Brief Announcement: Layering Data Structures over Skip Graphs for Increased NUMA Locality

Samuel Thomas
Hammurabi Mendes
sathomas@davidson.edu
hamendes@davidson.edu
Davidson College
Davidson, NC, USA

ABSTRACT

We present a lock-free, linearizable, and NUMA-aware data structure that implements sets, maps, and priority queue abstract data types (ADTs), based on using thread-local, sequential maps that are used to "jump" to suitable positions in a lock-free, linearizable variant of a skip graph. Our skip graph is suitably constrained in height and subjected to a data partition scheme that reduces contention and increases NUMA locality. We developed an additional skip graph variant, which we call sparse skip graph, that causes our thread-local maps as well as our shared structure to become more sparse. Compared to using regular skip graphs, sparse skip graphs show increased performance in workloads dominated by "insert" or "remove" operations, and comparable performance in workloads dominated by "contains" operations.

CCS CONCEPTS

• **Computing methodologies → Concurrent algorithms**; • **Software and its engineering → Concurrency control**.

KEYWORDS

synchronization, skip graphs, NUMA, lock-freedom, locality

ACM Reference Format:
Samuel Thomas and Hammurabi Mendes. 2019. Brief Announcement: Layering Data Structures over Skip Graphs for Increased NUMA Locality . In *2019 ACM Symposium on Principles of Distributed Computing (PODC '19), July 29-August 2, 2019, Toronto, ON, Canada.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3293611.3331576

1 OUR APPROACH

We implemented a lock-free, linearizable, and NUMA-aware data structure that implements a set, map, and priority queue abstract data types (ADTs), based on using thread-local, sequential maps *layered* on top of a lock-free, linearizable variant of a *skip graph* [1, 3]. Our *layered structure* is composed of multiple *local structures*, all thread-local, sequential, navigable maps, and a single *shared structure*, a skip graph, operated concurrently by all threads. A skip graph is comprised of multiple skip lists that increasingly share lower levels (see more details below).

We distinguish between *local nodes* or *shared nodes* depending on which structure they belong to. Each local structure's job is to map elements inserted by their owning thread to shared nodes in the skip graph. Inserting an element e adds a shared node s to the skip graph, and creates a mapping $e \longrightarrow \mathrm{s}$ in the local structure. A removal of element e will (i) *logically delete* the shared node s in the skip graph, (ii) cause a physical cleanup in the shared structure and (iii) cause the thread that contains the mapping $e \longrightarrow \mathrm{s}$ in its local structure to physically cleanup that association upon detection. Steps (ii) and (iii) can happen in any order.

The skip graph and its partitioning. Our skip graph contains multiple singly-linked lists at different levels (Fig. 1). Starting from level zero, each level i contains 2^i lists. All elements belong to the level-0 list, labeled as "λ", the empty string. The level-0 list is partitioned into two level-1 lists, labeled "0" and "1". Each level-1 list is further partitioned into two level-2 lists: the level-1 list labeled "0" (resp. "1") is partitioned in two level-2 lists, labeled "00" and "01" (resp. "10" and "11"). The partitioning of elements is *not* done in a probabilistic way as in the original skip graph: we have a *partitioning scheme* that assigns threads to levels.

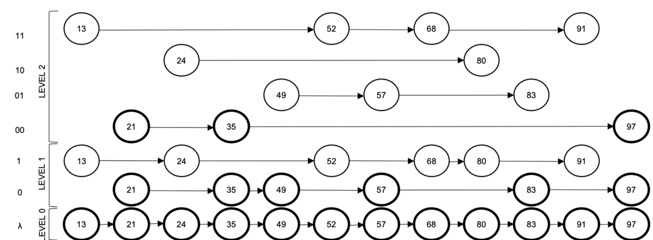

Figure 1: A skip graph is a collection of skip lists sharing their bottom levels. We partition our dataset suitably so that we increase NUMA locality and reduce contention.

