PALM: Parallel Architecture-Friendly Latch-Free Modifications to B+ Trees on Many-Core Processors

Jason Sewall Jatin Chhugani Changkyu Kim Nadathur Satish Pradeep Dubey

Parallel Computing Lab
Intel Corporation
Contact: jason.sewall@intel.com

ABSTRACT

Concurrency control on B^+ trees is primarily achieved with latches, but serialization and contention can hinder scalability. As core counts on current processors increase, it is imperative to develop *scalable* latch-free techniques for concurrency control.

We present PALM, a novel technique for performing multiple concurrent queries on in-memory B^+ trees. PALM is based on the Bulk Synchronous Parallel model, which guarantees freedom from deadlocks and race conditions. Input queries are grouped and processed in *atomic batches*, and work proceeds in stages that preclude contention. Transitions between stages are accomplished with scalable *point-to-point communication*. PALM exploits data- and thread-level parallelism on modern many-core architectures, and performs 40M[1] updates/second on trees with 128M keys, and 128M updates/second on trees with 512K keys on the latest CPU architectures. Our throughput is 2.3X–19X that of state-of-the-art concurrent update algorithms on in-memory B^+ trees. PALM obtains close to peak throughput at very low response times of less than 350μs, even for large trees. We also evaluate PALM on the Intel® Many Integrated Core (Intel® MIC) architecture, and demonstrate a speedup of 1.5–2.1X for out-of-cache tree sizes on an Intel® Knights Ferry over a pair of Intel® Xeon® processors DP X5680 (Westmere-EP) in a dual-socket configuration.

1. INTRODUCTION

The B^+ tree is one of the most widely-used data structures in databases [12], and there has been a great deal of work towards achieving the best possible performance with them. A large portion of this work deals concurrency control to enable parallel computation [3, 4, 23, 33, 18, 24, 11].

Much work in this area focuses on disk-based databases; the performance of these systems is generally limited by disk I/O bandwidth. However, as memory capacity has increased dramatically, many database tables and their corresponding index structures now reside completely in main memory — eliminating disk I/O operations. Systems exist now with more than 1TB of memory; this is projected to further increase in next-generation systems [26]. Furthermore, modern processors have increased compute power tremendously by integrating multiple cores on a single chip with widening Single Instruction, Multiple Data (SIMD) units in each core. For the foreseeable future, most gains in computation power will come from expanding amounts of parallelism; scalable parallel algorithms are essential.

Previous work on concurrent traversal and updates to B^+ trees has relied on the use of *latches*[2] to avoid race-conditions and inconsistencies [6]. However, their use forces serialization of some queries and can result in poor scaling, particularly in the presence of skewed input. To skirt these penalties, the use of several distinct classes of latches, along with versioning schemes, have been proposed [6]. Frequently, demanding scalability from latch-based schemes results in complicated code that is difficult to verify and maintain; robust schemes must preclude deadlocks and race-conditions, and considerable effort must be expended to do so.

The acquisition and mere inspection of latches can introduce significant overhead as communication costs — to do so, the memory hierarchy found in many-core architectures must fetch the memory associated with a latch into the cache of the processor the thread is running on, which in turn may require that it be flushed from other caches. Latches that are frequently modified and/or read will 'ping-pong' between caches, incurring latency and bandwidth costs [21].

These complications suggest a need for latch-free algorithms for efficient use of architectural resources (compute/memory bandwidth). In this paper, we present **PALM**, a novel technique for performing multiple read/modify queries on an in-memory B^+ tree in a *bulk synchronous* fashion, **without** the use of latches. Bulk Synchronous Parallel (BSP) algorithms proceed in a sequence of steps, each with a local computation, communication and barrier synchronization phase [34].

In our scheme, we apply the BSP model to concurrent queries on B^+-trees. Input queries are grouped and executed in *atomic batches*; the work in a batch is divided among threads. All read queries occur before modifications to the tree are allowed. We optimize the traditional BSP model for fine-grained queries on many-core processors — in particular, we minimize the use of expensive all-to-all barriers and instead use more efficient point-to-point synchronization between pairs of threads in order to between the stages of the algorithm. Dependencies between queries within a batch are resolved robustly and efficiently. This model ensures that PALM is inherently free of deadlocks and data race conditions, making the code easier to debug and maintain.

[1] The suffix 'K' indicates 'thousand', the suffix 'M' indicates 'million', and the suffix 'B' indicates 'billion.

[2] Following Graefe [11], we make a distinction between *locks* and *latches*; the former is a concept related to the semantics of database transactions, while the latter is used to control updates to in-memory data at a thread-level — these are the '*locks*' referred to outside the database community

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Articles from this volume were invited to present their results at The 37th International Conference on Very Large Data Bases, August 29th - September 3rd 2011, Seattle, Washington.
Proceedings of the VLDB Endowment, Vol. 4, No. 11
Copyright 2011 VLDB Endowment 2150-8097/11/08... $ 10.00.

PALM is readily parallelizable, scalable, and is *asymptotically superior* to sequential B^+ tree modification techniques. Unlike asynchronous techniques using latches, our technique scales on skewed input on even large numbers of cores.

Contributions

- Latch-free batched B^+ tree queries.
- Many-core-friendly — scalable, SIMD- and cache-aware.
- Fastest performance for 512K–1B.
  - 40M updates/second on trees with 128M keys, 128M updates/second on trees with 512K keys on CPUs.
  - 100%-update performance within 1.6X of 100% search.
- Obtain close to peak throughput at quick response times, typically between 60–350μs.
- 2.3X–19X the state of the art on skewed distributions.
- Further 1.5–2.1X on Intel® Knights Ferry.
- Straightforward to test and maintain.

We propose PALM as the algorithm of choice for maintaining modifiable indexes on current and future many-core architectures.

2. RELATED WORK

B-trees [2] were designed to accelerate search queries on disk-based databases while supporting modifications. The importance of performing updates concurrently with searches was recognized early on, and there is a large body of work dealing with concurrency control on B-trees. A number of early papers focused on latch coupling, using read-latches and update-latches to support concurrency and remain free of deadlocks. Bayer and McCreight [3] proposed a number of latching schemes that relied on predicting splits; Bernstein et al. [4] covered a number of protocols for tree latching.

B-link trees, proposed by Lehman and Yao [23], relax the structure of B-trees to allow nodes to expand using high fence keys and pointers to split successors. Srinivasan et al. [33] and Johnson et al. [18] compare various concurrency control schemes for B-trees; both concluded that B-link trees were superior. Follow-up work proposed techniques for B-link node deletion [24], and other modifications to lock coupling and B-link trees have been proposed; the reader is referred to Graefe's comprehensive survey [11].

These techniques rely on latching schemes, which can have high overhead and can limit concurrency in update and search queries — this motivates us to develop a latch-free scheme that efficiently supports a large number of concurrent searches and updates.

As main memory sizes have grown, in-memory databases have become a growing topic of interest. Cache-sensitive search (CSS) trees [28], and cache-sensitive B^+ trees (CSB$^+$-trees) [27] were proposed to take advantage of processor caches. Trees that buffer queries have also been proposed [36]. However, concurrency was not a consideration in these papers.

Cha et al. [6] propose a variation on B-link trees to handle concurrent queries on in-memory databases. Their scheme maintains multiple versions of nodes, and may require multiple retries for retrieval queries to account for nodes that are updated during search. Latches are used to handle concurrent updates.

In recent years, the use of many-core processors for database queries has grown; larger main memories have reduced the impact of I/O bottlenecks and, with the increasing on-die cores, expanded the potential for high-performance databases. A number of recent publications that have revisited various classic database problems from the perspective of many-core computations; aggregation [9], and sorting [30]. Fast architecture-sensitive search trees (FAST), recently proposed by Kim et al. [20] deals exclusively with searches on contemporary highly parallel architectures, and Harizopoulos and Ailamaki [16] proposed the 'Staged Database' concept, where functionally distinct modules of a database are partitioned among multiprocessing resources to improve manageability and performance.

PALM shares an underlying structure with the Bulk Synchronous Parallel (BSP) [34]. The BSP model is popular in many fields, such as graph analytics [25], graphics [17] and high-performance computing [35]. However, traditional BSP programs typically ran on clusters and must do sufficient work to amortize high network communication costs. This results in coarse-grained queries, resulting in unacceptable response times in the context of concurrent B^+-tree queries. Given the advent of many-core processors, we can now exploit fine-grained parallelism due to lower communication costs — resulting in low response times. In this work, we show that we can achieve high throughput with very low response times using a BSP model tuned for many-core processors.

PALM operates on queries in batches; each batch can have update and search queries. Buffered inserts was previously proposed by Arge et al. [1] and analyzed by Graefe [10]. These works were motivated by the desire to avoid disk I/O bottlenecks — our technique focuses on avoiding the penalties of latches in in-memory databases. Chip multiprocessors operate differently from conventional multiprocessor systems, in that inter-thread communication can be done efficiently with on-chip caches [15]. We exploit this property to help threads coordinate stages of the algorithm with a specialized low-overhead point-to-point synchronization mechanism that helps our algorithm to scale with increasing core count.

