Global-Local View: Scalable Consistency for Concurrent Data Types

Deepthi Akkoorath[1(✉)], José Brandão[2], Annette Bieniusa[1], and Carlos Baquero[2]

[1] Technical University of Kaiserslautern, Kaiserslautern, Germany
{akkoorath,bieniusa}@cs.uni-kl.de
[2] HASLab, Universidade do Minho and INESC TEC, Braga, Portugal
jose.brandao1994@gmail.com, cbm@di.uminho.pt

Abstract. Concurrent linearizable access to shared objects can be prohibitively expensive in a high contention workload. Many applications apply ad-hoc techniques to eliminate the need for synchronous atomic updates, which may result in non-linearizable implementations. We propose a new model which leverages such patterns for concurrent access to objects in a shared memory system. In this model, each thread maintains different views on the shared object: a thread-local view and a global view. As the thread-local view is not shared, it can be updated without incurring synchronization costs. These local updates become visible to other threads only after the thread-local view is merged with the global view. This enables better performance at the expense of linearizability. We discuss the design of several datatypes and evaluate their performance and scalability compared to linearizable implementations.

1 Introduction

As the number of cores increases in multi-core systems, the synchronization cost becomes more apparent [20]. While linearizability [14] is very useful for reasoning about the correctness of concurrent data structures, its implementation can be prohibitively expensive. As a consequence, programming patterns are emerging in practice, that attempt to limit the associated cost of the required synchronization on the memory accesses by relaxing the concurrent objects semantics. For example, in the widely used messaging library ZeroMQ, adding messages to the queue is performance-critical to the application. While lock-free linearizable queues are fast, the developers observed that the synchronous enqueue of each new messages was affecting the overall performance, especially in high contention workloads [21]. An analysis of the problem domain revealed that only the relative order of messages from a single thread is relevant for the semantics of the message queue; it is not necessary to maintain a strict order of enqueue operations when two independent threads try to insert messages. To overcome the linearizability penalty, the developers re-engineered their message queue such that multiple messages are added in batch, within a single atomic operation.

M. Aldinucci et al. (Eds.): Euro-Par 2018, LNCS 11014, pp. 492–504, 2018.
https://doi.org/10.1007/978-3-319-96983-1_35

For another example, consider a shared counter that is concurrently updated by several threads. The final value must include all increments performed, but the order of increments is not relevant since they are commutative. If each increment by each thread is an atomic operation made visible to all other threads, it can become a bottleneck [8]. In many cases, it is sufficient to execute the increment on a thread-local variable and to apply a combined update to the shared object.

In this paper, we propose a new model for shared objects that leverages the different views of an object, the *global-local view* model. In this model, each thread has a *local view* of the object which is private to it. Threads update and read primarily their local view. The local updates, though visible in a local view, are made visible on a *global view* only after an explicit two-way merge operation is performed. The other threads observe these changes once they synchronize, by merge, their local view with the global view. As the local view is non-shared, the local updates can be executed without requiring synchronization. Threads can execute many local updates without synchronizing with the global view, thus enabling better performance, albeit at the expense of linearizability.

In addition to the local operations, the model also provides synchronous operations on the global view. We call the operations that perform only on local view, *weak operations* and those on global view, *strong operations*. Combining operations on the global and the local views, we can build data types with customizable semantics on the spectrum between sequential and purely mergeable data types. *Mergeable data types* provide only weak and merge operations; *hybrid mergeable data types* offer both weak and strong operations. An application that uses a hybrid mergeable data type may use weak updates when a non-linearizable access is sufficient (e.g. weak enqueue on a local queue) and can switch to use only strong operations when stronger guarantees are required (e.g. strong dequeue to guarantee that item are dequeued only once).