We consider a system of T threads $\{t_1 \ldots t_T\}$. The skip graph's maximum level is $\mathrm{MaxLevel} = \lceil \log(T) \rceil - 1$. A low MaxLevel like this does not guarantee logarithmic searches on a skip graph standing by itself, but our local structures compensate this fact by "jumping" to positions in the skip graph near where data structure operations take place. This "jump" happens *without any contention*, as each thread's local structure is private. Each thread T_i will have a *membership vector* M_i, a sequence of MaxLevel bits,

Permission to make digital or hard copies of part or all 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. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
PODC '19, July 29-August 2, 2019, Toronto, ON, Canada
© 2019 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-6217-7/19/07.
https://doi.org/10.1145/3293611.3331576

whose suffixes indicate on which *unique* shared skip lists the particular thread can operate. This essentially makes all insertions from a single thread to happen in a unique skip list within the skip graph, denoted by L_i, and called the *associated skip list* of thread T_i. As an example, consider the skip graph with $\mathrm{MaxLevel} = 2$ of Fig. 1. Each thread will have a 2-bit membership vector. If, say, T_i has M_i = "10", then T_i will always insert or remove in the skip list $L_i = (\lambda, 0, 10)$. With this scheme, we have at most two threads operating in each of the top-level shared linked lists, and at most $T/2^i$ threads operating in any particular level-i list. Note that the original definition of skip graphs assumes a membership vector *per element*, while in our approach all elements inserted by a single thread share the *thread*'s membership vector. Membership vectors can be generated as simply as taking the binary suffix of appropriate length of each thread ID, or *according to physical NUMA characteristics* of the machine.

We define *local* memory accesses as those operating in memory initially allocated by the current thread (under first-touch NUMA policy), and *remote* accesses simply as those that are non-local. Note that we adopt a conservative notion of local, as data initially allocated by two different threads could be located within the same NUMA node. Nevertheless, our approach, implemented in ∼10K lines of C++14, operate up to 23% faster than a highly-optimized lock-free skip list, an optimized version of that in [6] (our fastest contender), with up to 70% of reduction on the number of remote synchronized reads, and up to a 3.21 times increase in CAS locality – that is, the ratio of local over remote CAS instructions – for 32 threads. In addition, we implemented another skip graph variant, called *sparse skip graph*. Using sparse skip graphs as our shared structure causes our local structures to be made more sparse, as we discuss later in page 2. Compared to using regular skip graphs, sparse skip graphs show increased performance in workloads dominated by "insert" or "remove" operations, and comparable performance in workloads dominated by "contains" operations. We discuss the details in our full paper (available in [7]). Essentially, the overhead of the sparse index (i.e. the increased number of node traversals on searches) is compensated by more efficient insertions and removals if the underlying skip graph is also more sparse. Our technique assumes a homogeneous workload among threads, yet we can imagine adaptations of our mechanism, such as searching using another thread's local structure, or using some form of delegation/helping mechanism to handle heterogeneous workloads.

A relevant previous work that also relies on using "indexes" to improve access locality on "shared structures" is NUMASK [4], a modified skip list that uses its higher levels as a hierarchical index to the bottom-level list. We differ from [4] as our shared structure consists of a multi-level skip graph, aiming (i) to avoid that traversals navigate back and forth across NUMA nodes, increasing locality; and (ii) to reduce contention with a careful partitioning scheme that creates areas within the shared structure where only subsets of threads operate. A similar separation of thread-local views and shared parts has been discussed in [2], although their approach is more akin to combining [5], as they merge local views into the global structure from time to time.

Sparse Skip Graphs. As noted before, we developed a second shared data structure, which we call *sparse skip graph*. This data structure is a skip graph where elements are present in level *i of any shared skip list* with expectation $1/2^i$, just like in a regular skip list. The sparse skip graph is still a set of skip lists sharing their bottom levels, although the levels become more and more sparse just like in a skip list. For instance, in Fig. 1, each of the level-1 lists "0" and "1" would partition only 50% of the elements of "λ", which would be selected at 50% chance independently. So, "0" and "1" would each have about 25% of the elements in "λ". Similarly, the level-2 lists "00" and "01" would partition only 50% of the elements of "0", each selected at 50% chance, independently. So, lists "00" and "01" would each have about 6.25% of the elements in "λ". Note that elements are present in level *i of a particular linked list* with expectation $1/4^i$. Importantly, in our technique, only elements that reach the top level are added to the local structures. Therefore, sparse skip graphs also cause the local structures to become more sparse. This is crucial because the local structures, besides pointing to shared nodes nearby the target destinations, should also point to maximum-level nodes from which we can start an efficient search. Hence, using sparse skip graphs gives two immediate advantages: (i) the local structures are smaller; and (ii) the insertion and removal in the shared structure requires changes in less than $\mathrm{MaxLevel}$ levels. The tradeoff is that the starting point given by the local structures is not as close to the requested element compared to regular skip graphs. Our preprint [7] has a complete discussion on linearizability, both for skip graphs and for sparse skip graphs.