3. PARALLEL BATCHED TREE QUERIES

3.1 Motivation

PALM operates on in-memory B^+ trees that index a column of a database D by keys from a totally-ordered set $\mathbb{K}$; the tree stores pairs (k, r_k^*). The pointer r_k^* refers to a secondary structure r_k that enumerates the ids of tuples with key k in D — this indirection is typical of databases supporting multiple tuples with the same key.

For a tree T_D indexing a database D, PALM supports three queries with the following semantics:

RETRIEVE (T_D, k)**:** Return r_k, or $\emptyset$ if $k \notin T_D$.

INSERT $(T_D, (k, e))$**:** If $k \in T_D$, append e to r_k; otherwise, add a new $r_k = \{e\}$, and add the new pair (k, r_k^*) to T_D.

DELETE $(T_D, (k, e))$**:** If $k \in T_D$, remove e from r_k, then if $|r_k| = 0$, remove (k, r_k^*) from T_D. If $k \notin T_D$, do nothing.

RETRIEVE is strictly a read query and returns results, while INSERT and DELETE are read/write (modify) queries returning nothing.

Furthermore, no RETRIEVE query has effect upon another RETRIEVE's result, and INSERT and DELETE affect each other's results (and the results of RETRIEVE) only when they use same key. These properties allow O to reordered (see Sec. 3.2.5).

3.1.1 *Asynchronous tree queries*

It can be difficult to effectively operate on data structures across multiple threads. Individual queries often are too little work to benefit from thread-level parallelism — the more common approach to utilizing threads in database indexes is to have each thread process a single query asynchronously.

Validity Assuring that such structures may be read and written by multiple threads properly and efficiently is challenging. By 'properly' we mean those qualities that are concerns for parallel algorithms:

Correctness: Write queries must not interact in such a way as to leave the tree invalid/incorrect.

Serializability: A sequence of queries must return the same results and result in the same D as if run sequentially.

Robustness: The method should preclude deadlock and livelock, or at least detect and address them.

Performance Some asynchronous methods use latches for read and update queries; others need them only for update queries. Such latches always incur performance penalties.
The problem of latch contention — when a thread tries to acquire a latch but cannot, and must wait — is the most easily recognized performance issue with latch. This typically depends on the types of queries, the size of the tree, and the number of threads. As a rule, contention increases with the number of modify queries, and with the number of threads working on the tree. These problems worsen as the number of threads (and concurrent queries) increases, which can hinder the scalability of latch-based methods, particularly when considering multi-socket machines.

Complexity For correctness and performance, asynchronous techniques for tree queries often modify the tree structure to include latches (sometimes several per node) and additional pointers are added to reduce contention. Sophisticated protocols ensure serializability and avoid race conditions/deadlock.

3.1.2 Synchronous tree queries

The difficulties associated with asynchronous, latch-based tree structures — their complexity and performance — suggests that, if possible, latches should be avoided.

Rather than have multiple threads each perform an query on a tree independently and relying on latches to handle communication and correctness, one can imagine a scheme where threads operate cooperatively to perform a series of queries. By using a *synchronous* approach to managing threads working on the tree, we eliminate contention — each tree node that needs to be modified will be implicitly 'owned' by a single thread, and all threads will automatically avoid reading nodes that are being modified.

PALM uses these concepts to execute scalable and robust queries on in-memory B^+ trees; details follow.

3.2 Algorithm

Algorithm 1 Our algorithm for batch queries

PALM (O, T_D, i, t)

    // O are queries, T_D is the tree
    // i is the thread-id, and t is the # of threads
1 $O_i = \text{Partition-Input}(O, i, t)$
2 $L_i = \text{Search}(O_i, T_D)$
3 $\text{Sync}(i, t)$
4 $L'_i = \text{Redistribute-Work}(L_0, \dots, L_{t-1}, i)$
5 $R_i, O'_{L'_i} = \text{Resolve-Hazards}\left(L'_i, O, D\right)$
6 **for** (O_λ, λ) **in** $(O'_{L'_i}, L'_i)$
7     $M_i^1 = M_i^1 \cup \text{Modify-Node}(O_\lambda, \lambda)$
8 $\text{Sync}(i, t)$
9 **for** $d = 1$ **to** $\text{depth}(T_D) - 1$
10     $M_i^{d'} = \text{Redistribute-Work}\left(M_0^d, \dots, M_{t-1}^d, i\right)$
11   **for** (Λ, η) **in** $M_i^{d'}$
12         $M_i^{d+1} = M_i^{d+1} \cup \text{Modify-Node}(\Lambda, \eta)$
13     $\text{Sync}(i, t)$
14 **if** i == 0
15     $\text{Handle-Root}\left(\bigcup M_i^{d+1}, T_D\right)$
16 **return** $R_0, \dots, R_{t-1}$

Our algorithm takes as input an ordered sequence (batch) $O = (o_0, o_1, \dots, o_{K-1})$ of K queries from Sec. 3.1 and performs them simultaneously on an in-memory B^+-tree T_D. It returns the results of any RETRIEVE queries given in O, and T_D is updated by the effects of any modification queries therein. O may be the buffered input from a single source of queries or the aggregated results of multiple sources — these are collected by a single thread in serial before the PALM algorithm begins.

The work is performed cooperatively by t threads, and care is taken to guarantee that (a), the resulting modifications to the tree and underlying database, as well as the results of any RETRIEVE queries in the input, are consistent (in the sense of *serializability*, see Sec. 3.1.1) with the order in O, and (b), that the resulting (modified) tree T'_D is a valid B^+-tree.

PALM avoids latches and contention through two invariants: that the tree is not modified until all reads (searches) have completed, and that each node's modifications are the responsibility of just a single thread. From a high level, the algorithm is as follows:

Stage 1
1. **Divide** tree queries in O among threads.
2. Independently **search** for leaves for each query.

Stage 2
3. **Redistribute work** to ensure no modification contention, and **ensure ordering** of queries.
4. **Modify leaves** independently.

Stage 3
5. Proceed in 'lock-step' up the tree, **modifying internal nodes** and **redistributing work**, up to the root.

Stage 4
6. A single thread **modifies the root**, (if necessary), potentially changing the depth of the tree.

It is convenient to think of the algorithm in terms of these conceptual 'stages'; at any point all threads occupy the same stage, and in so doing, avoid contention issues. See Fig. 1 for a visual description of our technique.

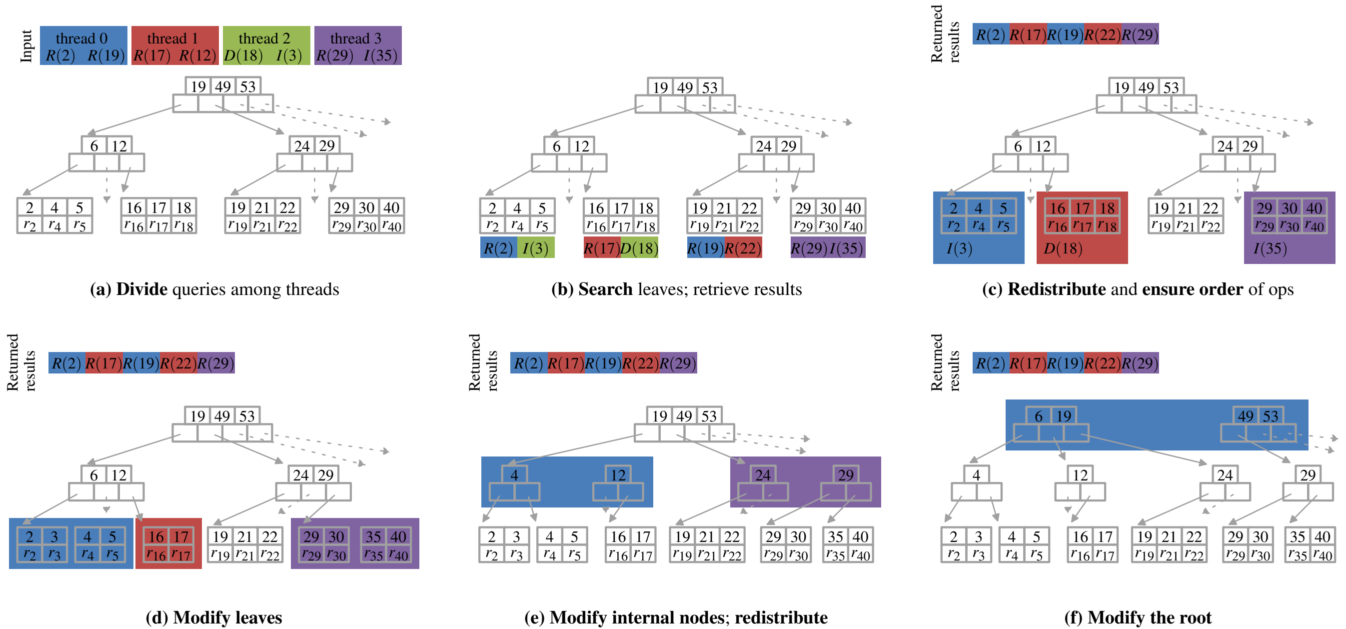
Tree nodes are modified using a generalization of the standard B^+-tree insertion and deletion algorithms; because there are many queries in a batch, it is sometimes necessary to insert or delete multiple items (keys or children) from a tree node. In particular, this can cause nodes to split more than once. Rather than propagate the results of node splits or underflows to parents immediately, the information about nodes to be added or removed is kept in *modification lists* — these are then coalesced at each node when the algorithm proceeds to updating the next level. We describe the details of generalized node modifications in Sec 3.2.3.