In distributed systems, similar concerns led to the development of conflict-free replicated data types (CRDTs) [19]. CRDTs allow asynchronous updates to local replicas, while guaranteeing strong eventual consistency. In this distributed setting, each replica can be concurrently updated without requiring any synchronization. It can then later be merged with other replicas, while it is guaranteed that all nodes reach a convergent state once all updates are known. CRDTs play an essential role in partition tolerance and scalability [1,2]. However, the applicability of CRDTs as described in literature [19] is limited in a concurrent shared memory environment. For example, a CRDT counter is implemented as a map of replica id to integer. The merge operation iterates over the two maps to be merged and returns a map with the maximum for each entry. Thus, the relative cost in space and time of the merge is linear in the number of entries and as such unfeasibly high. In the global-local view model, the merge is executed synchronously on the global view. If the cost of merge is high, we lose the benefits of allowing parallel updates. While our work is inspired by them, the current CRDT designs are not suitable for relaxing consistency in concurrent shared-memory objects.

Contributions. This paper makes the following contributions:

1. We describe the global-local view model for multi-threaded applications with high contention that implements an adaptable trade-off between update visibility and synchronization cost (Sect. 3).
2. We discuss the implementation of a mergeable counter, a hybrid counter and queue (Sect. 4) and compare their performance with their linearizable counterparts under both low and high contention workloads (Sect. 6).

2 Related Work

Programming Models: Maintaining per-thread replicas and performing updates on them has been considered by different programming models in the literature. In Concurrent Revisions [9], a forked thread applies changes on its copy which is merged (using type-specific merge) to the parent thread when it is joined back. The focus of this work is on a fork-join model, where threads can communicate their state only when they join their parent. In contrast, we provide a generic model for the data types where a two-way merge and strong updates can share states among the threads at any point in the execution.

The Global Sequence Protocol (GSP) [10] is a model for replicated and distributed data systems that allows offline client updates. Since GSP addresses a distributed system model, with no bounds on message delays, there is much less control on replica divergence and liveness of the global sequence evolution. In contrast, we address a shared-memory concurrent architecture that allows bounds on divergence and stronger guarantees on the evolution of shared state.

Read-copy-update (RCU) [13] is a synchronization mechanism, suitable for a single-writer/multiple-readers scenario, that allows processes to read a shared object while a concurrent modification is in progress. Read-log-update (RLU) [16] is an improvement over RCU that allows concurrent writers. Unlike our model, concurrent writes are serialized using fine-grained locking.

Relaxed Consistency Models: Many models attempt to relax the strict semantics of linearizability [14] to achieve better performance. Quasilinearizability [3] allows each operation to be linearized at a different point at some bounded distance from its strict linearization point. Our work is complimentary to this model, allowing a combination of strong and weak updates to achieve different consistency semantics. Weak and medium future linearizability [15] is applicable to data types implemented using futures which allow reordering of the operations. Others models, such as k-linearizability [4] and quiescent consistency [22], also define the correctness based on some sequential history, possible reordered, of the operations. Local linearizability [12] requires that each *thread induced history* (a subset of each thread operations) is linearizable.

Mergeable Data Types: Conflict free Replicated Data Types (CRDTs) [19] provide deterministic merges and are now widely used in distributed replicated data systems. Here, we present implementations of mergeable data types that are tailored for shared memory concurrent programs. We benefit from a stronger

system model, where idempotence and merging among arbitrary replicas are no longer required, as local state is merged atomically to a single global state.

Even though no consolidated theory on mergeable data types exists in the shared memory ecosystem, there have been systems that use such types with restricted properties. Doppel [18] is a multi-core database that uses a mechanism called phase reconciliation to parallelize conflicting transactions. When a high contention workload is detected, Doppel switches to a split phase where the transaction updates per-core copy of the objects. At the end of the split phase, per-core copies are merged. Only operations that are commutative are executed in the split phase, thus guaranteeing serializability.

3 Global-Local View Model

The system we consider is built upon a classical shared-memory architecture as supported by specifications such as the C++ or Java memory models. We assume that the system consists of a variable number of threads. Any thread can spawn new threads that may outlive their parent thread. The system distinguishes two types of memory: local memory is associated to a single thread and can only be accessed by this thread; shared memory can be accessed by any thread. Communication and coordination between the threads are done via shared-memory objects; we assume that there are no side channels. In particular, spawned threads do not inherit local objects from their parents.

