3.50s, 22.34% of 15.66s total
Dropped 72 nodes (cum <= 0.08s)
Showing top 10 nodes out of 62
      flat  flat%   sum%        cum   cum%
         0     0%     0%     15.66s   100%  thread_pool::worker [pid: 0]
         0     0%     0%      6.68s 42.68%  std::__1::__function::__func::operator
    ↪  [tid: 0x0]
     0.07s  0.46%  0.46%      5.27s 33.65%  Bplustree::search
     2.91s 18.59% 19.05%      5.13s 32.75%  Bplustree::findSearchPath
         0     0% 19.05%      3.29s 21.01%  ParallelBplustree::
    ↪ threadSearchCoordinator [tid: 0x0]
         0     0% 19.05%      2.18s 13.94%  ParallelBplustree::threadInsert [tid:
    ↪ 0x0]
     0.02s  0.13% 19.18%      2.02s 12.93%  std::__1::mutex::lock
     0.41s  2.60% 21.78%      1.72s 10.97%  free_tiny
         0     0% 21.78%      1.71s 10.90%  std::__1::__function::__func::
    ↪ destroy_deallocate [tid: 0x0]
     0.09s  0.56% 22.34%      1.71s 10.89%  operator new(unsigned long)
```

(b) Entries sorted by cumulative weight.

Figure 4.8: Top 10 nodes found in the reports on a worker thread's stack trace during the search test run.

expenditures for the configuration in question at 0 ms. Of course, spending little to no time distributing a few tasks makes this configuration get off to a good start.

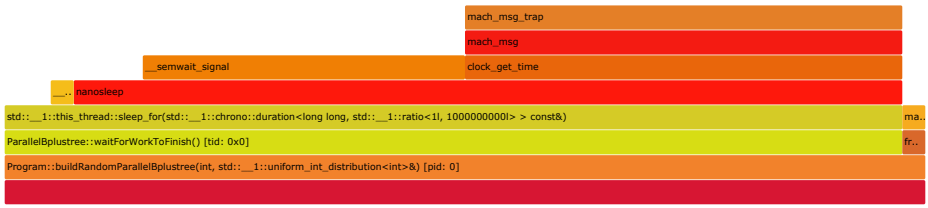


Figure 4.9: Flame graph displaying relevant portions of the main executing thread's stack trace during the insert test run.

Worker thread

The flame graph found in Figure 4.11, depicting how the worker thread spends it time, at a glance reveals that the actual insert operations are in focus. Cumulatively the most interesting figures are 97.08% spent at `ParallelBplustree::threadInsert` and 96.30% spent at `Bplustree::insert`. These numbers can be found in Figure 4.12b. Flat weight consumption displayed in Figure 4.12a shows that it is the tree traversal by `Bplustree::findSearchPath` taking up the most amount of time.

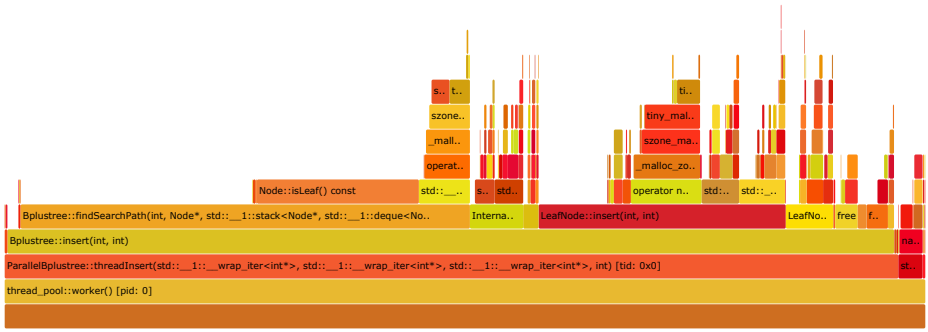


Figure 4.11: Flame graph displaying relevant portions of a worker thread's stack trace during the insert test run.

When comparing this configuration and the way it works to that of subsection 4.7.2, it becomes clear that avoiding locking, and creation and managment of multiple times more tasks are the causes for the performance difference observed. The operation types are of course different, but they both interact with

```
Type: cpu
Showing nodes accounting for 40ms, 100% of 40ms total
      flat  flat%   sum%        cum   cum%
      19ms 47.50% 47.50%       19ms 47.50%  mach_msg_trap
      15ms 37.50% 85.00%       15ms 37.50%  __semwait_signal
       3ms  7.50% 92.50%       36ms 90.00%  nanosleep
       2ms  5.00% 97.50%       39ms 97.50%  std::__1::this_thread::sleep_for
       1ms  2.50%   100%        1ms  2.50%  madvise
         0     0%   100%       39ms 97.50%  ParallelBplustree::waitForWorkToFinish
    ↪  [tid: 0x0]
         0     0%   100%       40ms   100%  Program::buildRandomParallelBplustree
    ↪ [pid: 0]
         0     0%   100%       19ms 47.50%  clock_get_time
         0     0%   100%        1ms  2.50%  free_large [tid: 0x0]
         0     0%   100%       19ms 47.50%  mach_msg
```

(a) Entries sorted by flat/own weight.