See Alg. 1 for a precise description of the overall algorithm; below, we discuss individual sub-procedures:

PARTITION-INPUT This function simply divides the list of queries O among the t threads evenly; any partitioning is suitable, but see Sec. 3.2.5, where we modify the order of O.

SEARCH Since our data structure is a standard B^+-tree, our search procedure is no different from the usual one, except that there are multiple items O_i to search for; we simply iteratively search for each and collect resulting leaf nodes in L_i. We are assured that the results reflect the state of the tree when each query was dispatched, because no modifications to the tree have occurred yet. Search in PALM is able to take advantage of the multiplicity of queries, resulting in gains in performance via architecture-aware optimizations; see Sec. 4.1.

SYNC As presented here, this is a barrier that stops threads from proceeding until all have arrived. In fact, a less stringent level of synchronization can serve that requires less inter-thread communication; see Sec. 3.2.6.



(a) Divide queries among threads

(b) Search leaves; retrieve results

(c) Redistribute and **ensure order** of ops

(d) Modify leaves

(e) Modify internal nodes; **redistribute**

(f) Modify the root

Figure 1: A visual depiction of the PALM algorithm; four threads (their work identified by color here) cooperate to execute 8 queries on a B$^+$-tree. This shows only some nodes in a given tree.

3.2.1 *Cooperation in* REDISTRIBUTE-WORK

In REDISTRIBUTE-WORK, threads re-partition the work at the current level based on the tree nodes to be modified; this ensures that each node is modified by exactly one thread. Each query and modification list affects a single node — the work is implicitly divided based on threads' 'ownership' of tree nodes in each stage.

Each thread i determines which subset L'_i of the leaf nodes in L_i it will operate on. The protocol for determining L'_i for thread i is given by the following:

$$L'_i = \{\lambda \in L_i | \lambda \notin L_j, \forall 0 \leq j < i\} \tag{1}$$

that is, L'_i for a thread i is all leaves that are in L_i that are not in L_j for threads j with index less than i. The procedure to re-partition the work done on internal nodes M^d into $M^{d'}$ during the internal modification steps is the same as that given in Eq. (1); simply substitute $M^{d'}$ for L' and M^d for L. Eq. (1) gives lower-numbered threads priority when multiple threads have updates that must occur on a single node; in general, as tree sizes grow and queries become more uniformly distributed, the likelihood of load imbalance due to this protocol diminishes; see Sec. 3.3.

The key observation here is that there is *no explicit communication*; threads do not message one another about work they are discarding or stealing — each thread determines its own work by inspecting others' and following an implicit protocol. We are assured that the process is immune to race conditions by virtue of synchronization points. In Sec. 3.2.5, we describe how this process can be improved to further minimize communication.

3.2.2 *Ensuring correctness with* RESOLVE-HAZARDS

It is important that any parallel tree scheme obey the *serializability* rule; the results of any RETRIEVE queries returned by our technique, as well as the final, updated database state, must be identical to the results of serially inserting each query in the input.

First, queries on distinct keys are independent of one another with respect to the state of the database D; it is sufficient to examine queries affecting each key independently. Furthermore, all queries on a given key are tied to a single leaf node, and this leaf node will be modified by only a single thread by virtue of the REDISTRIBUTE-WORK function.

Then each leaf λ resolves serializability concerns independently; for each unique key k belonging to λ, we visit the sequence of queries $O_k = (o_j(k,\cdot), o_{j+1}(k,\cdot), \ldots)$ that reference k in the order they appear in O and perform the query described; adding tuples for each INSERT, removing tuples for each DELETE, and, for each RETRIEVE, recording the state of the record in thread i's retrieval buffer R_i (see line 5 of Alg. 1).

The net result is that k may have been added to or removed from D; the T_D must be modified to reflect this — we mark k for insertion (or deletion) to (from) λ in O_λ. After all threads have finished, $\bigcup R_i$ will contain responses to all RETRIEVE queries in O.

3.2.3 *Bulk node modification with* MODIFY-NODE

PALM updates tree nodes using MODIFY-NODE; this can be viewed as a generalization of the standard B$^+$-tree algorithms for key insertion (and deletion) in two aspects. First, that $n \in \mathbb{N}$ insertions to (deletions from) a node can be handled at once, rather than one, and second, that insertion and deletion queries to a single node may be performed together. In particular, *any number of splits* are permitted to accommodate inserted items and to satisfy the tree conditions, but changes are not immediately inserted into the relevant parent node.

There are three distinct outcomes from this function:

Split(s) The node experienced a net gain of elements, and its degree now exceeds MAX-DEGREE; the node is split into two or more nodes, each of which satisfies the MIN-DEGREE requirement. A modification m is returned containing new nodes and indicating the parent node to be updated.

Underflow The node experienced a net loss of elements that put it under MIN-DEGREE; the node will be removed. A modification η is returned that indicates the node to be removed, which parent it should be removed from, and all keys found in descendants of η — these will be re-inserted in the tree after PALM completes.

No external change The node did not change enough to make its degree invalid. No modification is returned.

Algorithm 2 Our algorithm for updating nodes with multiple simultaneous modifications

```
MODIFY-NODE (Λ, η)
   // Λ is sequence of modifications to node η.
   // If η is internal, Λ is a modification list.
   // If η is a leaf, Λ is a series of INSERT and DELETE queries.
 1 E = items(η)
 2 K = ∅
 3 for m in Λ
 4     K = K ∪ orphaned-keys(m)
 5     if class(m) == +
 6         E = E ∪ items(m)
 7     elseif class(m) == −
 8         E = E \ items(m)
 9 if |E| > MAX-DEGREE
10     η, η′, η″, ... = BIG-SPLIT(E)
11     return {+, parent(η), η′, η″, ..., K}
12 elseif |E| < MIN-DEGREE
13     return {−, parent(η), η, K ∪ descendant-keys(E)}
14 else
15     child-ranges(η) = E
16     return {∅, K}
```

MODIFY-NODE returns information about how update each nodes' parent (if at all) in a modification list; as PALM traverses up the tree, modification lists from the previous level are coalesced at each node used in the next step. This function modifies leaf nodes with pairs of key/record-pointers (k, r_k), and modifies internal nodes with modification lists returned from the previous level; see Alg. 2.

The helper function BIG-SPLIT creates one or more new nodes and divides the node items E between the original node and new ones, ensuring that each is consistent with MIN- and MAX-DEGREE.

After processing the highest depth in the tree, there may be outstanding modifications — the root may have split, or may be have been marked for removal. HANDLE-ROOT resolves these queries.

In the case of outstanding splits in the last modification lists, we create a new root node and set its children to be the former root node and the nodes from these splits.

If there are outstanding deletions (i.e., the root node is marked for removal), we proceed based on the root's degree; if it is one, we delete it and promote its single child to the root position, decreasing the depth of the tree. If it is zero, the tree is empty.

3.2.4 Deletions

DELETE queries can remove keys from leaf nodes, and modification lists can remove children from internal nodes. The result is that nodes can underflow; we mark these for removal in successive modification lists. However, when deleting these nodes, we must account for any keys remaining in the node's descendants; we aggregate these in lines 4 and 13 of MODIFY-NODE (Alg. 2) and pass them up the tree through modification lists. After we have finished PALM where O contained DELETE queries, we re-run PALM to INSERT these orphaned keys.

3.2.5 Reducing communication costs

If the batch of queries O is *sorted* by the keys k before partitioning work, we may reduce the effort to determine threads' work. When O is sorted, the leaves $L_i = \{\lambda_0^i, \lambda_1^i, \ldots\}$ collected in search will also ordered (from left to right in the tree): within each L_i such that $\lambda_l^i < \lambda_{l+1}^i$, and also across threads:

$$\lambda^i \leq \lambda^j \qquad \forall \lambda^i \in L_i, \lambda^j \in L_j, i < j \tag{2}$$

Threads still follow the protocol defined in Eq. (1): each node belongs to the lowest-numbered thread that encountered it during search. However, now each thread inspects *adjacent* threads in order, potentially terminating early.

The L_i of each thread i contains zero or more distinct, ordered nodes $\{\lambda_0^i, \lambda_1^i, \ldots\}$; thread i must determine if some other thread $h < i$ has λ_0^i — if so, λ_0^i is discarded from L_i. Then, if any $\lambda_{L_i-1}^i \in L_i$ remains (the last ordered element, if $|L_i| > 0$), i must search $j > i$ for any $\lambda_l^j = \lambda_{L_i-1}^i$, and collect that work. Each of these searches has the potential to terminate early — as soon as some $\lambda_k^h \neq \lambda_l^i$ is encountered for the $h < i$ search, and as soon as $\lambda_l^j \neq \lambda_{L_i-1}^i$ is encountered in the $j > i$ search.

The procedure is the same for modifications to internal nodes η in stage 3; by following the above process, these will already be ordered — we need only enforce Eq. (1) to ensure that a single thread is responsible for updating any given η.