Each shared object o has a global view that is accessible by all threads that obtained a reference to it. In addition, each thread has its own local view of o. A thread may update and read its local view, but the view is not accessible by any other thread. The local updates are incorporated into the global copy when a merge operation is executed. Conflicting (non-commutative) updates from concurrent threads are resolved through a type-specific merge operation. In addition to the local updates and reads, we also provide updates and reads performed directly on the global view. This gives us flexibility for the data type semantics and the implementation of the underlying data structure.

An object in the global-local view model consists of a global view g, and for each thread identified by t, a non-shared local view consisting of two components, s_t and l_t. s_t denotes a local snapshot of the shared object state g which gets updated upon synchronization, and l_t refers to the local updates not yet incorporated in the shared global state g. The state variables – g, s_t, l_t – are each modeled as a sequence of updates, initially empty; a sequence x can be concatenated with another sequence y (or a single update), denoted by $x \cdot y$.

An operation opKind on an object performed by thread t can be formalized as a function

$$\mathsf{opKind}_t(m, g, s_t, l_t) = (r, g', s'_t, l'_t)$$

where m comprises the (optional) type-specific update (u) or query (q) method applied on the object. The operation returns a tuple (r, g', s'_t, l'_t) where r is the return value of the method m and the other variables refer to the updated global g' and local state s'_t, l'_t.

Following are the basic operations in the global-local view model; these are type-independent and mergeable data types typically implement only a subset of them:

$$\begin{aligned}
\mathsf{pull}_t(_, g, s_t, l_t) &= (\bot, g, g, l_t) \\
\mathsf{weakRead}_t(q, g, s_t, l_t) &= (q(s_t \cdot l_t), g, s_t, l_t) \\
\mathsf{strongRead}_t(q, g, s_t, l_t) &= (q(g \cdot l_t), g, s_t, l_t) \\
\mathsf{weakUpdate}_t(u, g, s_t, l_t) &= (s_t \cdot l_t \cdot u, g, s_t, l_t \cdot u) \\
\mathsf{strongUpdate}_t(u, g, s_t, l_t) &= (g \cdot u, g \cdot u, s_t, l_t) \\
\mathsf{merge}_t(_, g, s_t, l_t) &= (\bot, g', g', \bot) \qquad \text{where } g' = merge(g, (s_t, l_t))
\end{aligned}$$

pull updates the local object snapshot with the global object state; local operations are not modified. weakRead returns the result of a type-specific read-only operation q on the state obtained by applying local updates on the local snapshot. strongRead returns the result of a type-specific read-only operation q on the state obtained by applying local updates on global state. Neither the global state nor the local snapshot are modified. weakUpdate applies the update method u on the local copy without any synchronization to the global state. strongUpdate applies the update method u on the global state atomically. The previous weak updates that are batched in l_t are not merged at this point. merge incorporates the local updates to the global states and updates the local snapshot, using the type-specific $merge(g, (s_t, l_t))$ operation.

A merge must incorporate all local updates into the global state in a meaningful way, so that conflicting concurrent updates lead to a deterministic state. For example, if the updates are commutative, they can be appended to the global sequence $g' = g \cdot l_t$. If they are not commutative, the data types offer a conflict resolving merge operation, modifying the sequence of updates merged to g.

While weakRead and weakUpdate act exclusively on the local copy, strongRead and strongUpdate act on the global state. The combination of these two operations supports flexible optimizations on each given data type. For example, a queue can guarantee that an element is dequeued only once by executing dequeues in strongUpdate. At the same time, enqueues can use weakUpdate and merged later for better performance. For counters, we may enforce a weak limit on the maximum value, i.e. values should not diverge arbitrarily from the defined maximum value. We can use a strongRead to check the global value to adapt the merge interval or switch to a fully synchronized version.

4 Data Types

Each mergeable type defines a subset of the basic operations from the global-local view model, depending on the semantics needed. In this section, we discuss the specification of several data types and their implementation.