```
Type: cpu
Showing nodes accounting for 40ms, 100% of 40ms total
      flat  flat%   sum%        cum   cum%
         0     0%     0%       40ms   100%  Program::buildRandomParallelBplustree
    ↪ [pid: 0]
         0     0%     0%       39ms 97.50%  ParallelBplustree::waitForWorkToFinish
    ↪  [tid: 0x0]
       2ms  5.00%  5.00%       39ms 97.50%  std::__1::this_thread::sleep_for
       3ms  7.50% 12.50%       36ms 90.00%  nanosleep
         0     0% 12.50%       19ms 47.50%  clock_get_time
         0     0% 12.50%       19ms 47.50%  mach_msg
      19ms 47.50% 60.00%       19ms 47.50%  mach_msg_trap
      15ms 37.50% 97.50%       15ms 37.50%  __semwait_signal
         0     0% 97.50%        1ms  2.50%  free_large [tid: 0x0]
       1ms  2.50%   100%        1ms  2.50%  madvise
```

(b) Entries sorted by cumulative weight.

Figure 4.10: Top 10 nodes found in the reports on the main executing thread's stack trace during the insert test run.

the thread pool. However, it is not without downside that the insert configuration is able to excel at inserting. As presented and discussed before, the configuration responsible for these results also suffers quite severly during `search` and `update` when compared to the baseline.

```
Type: cpu
Showing nodes accounting for 1293ms, 77.15% of 1676ms total
Dropped 30 nodes (cum <= 8.38ms)
Showing top 10 nodes out of 39
      flat  flat%   sum%        cum   cum%
     422ms 25.18% 25.18%      818ms 48.81%  Bplustree::findSearchPath
     296ms 17.66% 42.84%      296ms 17.66%  Node::isLeaf const
     139ms  8.29% 51.13%      229ms 13.66%  tiny_malloc_should_clear
     126ms  7.52% 58.65%      451ms 26.91%  LeafNode::insert
      66ms  3.94% 62.59%       66ms  3.94%  tiny_size
      65ms  3.88% 66.47%       69ms  4.12%  tiny_malloc_from_free_list
      59ms  3.52% 69.99%      241ms 14.38%  std::__1::vector::insert
      54ms  3.22% 73.21%      132ms  7.88%  free_tiny
      33ms  1.97% 75.18%       33ms  1.97%  _platform_memmove$VARIANT$Haswell
      33ms  1.97% 77.15%       33ms  1.97%  small_malloc_should_clear
```

(a) Entries sorted by flat/own weight.