In general, sorting O has many practical benefits and can improve performance overall. We discuss these benefits and how sorting may be performed efficiently in Sec. 4.2.2.

3.2.6 Point-to-point synchronization

The SYNC function is used to ensure that threads execute each stage together. With sorted O, we use a synchronization protocol to eliminate all but one global barrier; this is accomplished with repeated *point-to-point* synchronizations. Each thread i communicates only with threads $i-1$ and $i+1$ to determine its work. This is a significant improvement over barriers. See App. A for details.

3.2.7 Transactions

Database systems typically use *transactions* to achieve meaningful reliability and consistency with users' requests. This often takes the form of satisfying ACID (Haerder et al. [14]). PALM deals with trees that *index* databases, rather than databases themselves, and as such does not directly handle transactions; rather, it complements their implementation in a different layer. The result of a RETRIEVE query is a list of tuple ids that refer to database entries; the consumer of this information is now free to access the database itself. In a transactional setting, this may involve consulting and obtaining locks. Similarly, INSERT and DELETE queries will add or remove entries from the index tree, respectively. *Ghost records* [11] can be used to support transaction and locking.

3.3 Load balancing

PALM initially divides work by evenly partitioning the O queries among the t threads. The work in the read-only SEARCH in stage 1 is therefore well balanced between threads.

The node ownership protocol described by Eq. (1) requires that threads redistribute work before updating leaf or internal nodes in stages 2 and 3; here the balance of work between threads depends on the distribution of input O and the size of T_D. Highly skewed input may touch only a few leaves; a pathological example is when all $k_i \in O$ are greater than the largest key k_j in T_D — all of O belongs to a single leaf node.

The contention avoidance scheme described above prevents multiple threads from cooperating on such input, but MODIFY-NODE is inherently more efficient for these inputs. Furthermore, PALM is still able scale on highly skewed by having threads cooperate to update a single node.

3.3.1 Skewed input to MODIFY-NODE

In BIG-SPLIT, the node's original items (I_λ) and the new items (I_{new}) are spread among the new leaves; if O is sorted, as suggested in Sec. 3.2.5, these can be merged in $O\left(|I_\lambda| + |I_{\text{new}}|\right)$ time.

The efficiency of MODIFY-NODE improves with the number of elements W to be inserted to a single leaf at once; with sorted leaves, the W inserts will take $O(W+|I_\lambda|)$ time, as will BIG-SPLIT (should one occur). When a batch of size K inserts go to L leaves, the total cost is $O(K+L|I_\lambda|)$: as L decreases, less computation is required to insert K keys.

Highly skewed input is characterized by high K/L ratios; PALM is capable of handling such data sets efficiently. More details, including analysis for unsorted leaves, can be found in App. C.

3.3.2 Scaling with highly skewed inputs

For highly skewed input, PALM achieves high performance by having all threads cooperatively assist ongoing BIG-SPLITs; the merge process therein can efficiently be done in parallel [8].

4. IMPLEMENTATION

Modern processors use caches to reduce latency; these operate with a certain granularity and fetch fixed-size sequences of memory. It is important to consider the size of *cache lines* when implementing data structures to avoid wasting bandwidth. Nodes in B$^+$ trees should be allocated in memory to fit in a integral number of cache lines, and the choice of degree for the tree should be guided by cache line size; other details about node representation can be found in App. B.

4.1 Hiding latency in SEARCH

When traversing a B$^+$-tree, the lowest levels will generally not be cache-resident, and the latency cost of fetching nodes from main memory to proceed can seriously hinder performance. The batched approach espoused by PALM allows us to use a latency-hiding technique to improve throughput.

From the group of O_i queries assigned to a thread i, a subset Q of them are performed at once — at each level of the tree, each one of Q inspects the current node it is visiting and determines which child it will need to visit next. Rather than proceed to the child directly, a series of prefetch instructions are issued to bring that node into cache. While node is being fetched, i has next search in Q inspect its current node and prefetch the appropriate child, and so on. By the time each of Q has determined the next node to visit, the next node for the first search in Q will be in cache.

Ideally, Q is chosen such that the computation time to find the next child node for all Q is at least as much as the latency of fetching a node from main memory.

A further benefit of sorting the input queries O is that the groups of keys assigned to each thread will generally follow a coherent search path through the tree, reducing demand for bandwidth.

4.2 Utilizing SIMD

PALM uses SIMD instructions available in modern CPUs to improve performance in several substeps:

4.2.1 Sorted and unsorted searches

We search the ranges stored in internal nodes to determine which child node to proceed to next, and these ranges are stored in increasing order. Binary search seems a logical choice from an asymptotic perspective, but it underperforms linear search for small arrays due to branch misprediction.

We accelerate individual searches by comparing a search key k with multiple ranges loaded with SIMD; the results of the comparison are translated through a lookup table and summed to produce the index of the child node that k belongs in. While the (key, rid) pairs in leaf nodes are stored unsorted, a similar SIMD comparison technique (with a different lookup table) can be used to accelerate each individual leaf search as well.

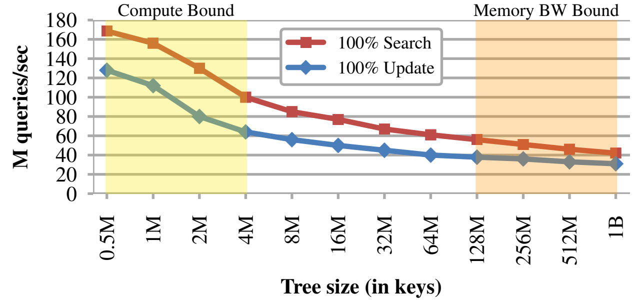


Figure 2: Performance of search and update queries for varying tree sizes (4-byte keys drawn from uniform distributions used.)

4.2.2 Sorting and merging

In Sec. 3.2.5, we propose sorting the input queries O to minimize communication and improve performance, and in Sec. 3.3.2, we discuss how node splits require a sort followed by a merge; these portions of PALM depend on efficient sorting. We use recently proposed SIMD techniques for sorting [30] to handle these sorting and merging problems efficiently.

4.3 Performance model

We have built an accurate analytical model to characterize the performance of PALM. The model computes the number of executed instructions and memory bandwidth for each step, and closely matches the actual run-times (within 5%); this is useful for projecting performance on current and future many-core architectures. For more details, refer to App. C.

5. RESULTS

We have evaluated PALM on the Intel® Xeon® processor DP X5680 (Westmere-EP). Our test platform has 2 sockets (a total of 12 cores running at 3.33 GHz), 96 GB RAM, and runs SusE Enterprise Edition Linux 11. The peak CPU compute power per socket is 150 Gops (300 Gops on 2 sockets) and achievable bandwidth of 44 GBps on 2 sockets.

We initialize our trees (minimum degree 16) by inserting (key, rid) tuples with 4-byte, 8-byte or 16-byte keys, and 8-byte rids, and draw (key, rid) queries from various input distributions — we vary the initial number of tuples in the tree from 512K to 1B. We vary the update ratio of the queries from 0% (corresponding to all search queries) to 100% (corresponding to all updates). We measure performance in millions of queries per second (higher numbers are better); we measure performance on a number of queries 10% of the tree size, which are performed in batches of 8192 queries each.

5.1 Performance evaluation

Fig. 2 shows the performance of search and update queries with varying tree sizes. The tree sizes in the figure show the initial number of tuples in the tree before any queries are handled; the keys used here are 4 bytes long and uniformly distributed; we present the performance of PALM with varying key sizes in App. D.2.

We first note that our search and update performance are **within 1.6X of each other** for the entire tree size range. Since update queries require searching for the key to find the leaf node to update, the gap between search and update performance reflects the overheads of redistributing work to threads and synchronizing threads in addition to modifying leaf and internal nodes. Fig. 2 shows that

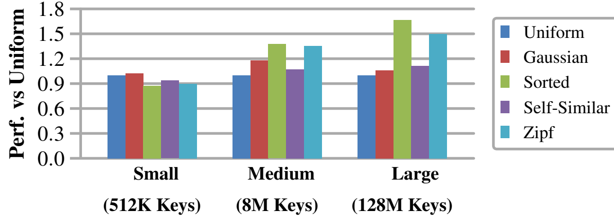


Figure 3: Performance of 100% update queries on small, medium, and large trees of 4-byte keys drawn from various distributions as compared to keys from a uniform distribution.

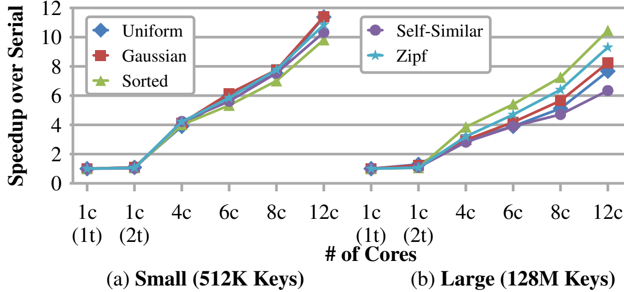


Figure 4: Performance scaling on different input distributions with a query mix consisting of 75% searches and 25% updates into (a) a small tree (512K tuples), and (b) a large tree (128M tuples) (4-byte keys)

our update algorithm has very low overheads in work distribution, synchronization, as well as low costs of modifying the tree nodes. The gap between search and update performance further decreases for larger tree sizes because the depth of the tree increase sand consequently, search takes more time, while the overhead and costs for updates remain nearly constant (modifications to internal nodes are very infrequent). For more on the performance breakdown of our algorithm, refer to App. D.