4.1 Specification

Given a sequential counter with methods *inc* (increments the counter by 1), and *value* (returns the current value), a purely mergeable counter implements the following operations.

- $\mathsf{weakValue}_t() = \mathsf{weakRead}_t(value, \_, s_t, l_t)$
- $\mathsf{weakInc}_t() = \mathsf{weakUpdate}_t(inc, \_, \_, l_t)$
- $merge(g, (s_t, l_t)) = g{\cdot}l_t$

The merge appends the local increments to the global sequence g, because the increments are commutative. A hybrid mergeable counter defines the following operations in addition to the above ones. The applications may choose weak or strong operations dynamically based on different criteria.

- $\mathsf{strongInc}_t() = \mathsf{strongUpdate}_t(inc, g, \_, \_)$
- $\mathsf{strongValue}_t() = \mathsf{strongRead}_t(value, g, \_, l_t)$

The queue datatype has operations $enqueue(e)$ and *dequeue*. A hybrid mergeable queue with mergeable enqueue and synchronized dequeue defines the following operations:

- $\mathsf{enqueue}_t(e) = \mathsf{weakUpdate}_t(enqueue(e), \_, \_, l_t)$
- $\mathsf{dequeue}_t() = \mathsf{strongUpdate}_t(dequeue, g, \_, \_)$
- $merge(g, (s_t, l_t)) = g{\cdot}l_t$

In the above semantics, if the global copy is empty, *dequeue* returns null even if there are local enqueue operations by the same thread which have not been merged yet. We can allow dequeue to include local enqueue operations by defining

$$\mathsf{dequeue}_t() = \mathsf{strongUpdate}_t(dequeue, g', \_, \_) \text{ with } (\_, g', \_, \_) = \mathsf{merge}_t(g, s_t, l_t).$$

In this way we can combine the operations to give different semantics. For example, a queue with weak enqueue and weak dequeue may be useful if redundant dequeue is not a problem for the application. A queue with only strong enqueue and strong dequeue behaves as a linearizable queue.

A grow-only bag is a set that provides only an *add* operation, and allows duplicate elements. A purely mergeable bag implements weakAdd and merge [7].

4.2 Implementation

The implementation of (hybrid) mergeable data types consists of two parts – a reference to the local view and another one to the global view.

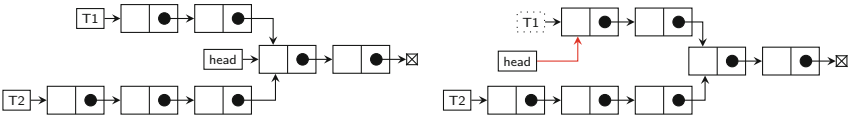
Counter. The global view of a mergeable counter is an integer g. The local view consists of a pair of integers `(s,l)`. The weak increments are collected in the thread-local state `l` and added to `g` during the merge. This design is inspired on *sloppy counters* [8], while using a local counter per thread instead of per core. The following pseudocode shows the implementation of a counter. It is easy to extend this implementation to allow decrements, explicit arguments for increments/decrements, and generalize to other commutative monoids.

```
type Counter: {
    int g,
    ThreadLocal int s,
    ThreadLocal int l
}
weakInc() {
    l++;
}
strongInc(){
     atomic {g++}
}
int weakValue(){
    return s+l;
}
int strongValue(){
    return g+l;
}
merge(){
    atomic {
        g += l; s = g; l = 0;
    }
}
```

A variable specified as `ThreadLocal` exists per thread in the thread's private storage. Many programming languages support some form of thread-local storage (TLS). A mergeable data type can also implement its own TLS by mapping thread ids to different instances of the local object. `atomic` refers to any synchronization mechanism such as mutex or lock-free techniques such as compare-and-swap or transactional memory that atomically executes the code block within.

For some data types, local views are isolated from each other and the global view, by maintaining a full copy of the object in each view. For large data structures, such as lists or trees, maintaining a full copy is not feasible. Thus, the local views may contain references to parts of the data structures that are shared by other local views and global view. The shared parts are not directly updated by the weak updates, but only read. For example, a *lookUp* on a list may first traverse the locally added items and then the shared parts of the list. The following are the designs of a few data types where this can be done efficiently and correctly without copying the entire data structure.