```
Type: cpu
Showing nodes accounting for 0.98s, 58.59% of 1.68s total
Dropped 30 nodes (cum <= 0.01s)
Showing top 10 nodes out of 39
      flat  flat%   sum%        cum   cum%
         0     0%     0%      1.68s   100%  thread_pool::worker [pid: 0]
         0     0%     0%      1.63s 97.08%  ParallelBplustree::threadInsert [tid:
    ↪ 0x0]
     0.02s  1.19%  1.19%      1.61s 96.30%  Bplustree::insert
     0.42s 25.18% 26.37%      0.82s 48.81%  Bplustree::findSearchPath
     0.13s  7.52% 33.89%      0.45s 26.91%  LeafNode::insert
     0.01s  0.89% 34.79%      0.33s 19.51%  operator new(unsigned long)
     0.03s  1.73% 36.52%      0.31s 18.32%  _malloc_zone_malloc
     0.30s 17.66% 54.18%      0.30s 17.66%  Node::isLeaf const
     0.01s  0.89% 55.07%      0.28s 16.41%  szone_malloc_should_clear
     0.06s  3.52% 58.59%      0.24s 14.38%  std::__1::vector::insert
```

(b) Entries sorted by cumulative weight.

Figure 4.12: Top 10 nodes found in the reports on a worker thread's stack trace during the insert test run.

Chapter 5

Conclusion and Future work

This chapter concludes the master's thesis by summarizing its contents. In doing so, the key findings are reiterated, and the most probable causes for the findings are stated. In addition to the summarization concluding the master's thesis, some future work stemming from the insights gained is listed.

5.1 Conclusion

This master's thesis was motivated by the modern hardware architecture's demand on horizontally scalable designs and implementations. DBMSs, being no exception in having to adapt to the new and ever-changing computer landscape, have to keep developing their systems. Index structures are utmost important when the DBMSs quickly and efficiently shall retrieve data records.

In this master's thesis the classical B^+-tree index structure was first implemented. Then, this implementation was used as the core of a B^+-tree variation's design and implementation. The B^+-tree variation that has been designed and implemented, most importantly parallelizes the already established B^+-tree's access methods. In search of an effective and usable parallel B^+-tree, the thread pool design pattern was employed, and Bloom filters used in some configurations. Two versions for each method of operation, termed single key operation mode and batch operation mode, were implemented such that the effect of batch processing also could be examined. Then the parallel B^+-tree implementation's potential was evaluated by obtaining throughput performance measurements on all operations, and operation modes, over a varied set of the parallel B^+-tree's configuration possibilites. Furthermore, profiling of both low and high performing runs was carried out, to gain further insight into when and why the parallel implementation scored better or worse than the baseline.

Looking at `insert`, throughput results beating the baseline by up to 10× were obtained by the parallel B$^+$-tree configuration which batches operations and does not make use of Bloom filters. Performance measurements regarding `search` largely tipped in favor of the baseline B$^+$-tree, but competitive performance was observed at the extreme ends of variable values tested. Once again, batching with Bloom filters disabled overall performed best of the parallel configurations, but utilizing such filters yielded a more leveled performance over the range of variables and values tested for most series of runs. As `update` requires a `search` and change of value, the parallel implementation must do more managment and careful handling to remain consistent. Thus, the results obtained here for the parallel configurations shows a resemblance to `search` with regards to development on variable value ranges, but throughput is lower than those of `search`, and baseline performs better than all configurations tested. The best overall parallel B$^+$-tree `update` results were obtained by the single key with Bloom filters operation mode variant. Concerning `delete`, baseline beating results were obtained by the batching without Bloom filters variant during some configuations, whereas the other variations underperformed the baseline.

It was discovered that the variables, whose impact on all operations and operation modes were studied, exhibited recognizable patterns in the throughput measurements, accross the different operations and configurations examined. Most notably, for each of the variables and their respective ranges of values tested on; increasing the value of `order` generally improves throughput, if it is this variable's value which imposes the current bottleneck of the configuration. Of course, there is an upper bound on the achievable throughput by adjusting the value of `order`, largely determined by the computer's cache layout, due to the data structure being in-memory. The variables `numThreads` and `numTrees` have a complex codependent relationship affected by the way workload is split, which makes itself known in the throughput measurements obtained. When taking all the operations results into consideration, it is found that increasing the value of `numThreads` until the number of computer cores is reached (and sometimes until the maximum number of hyper-threading supported threads is reached), in most cases positivly affects throughput during batch operation mode. As for `numTrees`, a proportional relationship is found between this variable and `insert` throughput, on parts of the range of values for all parallel B$^+$-tree variations. Whereas, for all other operations, this relationship is inversely proportional, with the parallel without Bloom filters variant being the only exception during measurements of the `remove` operation.

Profiling evidently revealed that the biggest challenge when aiming for high parallel throughput performance is the way in which to generate and distribute tasks. In many cases the thread pool's task queue becomes a bottleneck when partitioning the workload in too many tasks, as is done during single key operation mode. In order to leverage the parallel processing capabilities effectively, the tasks created and pushed to the thread pool's task queue have to be worthwhile, i.e. the time it takes to carry out a task must be costly to the degree that, the cost of task creation and managment becomes negligible. Only then, if the degree of