2 EVALUATION

We conduct experiments in a system with 2 (NUMA) sockets, 16 Intel Xeon E5-2620 cores (32 hardware threads), each running at 2.0-2.5GHz (varies due to TurboBoosting), and 128GB of memory. The tool `numactl --hardware` reports relative intra-node distances of 10 and inter-node distances of 21. The system is running Ubuntu 18.04 LTS with kernel version 4.15.0-43. We use `clang++` 6.0, passing the `-std=c++14 -O3` build flags. We run over 4700 tests, spanning 12.4 hours. We run experiments for 10s of CPU time *per thread*, reporting the total number of operations per millisecond, averaged from 5 runs. We run insert-remove-contains mixes of 60-30-10, 30-60-10, and 20-20-60. The numbers denote the percentage of each operation. Insert and contains operations are done with keys chosen uniformly at random from the element space. Removals are done with keys chosen uniformly at random from either (i) the element space; or (ii) the elements previously inserted by the current thread. Each time, a thread chooses either strategy (i) or (ii) with chance 50%. Threads run batches of 10 operations in order to amortize `clock_gettime()`, the *only* system call in the measured interval. The data structures are always preloaded with 20% of their maximum capacity before any measurements. Threads are pinned to each CPU, and we fill a socket before adding threads to another socket. We use `libnuma` on Linux to have each thread *preallocate* 3GB of private memory, under a first-touch NUMA memory policy. We preallocate memory in order to avoid intra-socket delays for operations in the local structure, and we never free it as we do not want to mistakenly account for the performance of the memory allocator. The actual method by which we report the ratio of local vs. total memory accesses is by *manual code instrumentation*.

1 - Remote read-syncs and CASes. Experiments in [7] showed that the absolute number of CAS and read-syncs per operation is highly reduced in our approach in comparison to the skip lists. CASes are reduced only about 7%, but read-syncs are reduced by half with 32 threads, considering our sparse skip graphs. We attribute this to *the impact of our local structure indexing, with the maps*, that indeed jumps to nearby locations. Our experiments also reveal a higher success rate of CASes in the layered approach, with up to 99.4% of successes vs. 95.7% in the skip lists for the 30-60-10. We attribute this to our *partitioning scheme*, designed to accomplish this goal as well as to increase NUMA locality.

2 - Indexing efficiency. Another experiment (see [7]) shows how a layered structure with a non-sparse skip graph has a 76% reduction in the number of traversed shared nodes compared to a regular, non-NUMA, concurrent skip list, on 32 threads, particularly for non-sparse skip graphs. Sparse skip graphs imply more sparse local structure indexing, as less shared nodes reach the topmost level and get included in the local structures. Our result shows that even though the initial point of the search is not as optimized as in the non-sparse skip graph, the number of traversed nodes per search operation is still about half of what we have in a traditional skip list. We also see a test showing our local structure indexing a simple linked list, meant to isolate the performance benefits of skip graphs. With 32 threads, in a workload dominated by contains (so fewer insertions), the number of traversed nodes per search of this indexed linked list can go over 50% higher than in a skip list, whereas in our structures they are at most 50% of the value in a skip list. See [7] for a full set of diagrams.

3 - NUMA locality. Our experiments in [7] also show that the *ratio* of local over total memory accesses is much improved in our approach. For CASes, with 32 threads, a non-sparse skip graph has 81.3% of its operations local (i.e., traversing elements inserted by the operating thread), while this percentage for a sparse skip graph is 40.7%, and for a skip list only 25.3%. For read-sync operations, 60% of them are local in the non-sparse skip graph, 17.4% in the sparse skip graph, but only 2.2% in a skip list (see [7]).