Both our search and update performance in Fig. 2 are **compute bound for small trees** (512K tuples) since the entire tree fits in the last level cache on the processor. As the number of tuples in the tree increases, the performance becomes gradually bound by memory bandwidth — progressively smaller portions of the tree are cache-resident. For **large trees** with 128M tuples (requiring upwards of 2GB of storage), **the performance is bandwidth bound**. We also compare the throughput of our algorithm to the analytical model developed in Sec. 4.3. Our results match within 1–5% of the predicted values (App. C). For such bandwidth-bound tree sizes, performance will generally be characterized by the depth of search required for each query.

In terms of absolute performance, the **search performance of PALM is only 2X lower than the performance of a state-of-the-art read-optimized FAST tree** [20]. Note that the FAST tree does not support update queries. In order to compare with trees that support updates, we implemented B-link trees [23] using latches and versioning [6] to support search queries. We further introduced SIMD and latency hiding schemes into the implementation for a fair comparison of concurrent performance. The **update performance on PALM is 2.3-19.1X better than the improved B-link algorithm**. We present more details regarding the implementation and results in Sec. 5.3.

5.1.1 Impact of skewed data

We have evaluated PALM with several input distributions, defined in App. D: *uniform*, *Gaussian*, *sorted*, *self-similar*, and *Zipf*.

Fig. 3 shows the results of our evaluation when the algorithm is run using update queries with 24 threads (12 cores, 2-way Simultaneous Multi-Threading (SMT) core) for small, medium, and large

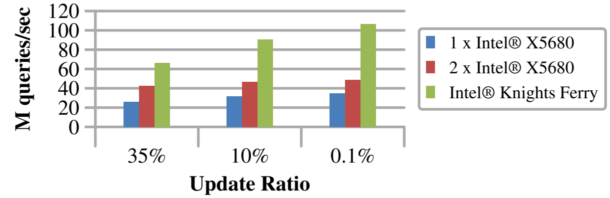


Figure 5: Comparison of query throughput on a 20M key tree for 1– and 2–socket configurations of Intel® Xeon® processors DP X5680 (Westmere-EP) and Intel® Knights Ferry for a variety of update ratios, for 4-byte keys drawn from a uniform distribution,

trees. The baseline for the graph is the performance of uniformly distributed queries. The performance of our algorithm is stable with respect to input variations, and our overall **performance on different distributions is within a factor of 1.6X from the uniform distribution**. A data becomes more skewed, queries repeatedly access the same portions of the tree — this results in smaller working sets and improved performance as cache behavior improves. In particular, the performance of queries from the sorted and Zipf distributions are not bound by memory bandwidth even for large trees. The performance of our scheme **improves with skewed data**.

5.1.2 Performance scaling

Fig. 4 shows our performance scaling from 1 to 12 cores of our dual-socket system for small and large trees. For 1 core, we show the effect of SMT through the ratio of performance on 1 thread to that on 2 threads running on the same core. The performance results for all other number of cores are with SMT turned on. For **small trees**, Fig. 4(a) shows that we obtain **10X–11.6X scaling** from 1 core to 12 cores. Scaling is **uniformly good across different distributions**; load imbalance and synchronization costs are low.

For **large trees**, we obtain **6.5–10.5X scaling on various distributions**. The performance on large trees is determined by the available memory bandwidth, since the working set does not fit in cache — applications generally do not scale once bandwidth has been saturated. Our tests using queries from the sorted and Zipf distributions only visit relatively small portions of the tree, as described earlier — PALM is not bandwidth-bound in these cases, and we achieve scaling of 10.5X and 9.3X for the sorted and Zipf distributions, respectively.

For both small and large trees with various distributions, we find that SMT only gives us a 10–25% benefit; SMT generally benefits the algorithms bound by memory latency and instruction dependencies. PALM efficiently utilizes data-level parallelism to achieve 2.8X on the 4-wide SIMD found on our test platform.

5.2 Comparison with Intel® MIC

We have implemented PALM for the Intel® MIC architecture[3], and analyzed the performance in a variety of scenarios.

Fig. 5 shows that our Intel® Knights Ferry implementation is 1.5X faster than our dual-socket CPU implementation for modest tree sizes (20M keys) for frequently updated trees and 2.1X faster for nearly pure search queries. This 20M key tree occupies over 360MB of memory; Intel® Knights Ferry's wide SIMD and higher bandwidth help it outperform the dual-socket CPU.

[3]The Intel® MIC architecture is an Aubrey Isle-based [31] silicon platform and Intel® Knights Ferry [32] is its first implementation with 32 cores running at 1.2GHz. It is an x86-based many-core processor architecture; each core has a scalar unit as well as a vector unit that supports 16 32-bit operations per clock. Intel® Knights Ferry has an 8MB globally-coherent L2 cache that is partitioned among the cores. Our implementation of PALM on Intel® Knights Ferry differs from the CPU implementation only in the addition of code to use the 16-wide SIMD capabilities and an increase of the input batch size to give a larger portion of queries to each of the 128 threads used.

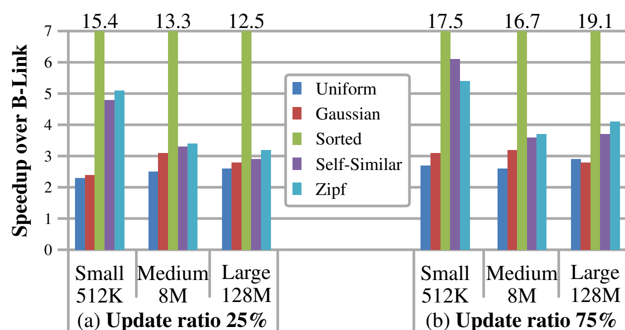


Figure 6: Relative performance of PALM as compared to B-link Trees for small, medium and large trees of 4-byte keys on different distributions and update ratios of (a) 25% and (b) 75%. PALM is 2.3X – 19.1X faster than B-link Trees.

The update/search ratios chosen for these experiments are based on those found in the Transaction Processing Performance Council (TPC)'s standard benchmarks; namely TPC-E, TPC-C, and TPC-H, which were determined by Chen et al. [7] and Kavalanekar et al. [19] to have update ratios of 35%, 10%, and 0.01%, respectively.

5.3 Latency and PALM vs other algorithms

Previous work [36] indicates that buffering methods do not guarantee fast response times, and therefore they force a flush every 0.5–1 seconds. In contrast, our scheme allows for very low response times of less than 350μs even without explicit interrupting the update process. Please see App. D.3 for a detailed analysis of throughput and latency in PALM.

We compare PALM to the best known technique for concurrent inserts — the B-link tree [23]. This uses latches and versioning [6] to support *asynchronous* search queries while inserts require latches on leaf nodes. PALM is 2.3X – 19.1X faster than B-link trees for a variety of tree sizes and input distributions; see Fig. 6, and see App.D.4 a detailed description of our comparison.

6. DISCUSSION AND CONCLUSION

This work describes a scheme for parallel queries on modifiable trees on many-core processors. Future hardware trends are towards increasing numbers of cores and increasing bandwidth, and our algorithm efficiently uses both these resources. Contention for locks will become an increasing source of overhead for many technique as cores increase in number; this can result in poor scaling — even the cost of obtaining locks without contention will increase due to growing costs of cross-core/cross-socket communication. Additionally, the cost of barrier synchronization increases with the number of cores [29]. It will become critical to use lock-free algorithms without all-to-all barriers. PALM uses point-to-point communication rather than all-to-all barriers and effectively utilizes widening SIMD resources, making it a good choice for current and upcoming many-core hardware.

PALM is simple to debug and maintain. The use of a Bulk Synchronous Parallel model avoids the problems of race conditions and deadlocks that traditional lock-based algorithms face. It is also possible to avoid these issues using transactional memory [22]. However, transactional memory does not solve the performance problems associated with contention, and may require multiple rollbacks if contention is high. In contrast, our algorithm can complete without any rollbacks and therefore guarantees low response times.

As part of future work, we intend to extend our implementation to handle variable-length keys. We will also extend our techniques to larger data sets that reside on disk, especially given recent trends towards high-bandwidth SSDs.

7. REFERENCES

[1] L. Arge, K. Hinrichs, J. Vahrenhold, and J. S. Vitter. Efficient bulk operations on dynamic R-trees. *Algorithmica*, 33(1):104–128, 2002.

[2] B. Bayer and E. M. McCreight. Organization and maintenance of large ordered indices. *Acta Inform.*, 1:173–189, 1972.

[3] R. Bayer and M. Schkolnick. Concurrency of operations on B-trees. *Acta Inform.*, 9:1–21, 1977.

[4] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. 1987.

[5] L. Breslau, P. Cue, P. Cao, L. Fan, et al. Web caching and zipf-like distributions: Evidence and implications. In *INFOCOM*, pages 126–134, 1999.