parallelism is sufficient, will the parallel implementation have a chance at beating the baseline.

In conclusion, this master's thesis shows that the B$^+$-tree index structure can see tremendous improvements in throughput, if adequately adapting to the modern hardware architecture, through means of employing horizontally scalable parallelization of operations. But doing so is hard, and it is the delicate nature of parallel programming that make it so. This makes a cost-benefit analysis applicable before execution, to see if parallelization of either entire operations or only certain parts should at all be used, at a per scenario basis.

5.2 Future work

This master's thesis only scratches the surface of what it takes to fully parallelize an index structure to be used in a DBMS. As for the specific parallel design and implementation presented, some of the most notable future work involves:

- Implementing scan.
- Change storage format to a more rigorous block-and-post storage format.
- Experiment with no-interrupt thread pool task pushing and fine-tuning of sleep parameter used by worker threads.
- Explore how different storage containers affect throughput of operations.
- Apply locking at a lower level.
- Implement the parallel B$^+$-tree in a DBMS.

Other interesting future work in a broader view concerns redesign and implementation of other index structures in a manner that allows concurrent and parallel operations. Some attempts at such work were already listed in section 2.4, but there are problems that needs to be addressed here as well. Taking a look at parallelizing some of the other classical index structures in a pure way, such as for example the R-tree and Hash index, would in its own right be interesting.

Bibliography

[1] J. Sewall, J. Chhugani, C. Kim, N. Satish, and P. Dubey, "PALM: Parallel Architecture-Friendly Latch-Free Modifications to B+ Trees on Many-Core Processors", *Proc. VLDB Endow.*, vol. 4, no. 11, pp. 795–806, Aug. 2011, ISSN: 2150-8097. DOI: 10.14778/3402707.3402719. [Online]. Available: https://doi.org/10.14778/3402707.3402719.

[2] P. L. Lehman and s. B. Yao, "Efficient Locking for Concurrent Operations on B-Trees", *ACM Trans. Database Syst.*, vol. 6, no. 4, pp. 650–670, Dec. 1981, ISSN: 0362-5915. DOI: 10.1145/319628.319663. [Online]. Available: https://doi.org/10.1145/319628.319663.

[3] J. J. Levandoski, D. Lomet, and S. Sengupta, "The Bw-Tree: A B-tree for new hardware platforms", *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pp. 302–313, 2013.

[4] Z. Wang, A. Pavlo, H. Lim, V. Leis, H. Zhang, M. Kaminsky, and D. G. Andersen, "Building a Bw-Tree Takes More Than Just Buzz Words", in *Proceedings of the 2018 International Conference on Management of Data*, ser. SIGMOD '18, Houston, TX, USA: Association for Computing Machinery, 2018, pp. 473–488, ISBN: 9781450347037. DOI: 10.1145/3183713.3196895. [Online]. Available: https://doi.org/10.1145/3183713.3196895.

[5] Y. Mao, E. Kohler, and R. T. Morris, "Cache Craftiness for Fast Multicore Key-Value Storage", in *Proceedings of the 7th ACM European Conference on Computer Systems*, ser. EuroSys '12, Bern, Switzerland: Association for Computing Machinery, 2012, pp. 183–196, ISBN: 9781450312233. DOI: 10.1145/2168836.2168855. [Online]. Available: https://doi.org/10.1145/2168836.2168855.

[6] G. E. Moore, "Cramming more components onto integrated circuits", *Electronics*, vol. 38, no. 8, 1965. [Online]. Available: https://newsroom.intel.com/wp-content/uploads/sites/11/2018/05/moores-law-electronics.pdf.

[7] M. M. Waldrop, "The chips are down for Moore's law", *Nature*, vol. 530, pp. 144–147, 2016. DOI: 10.1038/530144a. [Online]. Available: https:

//www.nature.com/news/the-chips-are-down-for-moore-s-law-1.19338.

[8] E. H. Trøen, "Utilizing parallel processing capabilities during operations on B^+-trees", unpublished: available upon request, 2021.

[9] R. Elmasri and S. B. Nevathe, *Fundamentals of Database Systems, 7th Edition*. Pearson, 2016, pp. 601, 622–630, 633–634.

[10] R. Bayer and E. M. Mccreight, "Organization and Maintenance of Large Ordered Indexes", *Acta Inf.*, vol. 1, no. 3, pp. 173–189, Sep. 1972, ISSN: 0001-5903. DOI: 10.1007/BF00288683. [Online]. Available: https://doi.org/10.1007/BF00288683.

[11] D. E. Knuth, *The art of computer programming, Second Edition*. Addison-Wesley, 1998, vol. Volume 3 / Sorting and Searching, p. 483.

[12] A. Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching", in *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '84, Boston, Massachusetts: Association for Computing Machinery, 1984, pp. 47–57, ISBN: 0897911288. DOI: 10.1145/602259.602266. [Online]. Available: https://doi.org/10.1145/602259.602266.

[13] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, *The Log-Structured Merge-Tree (LSM-Tree)*, 1996.

[14] R. Thelin. (2020). A Tutorial on Modern Multithreading and Concurrency in C++, [Online]. Available: https://www.educative.io/blog/modern-multithreading-and-concurrency-in-cpp (visited on 04/01/2022).

[15] C. Wilson. (2019). Multithreading and Concurrency Fundamentals, [Online]. Available: https://www.educative.io/blog/multithreading-and-concurrency-fundamentals (visited on 04/01/2022).

[16] B. Caulfield. (2009). What's the Difference Between a CPU and a GPU?, [Online]. Available: https://blogs.nvidia.com/blog/2009/12/16/whats-the-difference-between-a-cpu-and-a-gpu/ (visited on 04/01/2022).

[17] Oracle. (2010). Eliminating Performance Bottlenecks, [Online]. Available: https://docs.oracle.com/cd/E18283_01/server.112/e10578/tdpdw_perform.htm (visited on 04/01/2022).

[18] Oracle. (2016). Optimizing Performance by Creating Indexes in Parallel, [Online]. Available: https://docs.oracle.com/database/121/VLDBG/GUID-5A52793F-15AE-4C86-B8A4-4D7965575BB0.htm#VLDBG1537 (visited on 04/01/2022).

[19] PostgreSQL. (2020). CREATE INDEX, [Online]. Available: https://www.postgresql.org/docs/13/sql-createindex.html (visited on 04/01/2022).