4 - General performance and design by experimentation. As discussed in [7], we noticed that when we use padding to align individual pointers of the `next` array into distinct cache lines, we not only increase memory usage dramatically, but also *hurt* performance ($\sim$ 30% slowdown). This makes sense as our algorithms typically perform read-syncs or CAS instructions across all levels of a node's successor pointers. We also tried to physically delete immediately after its logical deletion, noticing a $\sim$ 10% slowdown even at relatively low contention. We attribute this to missing the chance of using the "hot" cache lines. We do introduce padding in the `head` of the skip graph, as threads only access specific indexes. Figure 2 shows the performance of a contains-heavy scenario in operations/msec. The structures tested are: `map_{sg/ssg}`, C++ maps over skip graphs (resp. sparse skip graphs); `sl_{sg/ssg}`, the same but with a skip list used as the local structure; a lock-based skip list; a concurrent skip graph and a skip list without our layering technique; Finally, `indexed_list` is the approach of merely layering the local structures on top of a shared linked list. We use this test to demonstrate the benefit of skip graphs and our partitioning scheme. We also show insert-heavy and remove-heavy workloads in [7], and *we perform even better in those workloads.*

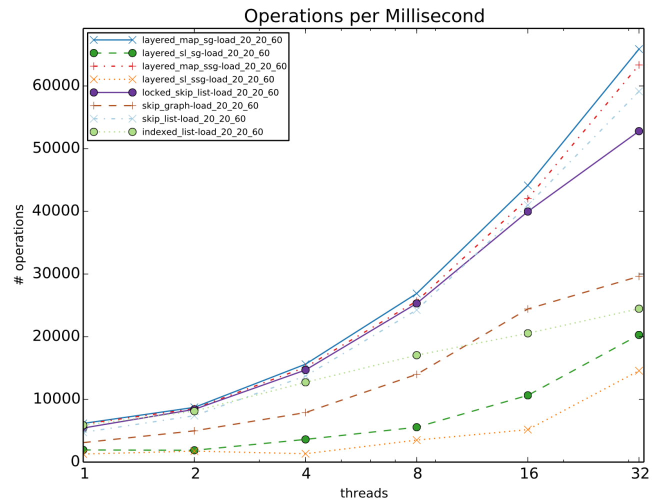

Figure 2: Average ops/msec, 20-20-60

3 CONCLUSION

We presented a technique based on layering local, sequential data structures over skip graphs variants aiming at increasing NUMA locality. At a high level, our design consists of a carefully designed and partitioned shared structure, well-integrated with sequential local structures, operating without contention. Together, these data structures promote increased NUMA locality, as verified by experimentation. Compared to skip lists, our experiments show (i) a much higher ratio of memory reference locality (up to 3.21 times); (ii) better performance for our approach (up to 23%) even though some variants require more synchronization per operation than others; (iii) a slight increase in cache performance (hit ratio), particularly for the sparse skip graphs; (iv) a strong reduction on the number of remote synchronized reads (up to 70%). As future work, we are interested in exploring our structural advantages in the design of exact and relaxed priority queues (see in [7]).

REFERENCES

[1] Nicholas J. A, Dunagan Michael, B. Jones, and Stefan Saroiu. Skipnet: A scalable overlay network with practical locality properties. In *Proceedings of USENIX Symposium on Internet Technologies and Systems*, 2003.

[2] Deepthi Akkoorath, José Brandão, Annette Bieniusa, and Carlos Baquero. Global-local view: Scalable consistency for concurrent data types. In Marco Aldinucci, Luca Padovani, and Massimo Torquati, editors, *Euro-Par 2018: Parallel Processing*, pages 492–504, Cham, 2018. Springer International Publishing.

[3] James Aspnes and Gauri Shah. Skip graphs. *ACM Transactions on Algorithms*, 3(4), November 2007.

[4] Henry Daly, Ahmed Hassan, Michael F. Spear, and Roberto Palmieri. NUMASK: High Performance Scalable Skip List for NUMA. In Ulrich Schmid and Josef Widder, editors, *32nd International Symposium on Distributed Computing (DISC 2018)*, volume 121 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 18:1–18:19, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[5] Panagiota Fatourou and Nikolaos D. Kallimanis. Revisiting the combining synchronization technique. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 257–266, New York, NY, USA, 2012. ACM.

[6] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[7] Samuel Thomas and Hammurabi Mendes. Layering data structures over skip graphs for increased NUMA locality. *CoRR*, abs/1902.06891, 2019.