[6] S. Cha, S. Hwang, K. Kim, and K. Kwon. Cache-conscious concurrency control of main-memory indexes on shared-memory multiprocessor systems. In *VLDB*, pages 181–190, 2001.

[7] S. Chen, A. Ailamaki, M. Athanassoulis, P. Gibbons, R. Johnson, I. Pandis, and R. Stoica. Tpc-e vs. tpc-c: characterizing the new tpc-e benchmark via an i/o comparison study. *ACM SIGMOD Record*, 39(3):5–10, 2011.

[8] J. Chhugani, A. Nguyen, V. Lee, et al. Efficient implementation of sorting on multi-core SIMD CPU architecture. *VLDB*, 1(2):1313–1324, 2008.

[9] J. Cieslewicz and K. A. Ross. Adaptive aggregation on chip multiprocessors. In *VLDB*, pages 339–350, 2007.

[10] G. Graefe. B-tree indexes for high update rates. *SIGMOD*, 35(1):39–44, 2006.

[11] G. Graefe. A survey of B-tree locking techniques. *ACM Trans. Database Syst.*, 35:16:1–16:26, July 2010.

[12] J. Gray and A. Reuter. Transaction processing: Concepts and techniques. 1993.

[13] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly generating billion-record databases. In *SIGMOD*, pages 243–252, 1994.

[14] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15:287–317, December 1983.

[15] N. Hardavellas, I. Pandis, R. Johnson, N. Mancheril, A. Ailamaki, and B. Falsafi. Database servers on chip multiprocessors: Limitations and opportunities. In *CIDR*, pages 79–87, 2007.

[16] S. Harizopoulos and A. Ailamaki. A case for staged db systems. In *CIDR*, 2003.

[17] Q. Hou, K. Zhou, and B. Guo. BSGP: bulk-synchronous GPU programming. *ACM Trans. Graph.*, 27(3), 2008.

[18] T. Johnson and D. Sasha. The performance of current B-tree algorithms. *ACM Trans. Database Syst.*, 18:51–101, March 1993.

[19] S. Kavalanekar, D. Narayanan, S. Sankar, E. Thereska, K. Vaid, and B. Worthington. Measuring database performance in online services: a trace-based approach. *Performance Evaluation and Benchmarking*, pages 132–145, 2009.

[20] C. Kim, J. Chhugani, N. Satish, et al. FAST: fast architecture sensitive tree search on modern CPUs and GPUs. In *SIGMOD*, pages 339–350, 2010.

[21] S. Kumar, D. Kim, M. Smelyanskiy, Y.-K. Chen, C. J. Hughes, C. Kim, et al. Atomic vector operations on cmps. In *ISCA*, pages 441–452, 2008.

[22] J. R. Larus and R. Rajwar. *Transactional Memory*. Synthesis Lectures on Computer Architecture. Morgan and Claypool Publishers, 2006.

[23] P. Lehman and S. Yao. Efficient locking for concurrent operations on B-trees. *ACM Transactions on Database Systems (TODS)*, 6(4):650–670, 1981.

[24] D. Lomet. Simple, robust and highly concurrent B-trees with node deletion. In *Data Engineering*, pages 18–27, 2004.

[25] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146, 2010.

[26] H. Plattner. A common database approach for OLTP and OLAP using an in-memory column database. In *SIGMOD*, pages 1–2, 2009.

[27] J. Rao and K. Ross. Making B+-trees cache conscious in main memory. *ACM SIGMOD Record*, 29(2):475–486, 2000.

[28] J. Rao and K. A. Ross. Cache conscious indexing for decision support in main memory. In *VLDB*, pages 78–89, 1999.

[29] J. Sampson, R. Gonzalez, J.-F. Collard, N. P. Jouppi, et al. Exploiting fine-grained data parallelism with cmps and fast barriers. In *MICRO*, pages 235–246, 2006.

[30] N. Satish, C. Kim, J. Chhugani, et al. Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort. In *SIGMOD*, pages 351–362, 2010.

[31] L. Seiler et al. Larrabee: A Many-Core x86 Architecture for Visual Computing. *SIGGRAPH*, 27(3), 2008.

[32] K. B. Skaugen. Keynote at International Supercomputing Conference, 2010.

[33] V. Srinivasan and M. Carey. Performance of B+ tree concurrency control algorithms. *VLDB*, 2(4):406, 1993.

[34] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33:103–111, 1990.

[35] L. T. Yang and H.-X. Lin. Parallel performance analysis of the improved quasi-minimal residual method on bulk synchronous parallel architectures. *The Journal of Supercomputing*, 13(2):191–210, 1999.

[36] J. Zhou and K. Ross. Buffering accesses to memory-resident index structures. In *VLDB*, pages 405–416, 2003.

APPENDIX

A. POINT-TO-POINT SYNCHRONIZATION

Global all-to-all barriers are conceptually convenient, but can carry performance penalties by stiffly penalizing thread divergence and by incurring communication costs; these costs grow increasingly severe when dealing with a large number of threads. It is possible to relax the need for such barriers in PALM trees through a synchronization protocol. In Sec. 3.2.5, we show that a sorted O makes it simple to determine how to re-divide work between threads; this is because it precisely limits each threads' *domain of dependence* between queries. In order to proceed from stage 1 to stage 2, from stage 2 to stage 3, or to climb the tree successively in stage 3, each thread needs to have all its work defined and in accordance with Eq. (1).

Each threads' communication is limited to its 'neighbor' threads $i-1$ and $i+1$ in PALM's point-to-point synchronization scheme. For REDISTRIBUTE-WORK to be correct, a thread i must know if it must discard its first node λ_0^i and if it must collect work from a higher-numbered thread. In most cases, threads need only wait for their immediate neighbors to be finished with a stage to find this information; however, it is possible that immediate neighbors are unable to immediately provide it. Threads that have no items in L_i have no useful information to convey to their neighbors; our scheme uses these as *relays* to convey information about work being done by adjacent threads. Alg. 3 shows our scheme for synchronization with minimal communication. Note that the boundary threads 0 and $t-1$ must be handled specially — they have no *sent-first* and *sent-last*, respectively.

B. NODE REPRESENTATION

Internal nodes in B$^+$ trees with key byte length B have space for MAX-DEGREE 4-byte child references — each represents the cache line offset of the child (this allows us to address 256GB of memory for nodes). They also store (MAX-DEGREE $-$ 1) B-byte keys that indicate the ranges of the keys in each child's descendants, and a *nchildren* value with enough bits to represent numbers in [0, MAX-DEGREE] that indicates how many children are referenced by the node at any time.

Leaf nodes store MAX-DEGREE B-byte keys and 8-byte pointers representing each r^*, and a *nkeys* value long enough to represent the available sizes.

B.1 Parent pointers

Tree queries begin with a downward traversal of the tree using child references, but it is also necessary to walk *up* the tree — e.g., when a node splits and its parent needs to be updated. Because these upward walks always ensue downward ones, B$^+$-tree techniques simply keep track of nodes visited during the descent to use if split or underflow occurs. This is conceptually sound for PALM trees, but then a path must be recorded for each query in a batch. It is more practical to have each node store a reference to its parent that is properly maintained during modification queries. The necessary 4 bytes for this reference are typically available in the padding added to nodes to ensure they fill cache lines.

B.2 Unsorted leaves

While the (child, range) information kept in internal nodes is always kept in sorted order, PALM trees do not keys in leaf nodes are not. The practical benefit of keeping internal nodes in order is to allow for quick child lookups, and the expense of maintaining this order is mitigated by the low frequency of splits at higher levels in the tree.

Leaf nodes are much more frequently updated — the cost of moving keys and pointers to maintain this order would be more frequently paid. We simply append inserted (key, rid) pairs to the items in each leaf, and replace deleted ones with pairs from the end the list. When a node must be split, we sort the (at most) MAX-DEGREE keys before merging — this a small part of the cost of the overall work done in a split.

C. PERFORMANCE MODELING

In this section, we build a simple yet representative analytical model that characterizes the performance of PALM. This model also helps in comparing the results projected by the analytical model to those achieved by our implementation. We focus on the **insertion cost** in this section, while the retrieval and deletion costs can be modeled in a similar fashion.

We compute the number of executed instructions ($\mathcal{C}^I{}_{batch}$), and also the bandwidth required ($\mathcal{C}^B{}_{batch}$, in bytes) for each step of the insertion. We provide a rough sketch of the costs of various components of the insertion process due to lack of space. We introduce the following notations:

K : Number of queries in the batch.
$\mathcal{E}$: Key size (in bytes).
$\mathcal{C}$: Last Level Cache Size (in bytes).
$\mathcal{L}$: Cache line size (in bytes) (equals 64B for CPUs).
$\mathcal{T}$: Minimum Degree of a Node. Max. Degree = ($2\mathcal{T}$-1)
$\mathcal{N}$: Number of keys (all unique) in the B$^+$-tree.
$\mathcal{K}$: SIMD width (in bytes).
$\mathcal{P}$: Number of executing threads.