[20] W. Stallings, *Operating systems: internals and design principles, 9th global edition*. Pearson, 2018.

[21] A. Suleman. (2011). Parallel Programming: When Amdahl's law is inapplicable?, [Online]. Available: https://archive.ph/20130414224506/http://www.futurechips.org/thoughts-for-researchers/parallel-programming-gene-amdahl-said.html (visited on 04/01/2022).

[22] D. W. Edsger, *Cooperating sequential processes*. Technological University Eindhoven, 1965.

[23] C. A. R. Hoare, "Monitors: An operating system structuring concept", *Commun. ACM*, vol. 17, no. 10, pp. 549–557, Oct. 1974, ISSN: 0001-0782. DOI: 10.1145/355620.361161. [Online]. Available: https://doi.org/10.1145/355620.361161.

[24] B. W. Lampson and D. D. Redell, "Experience with processes and monitors in mesa", *Commun. ACM*, vol. 23, no. 2, pp. 105–117, Feb. 1980, ISSN: 0001-0782. DOI: 10.1145/358818.358824. [Online]. Available: https://doi.org/10.1145/358818.358824.

[25] V. Leis, A. Kemper, and T. Neumann, "The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases", 2013, pp. 38–49.

[26] V. Leis, F. Scheibner, A. Kemper, and T. Neumann, "The ART of Practical Synchronization", in *Proceedings of the 12th International Workshop on Data Management on New Hardware*, ser. DaMoN '16, San Francisco, California: Association for Computing Machinery, 2016, ISBN: 9781450343190. DOI: 10.1145/2933349.2933352. [Online]. Available: https://doi.org/10.1145/2933349.2933352.

[27] T. Crain, V. Gramoli, and M. Raynal, "No Hot Spot Non-Blocking Skip List", in *Proceedings of the 2013 IEEE 33rd International Conference on Distributed Computing Systems*, ser. ICDCS '13, USA: IEEE Computer Society, 2013, pp. 196–205, ISBN: 9780769550008. DOI: 10.1109/ICDCS.2013.42. [Online]. Available: https://doi.org/10.1109/ICDCS.2013.42.

[28] E. H. Trøen. (2022). thesis, [Online]. Available: https://github.com/eskildht/thesis (visited on 02/08/2022).

[29] ISO/IEC. (2020). Working Draft, Standard for Programming Language C++, [Online]. Available: http://open-std.org/jtc1/sc22/wg21/docs/papers/2020/n4868.pdf (visited on 02/08/2022).

[30] isocpp. (2020). Current Status, [Online]. Available: https://isocpp.org/std/status (visited on 02/15/2022).

[31] J. Ryfetten, "High Performace B-trees for Modern Hardware", unpublished: available upon request, 2020.

[32] cppreference. std::thread, [Online]. Available: https://en.cppreference.com/w/cpp/thread/thread (visited on 02/12/2022).

[33] FreeBSD. FreeBSD Manual Pages - pthread, [Online]. Available: https://www.freebsd.org/cgi/man.cgi?query=pthread (visited on 02/08/2022).

[34] cppreference. Thread support library, [Online]. Available: https://en.cppreference.com/w/cpp/thread (visited on 02/12/2022).

[35] R. Ramakrishnan and J. Gehrke, *Database Management Systems: (3rd ed)*. McGraw-Hill, 2003.

[36] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, "An Improved Construction for Counting Bloom Filters", in *Algorithms – ESA 2006*, Y. Azar and T. Erlebach, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 684–695, ISBN: 978-3-540-38876-0. DOI: https://doi.org/10.1007/11841036_6.

[37] S. Sarkar, T. I. Papon, D. Staratzis, and M. Athanassoulis, "Lethe: A tunable delete-aware lsm engine", *CoRR*, vol. abs/2006.04777, 2020. arXiv: 2006.04777. [Online]. Available: https://arxiv.org/abs/2006.04777.

[38] A. Partow. (2000). C++ Bloom Filter Library, [Online]. Available: https://www.partow.net/programming/bloomfilter/index.html (visited on 02/14/2022).

[39] cppreference. std::unique_lock, [Online]. Available: https://en.cppreference.com/w/cpp/thread/unique_lock (visited on 02/15/2022).

[40] cppreference. std::shared_lock, [Online]. Available: https://en.cppreference.com/w/cpp/thread/shared_lock (visited on 02/15/2022).

[41] cppreference. std::shared_mutex, [Online]. Available: https://en.cppreference.com/w/cpp/thread/shared_mutex (visited on 02/15/2022).

[42] B. Shoshany, "A C++17 Thread Pool for High-Performance Scientific Computing", *arXiv e-prints*, arXiv:2105.00613, May 2021. DOI: 10.5281/zenodo.4742687. arXiv: 2105.00613 [cs.DC].

[43] cppreference. std::future, [Online]. Available: https://en.cppreference.com/w/cpp/thread/future (visited on 02/16/2022).

[44] cppreference. std::promise, [Online]. Available: https://en.cppreference.com/w/cpp/thread/promise (visited on 02/18/2022).

[45] I. Corporation. (2022). What Is Hyper-Threading?, [Online]. Available: https://www.intel.com/content/www/us/en/gaming/resources/hyper-threading.html (visited on 04/01/2022).

[46] cppreference. std::uniform_int_distribution, [Online]. Available: https://en.cppreference.com/w/cpp/numeric/random/uniform_int_distribution (visited on 02/28/2022).

[47] cppreference. std::mersenne_twister_engine, [Online]. Available: https://en.cppreference.com/w/cpp/numeric/random/mersenne_twister_engine (visited on 02/28/2022).

[48] B. Gregg. (2020). Flame Graphs, [Online]. Available: https://www.brendangregg.com/flamegraphs.html (visited on 03/12/2022).

[49] google. (2022). pprof, [Online]. Available: https://github.com/google/pprof (visited on 03/12/2022).

[50] A. Inc. (2019). Instruments Help, [Online]. Available: https://help.apple.com/instruments/mac/current/ (visited on 03/12/2022).

[51] google. (2021). instrumentsToPprof, [Online]. Available: https://github.com/google/instrumentsToPprof (visited on 03/12/2022).

[52] F. Geisendörfer. (2022). pprofutils, [Online]. Available: https://github.com/felixge/pprofutils (visited on 03/12/2022).

[53] cppreference. std::deque, [Online]. Available: https://en.cppreference.com/w/cpp/container/deque (visited on 03/15/2022).

Appendix A

Commands to run performance evaluation tests

This appendix lists all commands used to generate the throughput performance results presented in chapter 4. `<range>` should be replaced by a value in the range one wishes to examine. The values of `<range>` used for each command in this thesis is given before the commands.

The output from using `--help` option can be seen in Listing A.1. A description of all available flags and options can be found here.

A.1 Insert performance

A.1.1 Single key

order

```
<range> = [8, 16, 32, 64, 128, 256, 512, 1024]
$ ./optimized --op 5000000 --op-distr-high 5000000 --test insert --tree
basic --order <range>
$ ./optimized --op 5000000 --op-distr-high 5000000 --test insert --tree
parallel --order <range>
$ ./optimized --op 5000000 --op-distr-high 5000000 --test insert --tree
parallel --bloom-disable --order <range>
```

```
$ ./optimized --help
USAGE:
  ./optimized [FLAGS] [OPTIONS] --test <test>

FLAGS:
  --batch                     Enable batching during test and
general build, if --tree option has value parallel
  --bloom-disable             Disable bloom filter usage if --tree
option has value parallel
  --help                      Print this help information
  --show                      Print the tree after build if
--tree-size value <= 1000

OPTIONS:
  --build-distr-high <num>    Highest possible key value during
tree build [default:  1000000]
  --build-distr-low <num>     Lowest possible key value during tree
build [default:  1]
  --op <num>                  Number of operations to perform for
the --test value specified [default:  1000000]
  --op-distr-high <num>       Highest possible key value during
test operation [default:  1000000]
  --op-distr-low <num>        Lowest possible key value during test
operation [default:  1]
  --order <num>               Order of the Bplustree(s) [default:
5]
  --test <test>               The test to carry out [default:  ]
[possible values:  delete, insert, search, update]
  --threads <num>             Number of threads to use in the
thread pool if --tree has value parallel [default:
std::thread::hardware_concurrency()]
  --tree <type>               The tree data structure to create
[default:  parallel] [possible values:  basic, parallel]
  --tree-size <num>           The number of inserts to do during
tree build (overridden by --op if --test has value insert)
[default:  1000000]
  --trees <num>               Number of Bplustrees to use if --tree
option has value parallel [default:
std::thread::hardware_concurrency()]
```

Listing A.1: `./optimized --help`.

numThreads

```
<range> = [1, 4, 12, 16, 20, 24, 32, 35]
$ ./optimized --op 5000000 --op-distr-high 5000000 --test insert --tree
```

```
parallel --order 128 --threads <range>
$ ./optimized --op 5000000 --op-distr-high 5000000 --test insert --tree
parallel --order 128 --bloom-disable --threads <range>
```

numTrees

```
<range> = [8, 16, 24, 32, 40, 48, 56, 64]
$ ./optimized --op 5000000 --op-distr-high 5000000 --test insert --tree
parallel --order 128 --threads 16 --trees <range>
$ ./optimized --op 5000000 --op-distr-high 5000000 --test insert --tree
parallel --order 128 --bloom-disable --threads 16 --trees <range>
```

A.1.2 Batch

order

```
<range> = [8, 16, 32, 64, 128, 256, 512, 1024]
$ ./optimized --op 5000000 --op-distr-high 5000000 --test insert --tree
basic --order <range>
$ ./optimized --op 5000000 --op-distr-high 5000000 --test insert --tree
parallel --batch --order <range>
$ ./optimized --op 5000000 --op-distr-high 5000000 --test insert --tree
parallel --batch --bloom-disable --order <range>
```

numThreads

```
<range> = [1, 4, 12, 16, 20, 24, 32, 35]
$ ./optimized --op 5000000 --op-distr-high 5000000 --test insert --tree
parallel --order 128 --batch --threads <range>
$ ./optimized --op 5000000 --op-distr-high 5000000 --test insert --tree
parallel --order 128 --batch --bloom-disable --threads <range>
```

numTrees

```
<range> = [8, 16, 24, 32, 40, 48, 56, 64]
$ ./optimized --op 5000000 --op-distr-high 5000000 --test insert --tree
parallel --order 128 --batch --threads 16 --trees <range>
$ ./optimized --op 5000000 --op-distr-high 5000000 --test insert --tree
parallel --order 128 --batch --bloom-disable --threads 16 --trees <range>
```

A.2 Search performance

A.2.1 Single key

order

```
<range> = [8, 16, 32, 64, 128, 256, 512, 1024]
$ ./optimized --tree-size 5000000 --build-distr-high 5000000 --op 5000000
--op-distr-high 5000000 --test search --tree basic --order <range>
$ ./optimized --tree-size 5000000 --build-distr-high 5000000 --op 5000000
--op-distr-high 5000000 --test search --tree parallel --order <range>
$ ./optimized --tree-size 5000000 --build-distr-high 5000000 --op 5000000
--op-distr-high 5000000 --test search --tree parallel --bloom-disable
--order <range>
```

numThreads

```
<range> = [1, 4, 12, 16, 20, 24, 32, 35]
$ ./optimized --tree-size 5000000 --build-distr-high 5000000 --op 5000000
--op-distr-high 5000000 --test search --tree parallel --order 1024 -
-threads <range>
$ ./optimized --tree-size 5000000 --build-distr-high 5000000 --op 5000000
--op-distr-high 5000000 --test search --tree parallel --order 1024 -
-bloom-disable --threads <range>
```

numTrees

```
<range> = [8, 16, 24, 32, 40, 48, 56, 64]
$ ./optimized --tree-size 5000000 --build-distr-high 5000000 --op 5000000
--op-distr-high 5000000 --test search --tree parallel --order 1024 -
-threads 32 --trees <range>
$ ./optimized --tree-size 5000000 --build-distr-high 5000000 --op 5000000
--op-distr-high 5000000 --test search --tree parallel --order 1024 -
-bloom-disable --threads 32 --trees <range>
```

A.2.2 Batch

order

```
<range> = [8, 16, 32, 64, 128, 256, 512, 1024]
$ ./optimized --tree-size 5000000 --build-distr-high 5000000 --op 5000000
```

```
--op-distr-high 5000000 --test search --tree basic --order <range>
$ ./optimized --tree-size 5000000 --build-distr-high 5000000 --op 5000000
--op-distr-high 5000000 --test search --tree parallel --batch --order
<range>
$ ./optimized --tree-size 5000000 --build-distr-high 5000000 --op 5000000
--op-distr-high 5000000 --test search --tree parallel --bloom-disable
--batch --order <range>
```

numThreads

```
<range> = [1, 4, 12, 16, 20, 24, 32, 35]
$ ./optimized --tree-size 5000000 --build-distr-high 5000000 --op 5000000
--op-distr-high 5000000 --test search --tree parallel --order 1024 -
-batch --threads <range>
$ ./optimized --tree-size 5000000 --build-distr-high 5000000 --op 5000000
--op-distr-high 5000000 --test search --tree parallel --order 1024 -
-batch --bloom-disable --threads <range>
```

numTrees