Algorithm 3 Our algorithm for synchronizing threads using point-to-point communication

SYNC (i, d)

// i is thread id, d the tree depth we wish to proceed to
1 $\quad$ *sent-first*, *sent-last* = FALSE
2 $\quad$ *their-first*, *their-last* = $\emptyset$
3 $\quad$ **if** $|M_i^d| > 1$
4 $\quad\quad$ *my-first* = λ_0^d
5 $\quad\quad$ *my-last* = $\lambda_{|M_i^d|-1}^d$
6 $\quad$ **elseif** $|M_i^d|$ == 0
7 $\quad\quad$ *my-first* = $\emptyset$
8 $\quad\quad$ *my-last* = $\lambda_{|M_i^d|-1}^d$
9 $\quad$ **else**
10 $\quad\quad$ *my-first*, *my-last* = $\emptyset$
11 $\quad$ **while** $\neg \bigwedge$ {*their-first*, *their-last*, *sent-first*, *sent-last*}
12 $\quad\quad$ **if** *my-first* $\wedge\ \neg$*sent-first*
13 $\quad\quad\quad$ SEND-FIRST $(i-1, d,$ *my-first*$)$
14 $\quad\quad\quad$ *sent-first* = TRUE
15 $\quad\quad$ **if** *my-last* $\wedge\ \neg$*sent-last*
16 $\quad\quad\quad$ SEND-LAST $(i+1, d,$ *my-last*$)$
17 $\quad\quad\quad$ *sent-last* = TRUE
18 $\quad\quad$ **if** $\neg$*their-first*
19 $\quad\quad\quad$ *their-first* = TRY-RECV-FIRST $(i+1, d)$
20 $\quad\quad$ **if** *their-first* $\wedge\ \neg$*my-first*
21 $\quad\quad\quad$ *my-first* = *their-first*
22 $\quad\quad$ **if** $\neg$*their-last*
23 $\quad\quad\quad$ *their-last* = TRY-RECV-LAST $(i-1, d)$
24 $\quad\quad$ **if** *their-last* $\wedge\ \neg$*my-last*
25 $\quad\quad\quad$ *my-last* = *their-last*

ρ : Depth of the Tree.
β : Average Occupancy of the Nodes in the B$^+$-tree.
κ : Number of times the *split* routine is called for a given batch. κ is dependent on both the buffer size (K), and the input distribution.
μ : Average number of keys in a node, equal to $2\beta\mathcal{T}$.
$\mathcal{S}_{NL}$: Non-Leaf Node Size (in Bytes) = $\mathcal{L}\lceil((4+\mathcal{E})(2\mathcal{T}\text{-}1)+4)/\mathcal{L})\rceil$.
$\mathcal{S}_L$: Leaf Node Size (in Bytes) = $\mathcal{L}\lceil((8+\mathcal{E})(2\mathcal{T}\text{-}1)+4)/\mathcal{L})\rceil$.

C.1 Single-thread performance

The time spent for insertion of *a batch* into the tree can be divided into the following *three* categories:

1. **Sorting the batch of K queries:** $\mathcal{C}^I{}_{sort} = \mathcal{C}_0 K\log(K)$, where $\mathcal{C}_0$ is the number of executed instructions per element for every iteration of the mergesort algorithm (log$K$ iterations in total). We use a SIMD friendly mergesort implementation [8]. Since we sort tuples consisting of keys and index, the total size of each tuple is ($\mathcal{E}$+4) bytes. $\mathcal{C}^B{}_{sort} = (\mathcal{E}+4)K$.
2. **Searching a key from the root to the leaf:** The depth of the tree (ρ) = $\lceil \log_\mu \mathcal{N} \rceil$. Average number of keys in a node (μ) = $2\beta\mathcal{T}$. We use a SIMD-based comparison algorithm, and compute the number of set bits in the comparison mask. Since the algorithm works on a SIMD group of $\mathcal{K}/\mathcal{E}$ elements simultaneously, Hence, number of such groups in a node = $\lceil ((\mu\mathcal{E})/\mathcal{K}) \rceil$. Therefore, $\mathcal{C}^I{}_{search} = \rho\mathcal{C}_1\lceil\mu\mathcal{E}/\mathcal{K}\rceil$, where $\mathcal{C}_1$ equals the cost of performing the SIMD search on a SIMD-wide register of keys.

   As far as the total bandwidth required is concerned, it is possible that only a few levels of the tree can fit in cache (true for medium and large size trees). Say ρ' levels of the tree fit in cache. Hence ρ-ρ') levels need to be fetched from the main memory, which would include one leaf node, and the remaining ones would be non-leaf-nodes.

   Hence total bandwidth required = Max(0,(ρ-ρ'-1)$\mathcal{S}_{NL}$) + $\mathcal{S}_L$.
3. **Cost of Actual Insertion:** Having identified the node of the tree where the list of (key,rid) pairs need to be inserted, the actual process can lead to the following *two* scenarios: (1) The pairs can simply be inserted into the leaf node without causing any splits. (2) Insertion would cause a split, and hence the keys in the leaf need to be rearranged to obtain a valid B$^+$ tree. In case of no splitting, the cost of insertion of an element is simply an insertion cost (copying of key, rid) and incrementing some counter. Since this also includes some book keeping cost (copying keys from various buffers), we denote the total sum of this as $\mathcal{C}^I{}_{insert_one}$.

   In the case of splits, we need to perform sorting of the (key,rid) pairs already present in the leaf node, followed by the merging of the query (key,rid) pairs. We use a SIMDfied mergesort implementation, followed by the SIMD friendly merge operation. The total cost of split ($\mathcal{C}^I{}_{split}$) equals $\mathcal{C}_0\mu\log(\mu)$. The cost of merging is already accounted for in $\mathcal{C}^I{}_{insert_one}$. Note that our batched insert scheme ensures that we split a node into multiple nodes by sorting the already existing keys in the node **only once**, as opposed to using single inserts, that would cause as many splits as the number of new leaf nodes created.

   Note that when a non-leaf-node is split, we do not need to sort the exising keys, and hence only a merge is required. In addition, since non-leaf-node splits occur μ times less frequently, the impact of non-leaf-nodes has a *negligible impact* on the run-time (for $\mathcal{T} \geq 8$).

As far as memory bandwidth is concerned, $\mathcal{C}^B{}_{split} = \mathcal{S}_L$. Note that $\mathcal{S}_L$ will be replaced by $\mathcal{S}_{NL}$ in case we are dealing with splitting of non-leaf nodes. Also, note that new nodes are created during this process, but one can allocate this memory in batches during some compute-bound phase of the algorithm, and potentially hide the bandwidth associated with it.

Another important factor that dictates the total run-time is the number of splits (κ). For uniformly random data, we expect the input queries to be uniformly spread across the leaf nodes, and as such, the number of simultaneous splits for any node may be one (asymptotically $\kappa = (K/\mu)$. In case of skewed data, we expect the number of nodes being split to reduce substantially. Consider a completely sorted sequence of input queries, such the minimum key of the batch being inserted is greater than the maximum key in the tree. Now all the queries will will go to the right most leaf node of the tree. Although $K/\mathcal{T}$ nodes will be created, the split routine will only be called once, and hence κ = 1.

Total cost for a batch. As far as the number of instructions is concerned, $\mathcal{C}^I{}_{batch}$ [4] $= \mathcal{C}^I{}_{sort} + K\mathcal{C}^I{}_{search} + K\mathcal{C}^I{}_{insert_one} + \kappa\mathcal{C}^I{}_{split}$.

The bandwidth cost can be formulated in a similar way from the description above. Now lets consider two examples to compute the cost, and compare it with the obtained run-times. We will then compare the run-times of the parallel implementation with the values predicted by the model.

Consider a tree with 512K unique keys, with keysize ($\mathcal{E}$) = 4B. Assume the keys are arriving in a uniformly random fashion, and hence β = 0.7 [11]. We chose $\mathcal{T}$ = 16 for our implementation, since it provides the right balance between the tree depth and the frequency of splits (for various input distributions). Hence, μ = 2(0.7)(16) = 22.4, and ρ = 5. As far as costs $\mathcal{C}_0$ and $\mathcal{C}_1$ are concerned, $\mathcal{C}_0$ = 5 [8], and $\mathcal{C}_1$ = 6 (computed from our code by inspecting the assembly instructions generated). Furthere, $\mathcal{C}^I{}_{insert_one}$ = 20 ops. Say the batch size (K) equals 8192. Furthermore, κ = 8192/22.4 = 365. The cache size ($\mathcal{C}$ = 24MB), and hence the tree completely fits in the last level cache. Hence we focus on the executed instructions to predict the run-time.

Putting it all together,
$\mathcal{C}^I{}_{sort}$ = (5)(8192log(8192)) = (8192)(65).
$\mathcal{C}^I{}_{search}$ = 8192(6)(16/4)$\lceil$((22.4)(4)/16)$\rceil$ = 8192(180).
$\mathcal{C}^I{}_{insert_one}$ = 8192(20).
$\mathcal{C}^I{}_{split}$ = (365)(5)(24)(5) = 365(600) = 8192(26.7).

Hence, total cost per key = 65 + 180 + 46.7 = **291.7 cycles** (assuming IPC=1). Our implementation reports **294.2 cycles** on 1-core, which is **within 1% of the predicted performance**.