```
<range> = [8, 16, 24, 32, 40, 48, 56, 64]
$ ./optimized --tree-size 5000000 --build-distr-high 5000000 --op 5000000
--op-distr-high 5000000 --test search --tree parallel --order 1024 -
-batch --threads 32 --trees <range>
$ ./optimized --tree-size 5000000 --build-distr-high 5000000 --op 5000000
--op-distr-high 5000000 --test search --tree parallel --order 1024 -
-batch --bloom-disable --threads 32 --trees <range>
```

A.3 Update performance

A.3.1 Single key

order

```
<range> = [8, 16, 32, 64, 128, 256, 512, 1024]
$ ./optimized --tree-size 5000000 --build-distr-high 5000000 --op 5000000
--op-distr-high 5000000 --test update --tree basic --order <range>
$ ./optimized --tree-size 5000000 --build-distr-high 5000000 --op 5000000
--op-distr-high 5000000 --test update --tree parallel --order <range>
$ ./optimized --tree-size 5000000 --build-distr-high 5000000 --op 5000000
```

```
--op-distr-high 5000000 --test update --tree parallel --bloom-disable
--order <range>
```

numThreads

```
<range> = [1, 4, 12, 16, 20, 24, 32, 35]
$ ./optimized --tree-size 5000000 --build-distr-high 5000000 --op 5000000
--op-distr-high 5000000 --test update --tree parallel --order 5<range>2
--threads <range>
$ ./optimized --tree-size 5000000 --build-distr-high 5000000 --op 5000000
--op-distr-high 5000000 --test update --tree parallel --order 5<range>2
--bloom-disable --threads <range>
```

numTrees

```
<range> = [8, 16, 24, 32, 40, 48, 56, 64]
$ ./optimized --tree-size 5000000 --build-distr-high 5000000 --op 5000000
--op-distr-high 5000000 --test update --tree parallel --order 5<range>2
--threads <range>2 --trees <range>
$ ./optimized --tree-size 5000000 --build-distr-high 5000000 --op 5000000
--op-distr-high 5000000 --test update --tree parallel --order 5<range>2
--bloom-disable --threads <range>2 --trees <range>
```

A.3.2 Batch

order

```
<range> = [8, 16, 32, 64, 128, 256, 512, 1024]
$ ./optimized --tree-size 5000000 --build-distr-high 5000000 --op 5000000
--op-distr-high 5000000 --test update --tree basic --order <range>
$ ./optimized --tree-size 5000000 --build-distr-high 5000000 --op 5000000
--op-distr-high 5000000 --test update --tree parallel --batch --order
<range>
$ ./optimized --tree-size 5000000 --build-distr-high 5000000 --op 5000000
--op-distr-high 5000000 --test update --tree parallel --bloom-disable
--batch --order <range>
```

numThreads

```
<range> = [1, 4, 12, 16, 20, 24, 32, 35]
$ ./optimized --tree-size 5000000 --build-distr-high 5000000 --op 5000000
--op-distr-high 5000000 --test update --tree parallel --order 5<range>2
--batch --threads <range>
$ ./optimized --tree-size 5000000 --build-distr-high 5000000 --op 5000000
--op-distr-high 5000000 --test update --tree parallel --order 5<range>2
--batch --bloom-disable --threads <range>
```

numTrees

```
<range> = [8, 16, 24, 32, 40, 48, 56, 64]
$ ./optimized --tree-size 5000000 --build-distr-high 5000000 --op 5000000
--op-distr-high 5000000 --test update --tree parallel --order 5<range>2
--batch --threads <range>2 --trees <range>
$ ./optimized --tree-size 5000000 --build-distr-high 5000000 --op 5000000
--op-distr-high 5000000 --test update --tree parallel --order 5<range>2
--batch --bloom-disable --threads <range>2 --trees <range>
```

A.4 Remove performance

A.4.1 Single key

order

```
<range> = [8, 16, 32, 64, 128, 256, 512, 1024]
$ ./optimized --tree-size 5000000 --build-distr-high 5000000 --op 5000000
--op-distr-high 5000000 --test delete --tree basic --order <range>
$ ./optimized --tree-size 5000000 --build-distr-high 5000000 --op 5000000
--op-distr-high 5000000 --test delete --tree parallel --order <range>
$ ./optimized --tree-size 5000000 --build-distr-high 5000000 --op 5000000
--op-distr-high 5000000 --test delete --tree parallel --bloom-disable
--order <range>
```

numThreads

```
<range> = [1, 4, 12, 16, 20, 24, 32, 35]
$ ./optimized --tree-size 5000000 --build-distr-high 5000000 --op 5000000
```

```
--op-distr-high 5000000 --test delete --tree parallel --order 1024 -
-threads <range>
$ ./optimized --tree-size 5000000 --build-distr-high 5000000 --op 5000000
--op-distr-high 5000000 --test delete --tree parallel --order 1024 -
-bloom-disable --threads <range>
```

numTrees

```
<range> = [8, 16, 24, 32, 40, 48, 56, 64]
$ ./optimized --tree-size 5000000 --build-distr-high 5000000 --op 5000000
--op-distr-high 5000000 --test delete --tree parallel --order 1024 -
-threads 32 --trees <range>
$ ./optimized --tree-size 5000000 --build-distr-high 5000000 --op 5000000
--op-distr-high 5000000 --test delete --tree parallel --order 1024 -
-bloom-disable --threads 32 --trees <range>
```

A.4.2 Batch

order

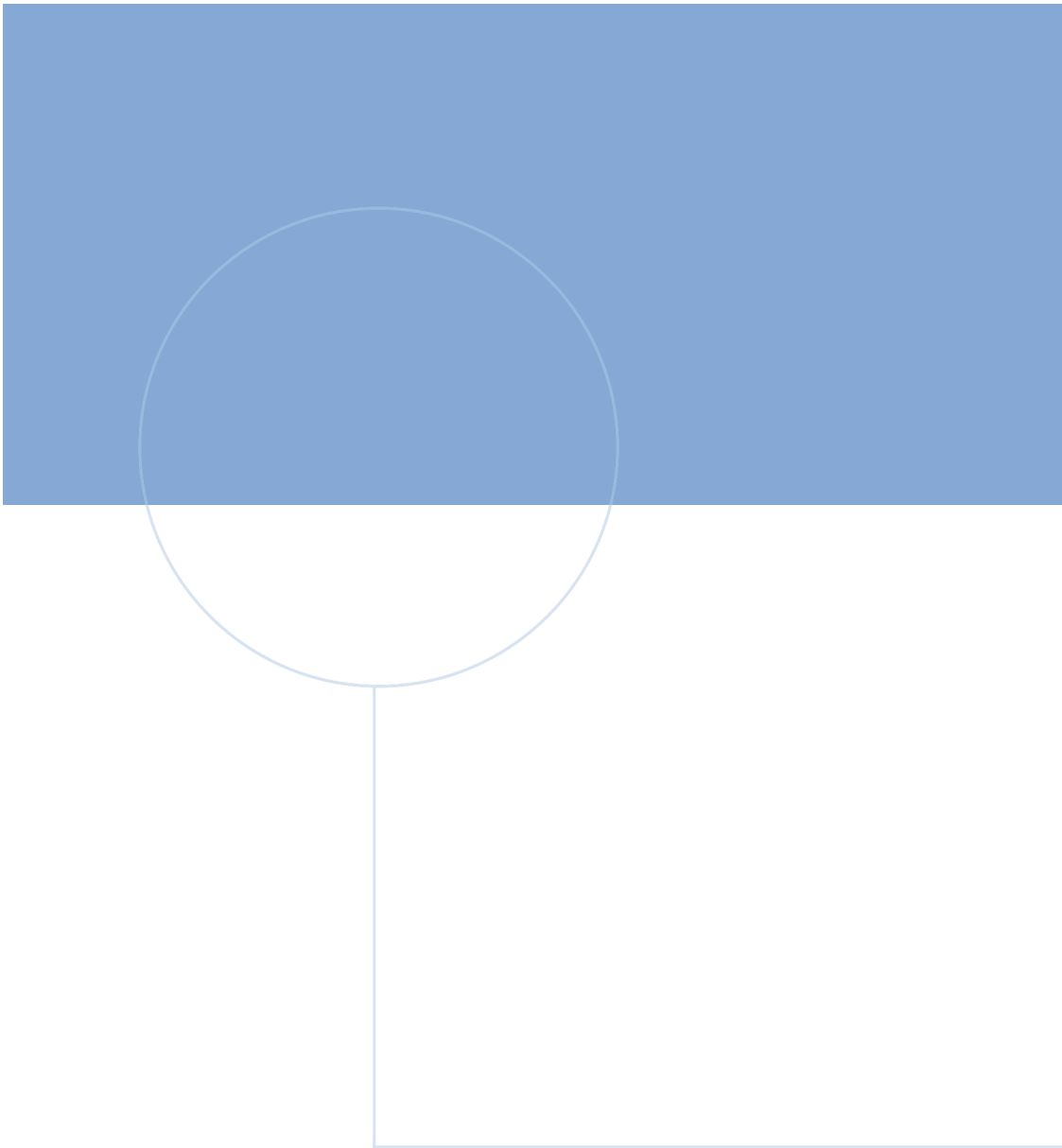
```
<range> = [8, 16, 32, 64, 128, 256, 512, 1024]
$ ./optimized --tree-size 5000000 --build-distr-high 5000000 --op 5000000
--op-distr-high 5000000 --test delete --tree basic --order <range>
$ ./optimized --tree-size 5000000 --build-distr-high 5000000 --op 5000000
--op-distr-high 5000000 --test delete --tree parallel --batch --order
<range>
$ ./optimized --tree-size 5000000 --build-distr-high 5000000 --op 5000000
--op-distr-high 5000000 --test delete --tree parallel --bloom-disable
--batch --order <range>
```

numThreads

```
<range> = [1, 4, 12, 16, 20, 24, 32, 35]
$ ./optimized --tree-size 5000000 --build-distr-high 5000000 --op 5000000
--op-distr-high 5000000 --test delete --tree parallel --order 1024 -
-batch --threads <range>
$ ./optimized --tree-size 5000000 --build-distr-high 5000000 --op 5000000
--op-distr-high 5000000 --test delete --tree parallel --order 1024 -
-batch --bloom-disable --threads <range>
```

numTrees

```
<range> = [8, 16, 24, 32, 40, 48, 56, 64]
$ ./optimized --tree-size 5000000 --build-distr-high 5000000 --op 5000000
--op-distr-high 5000000 --test delete --tree parallel --order 1024 -
-batch --threads 32 --trees <range>
$ ./optimized --tree-size 5000000 --build-distr-high 5000000 --op 5000000
--op-distr-high 5000000 --test delete --tree parallel --order 1024 -
-batch --bloom-disable --threads 32 --trees <range>
```



NTNU

Kunnskap for en bedre verden