For a larger input set, say with 128M keys in the tree, ρ = 7, and only 5 levels (at an average) can be cache-resident (cache-size on our machine is 24MB, while the dataset size is around 2GB). Hence the search time for the first 5 levels is around 5(36) cycles, while the remaining two levels need to be fetched from the main memory. One non-leaf node, with $\mathcal{S}_{NL}$ = 256B, and one leaf node, with $\mathcal{S}_L$ = 384B. In general, since we are accessing multiple cache lines together, the hardware pre-fetcher starts preemptively fetching extra cache lines. Although this number of extra cache-lines varies between different hardwares, our separate benchmarks reveal that around 2 extra cache-lines are also brought in (once we access more than 2 cache-lines). Note that since the last two levels are fetched

[4] There also exist other costs like loop overhead, register spills/fills, etc. We choose the degree of the tree in a way that such overheads are reduced to an insignificant portion of the run-time. Choosing a small value of $\mathcal{T}$ exposes such overheads, and reduces the efficiency of the code.

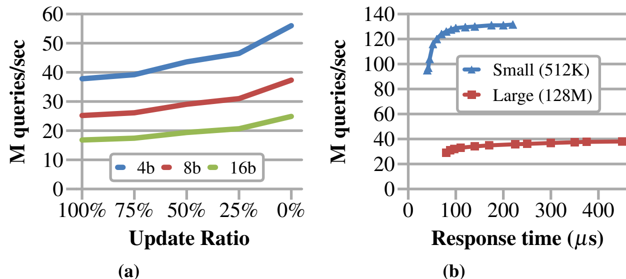


Figure 7: (a) Performance of queries on large trees with varying update ratios for 4-, 8- and 16-byte key sizes, drawn from uniform distributions. (b) Throughput vs response times for small trees and large trees achieved by varying query batch sizes.

from main memory, the time to sort (for splits) can be interleaved with cache line transfer, and the total time spent on the last two levels of the tree would simply be the sum of times spent on fetching these two levels. Since one core can utilize up to 50% of the peak bandwidth of the socket, we expect to get around 3.5 bytes/cycle. Hence total search time = 180 + (256+128+384+128)/3.5 = 436 cycles. Adding the time to sort the queries, the total time adds up to around **501 cycles** per input query. Our implementation reports around **526.5 cycles**, which is **within 5% of the predicted performance**.

Now consider a distribution where input keys are all unique and sorted. For sorted input distribution, β = 0.5 [11], and hence μ = 16. For input tree with 512K keys , the depth of the tree is still 5. Hence the search time becomes 24*5 = 120 cycles per key. As far as splitting cost is concerned, since κ in this case would be 1 for the batch, the sorting cost per key is negligible, around 0.05 cycles. Hence the cost of performing inserts is actually much faster than the cost of insertion on trees with random distributions. The overall run-time per key sums up to 65 + 120 + 20 = 205 cycles, while the implementation reports around 204.2 cycles, which is **within 1% of the predicted performance**.

As the tree size gets larger, since the input insert queries are all greater than the maximum key in the tree, they all traverse down the right-most node at each level, and hence the resultant implementation is expected to be compute bound. For a tree with 128M keys, the total run-time would be 65 + (7)(24) + 20 = 253 cycles. The implementation reports around 260.8 cycles, which is within 3% of the predicted performance.

C.2 Multi-thread performance

Our parallel algorithm performs all the above tasks in parallel, with appropriate sync points. For trees that completely fit in cache, $\mathcal{C}^I{}_{search}$ would scale linearly for part of the tree that fits in cache, while the remaining levels would scale up to the bandwidth limitations of the system. The cost of the barrier (denoted as $\mathcal{C}_{barrier}$) on our system was around 5000 cycles (for 12-cores).

Consider the uniformly random distribution with 512K keys. Assuming linear scaling (and negligible cost of communication), the parallel time should be $\mathcal{C}^I{}_{sort}/\mathcal{P}$ + $\mathcal{C}^I{}_{search}/\mathcal{P}$ + $\mathcal{C}^I{}_{insert_one}/\mathcal{P}$ + $\mathcal{C}^I{}_{split}/\mathcal{P}$ + $\mathcal{C}_{barrier}$ = 65/12 + 180/12 + 46.7/12 + 5000/8192 = 24.9 cycles. In practice, we achieve around 25.7 cycles, which is within 4% of the predicted values.

We can similarly project the performance for large trees and other distributions.

Summary. Our analytical formulation models the compute and bandwidth performance limits. Since our actual results match these limits, we conclude that our implementation is either compute or bandwidth limited, and is able to efficiently utilize the underlying architectural resources.

D. RESULTS

D.1 Input distributions

We used a variety of input key distributions to evaluate PALM:

Uniform The queries are chosen from a uniform random distribution over the entire range of 4-byte, 8-byte or 16-byte values.

Gaussian The queries are chosen to follow a Gaussian distribution with mean of $N/2$, where N is the number of tuples in the tree and a standard deviation of 0.5% of the mean.

Sorted The set of queries is sorted in increasing order with respect to the keys. This can occur when keys are sorted prior to tree building. Since each successive query is larger than all preceding queries, all search and update queries will only access and update the rightmost path and leaf node in the tree.

Self-similar The queries are chosen using a 80–20 rule as described in [13]: 80% of the input tuples cover only 20% of the overall cardinality of the keys.

Zipf The keys of the queries follow the Zipf distribution [13], which generates a skewed distribution of keys. The Zipf distribution is parameterized by the value of θ, and we use θ = 1. We use this as a proxy of the actual query distribution in real world data [5]. Queries following a Zipf distribution will access some paths of the tree more often than others.

D.2 Impact of varying key sizes/update ratios

Fig. 7(a) shows the performance of a combination of search and update queries with varying update ratios for tuples with 4-byte, 8-byte and 16-byte keys chosen from a uniformly random distribution. All queries were run on a large tree with 128M existing tuples. The performance on 4-byte keys varies from 38M queries per second for 100% updates to 56M queries per second for 0% updates. For larger keys, the performance drops — we need to execute more instructions and consume more memory bandwidth to sort, search and update nodes. For 16-byte keys, the performance for 100% updates is about 17M queries/second and 25M queries/second for 0% updates.

D.3 Latency vs throughput

Fig. 7(b) shows the trade-off between response time (latency) to a single query and throughput of query processing by varying the batch size. The response time of a query is defined as the time that taken to process all queries in the input batch.

Fig. 7(b) shows that for **small trees**, we obtain **80% of peak throughput** even at a **very low response time of 60 microseconds**. We also get to **99% of the peak throughput with a response time of 150 microseconds**. For large trees with 128M tuples, we get to 80% and 99% of peak throughput at **100 and 350 microseconds** respectively. These response times are low enough to allow our scheme to be used even in real-time databases. The main reason for our low response times is that even for small buffer sizes of a few thousand elements, we obtain close to peak throughput. We advocate the use of buffering for updating index trees since it can result in improved throughput as well as responsive systems.

D.4 Comparison to B-link

In order to perform a fair comparison, we have also incorporated our SIMD and latency hiding scheme into B-link, so that the

Tree Size	Sorting	Searching	Modify
Small	20%	49%	31%
Medium	9%	67%	24%
Large	6%	69%	25%

Table 1: Performance Breakdown in terms of the percentage of time spent in each step for a query mix consisting of 100% update queries on small, medium and large size trees. Tuples are drawn from a uniformly random distribution. All breakdowns are for runs with 24 threads.

resultant numbers bring out the scaling of their algorithm[5]. The resultant B-link numbers of our implementation on single-thread are comparable (within 5%) of the performance numbers of PALM on single-thread.

Fig. 6 shows the relative speedup of PALM with varying update rates. The numbers for the various input distributions are shown. In Fig. 6(a), for *uniformly random* distributions, PALM is 2.3X – 2.6X faster than B-link trees for sizes; B-link suffers from the overhead of communication of the latch data structure, which potentially is touched by various cores on the same socket and across sockets, increasing latency and reducing scaling.

As data skewness increases (*Gaussian*, *self-similar* and *Zipf*), contention on latches increases, further reducing scaling of B-link. The relative performance of PALM increases to 2.9X – 3.2X, even on large trees. *Sorted input* reflects the worst case, where all the threads compete to update nodes at each level of the tree; only *one* thread can make progress. Fig. 6(b) shows further increase in relative performance for a larger quantity of updates.

To evaluate PALM's single-thread performance, we evaluated it against a reference scalar buffered CSB+ implementation [36]. PALM demonstrated 2.3X higher throughput for small and medium trees and around 2.8X higher throughput on large trees (primarily due to a combination of better SIMD utilization and latency hiding).

D.5 Performance breakdown

Table 1 shows the percentage of time spent in sorting, searching and tree modification for a query mix consisting entirely of update queries. Results are shown for varying tree sizes (number of tuples in the tree) with 24 threads. The table shows that about 50–70% of the runtime in spent in searching for the leaf node in which to insert the tuple. The time spent in the actual modification of tree nodes is only around 30%. Moreover, the percentage of time spent in search increases with tree size, since more levels must be traversed in search. We also note that the time spent in **sorting the queries is only 10–20% of overall time**. We find that sorting the queries results in a net speedup in overall runtime.

[5]Performance numbers on much older hardware are given the original paper [6]; even after normalizing for frequency, our performance on a single-thread is 2X theirs. They reports data only for uniform distributions, while we focus on a wide-range of distributions.