Grow-only Bag. A grow-only bag [7] is implemented using a multi-headed list as shown in Fig. 1. The thread local view consists of a pointer to the local head. A merge updates the global head of the list and does not change the local views of other threads. A lookup that traverses the list starting from the local head will never see an item that is concurrently added or merged.



(a) Two threads with different local views.

(b) After T1's local view is merged.

Fig. 1. Mergeable grow-only bag.

Queue. A hybrid mergeable queue can be implemented using a single-linked list similar to a linearizable queue. The items enqueued are added to the tail of the list, while dequeue is performed from the head. A mergeable queue instance contains a global view – (`head, tail`), which points to the head and tail nodes respectively of the global list and local view – (`ThreadLocal lhead, ThreadLocal ltail`), which are the head and the tail of the local list of each thread. The local list collects the items enqueued by the thread that are not yet merged. The merge atomically appends the local list to the global list. The time needed to merge a group of nodes is the same as the time needed to enqueue a single node. By batching the enqueues, we can reduce the number of synchronization operations, thus improving the overall throughput.

The *dequeue* operation directly updates the shared part of the list. For some data types, an update on the shared part of the data structure should preserve the old version, because local views may be keeping reference to it. However, there is no weakRead, such as a weak lookup, defined on queue that must observe a version before a concurrent dequeue. Hence, there is no need to keep those versions, which simplifies the implementation.

5 Applications

In this section, we sketch some application scenarios that benefit from multi-view mergeable data types.

A work-stealing queue is used to distribute tasks among threads running in parallel. In Cilk runtime [11] each thread owns a queue with operations *pushTop*, *popTop*, and *popBottom*. There is no *pushBottom*. When a thread is devoid of tasks, it retrieves one from its queue using *popTop*, executes it and may generate new tasks that are added to its queue using *pushTop*. When a thread's task queue is empty, it steals from other threads' queue using *popBottom*. A work stealing queue with this semantics is a natural fit to the global-local view model. Instead of a queue per thread, we have a multi-view queue with a global view and a local-view per thread. *pushTop* and *popTop* executes on the thread-local views, and *popBottom* on the global view. One downside of this design is that it may prevent threads from stealing tasks when the global view is empty even if there are unmerged tasks in the local views. To avoid this, threads can be forced to merge when the global view drops below a threshold.

In-memory multi-core databases. In high contention workloads, we can achieve high performance by allowing concurrent conflicting transactions to proceed in parallel on different cores. Instead of serializing the access to the objects, the transactions can update a per core copy of the object and merge them later. In [18], authors describe a system that automatically parallelize high contention transactions. A multi-view data type implemented in the global-local view model is a natural fit to such scenario.

Message queues where multiple messages can be batched together and added to the shared queue is a direct application of the hybrid queue described in this paper. The applications that use aggregation counters that are computed by

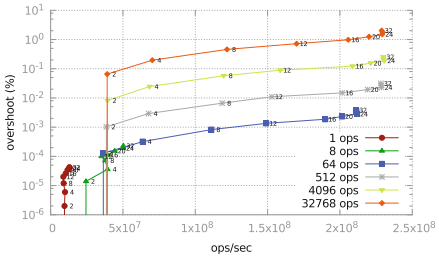


Fig. 2. Throughput vs Overshoot of mergeable counter. Points on the lines are labeled with the number of threads.

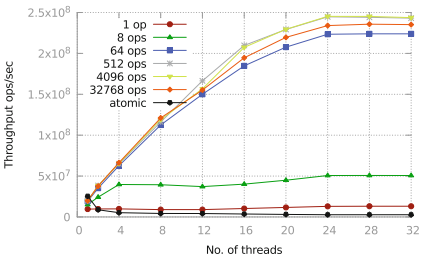


Fig. 3. Throughput of hybrid mergeable counter (overshoot free) vs atomic counter, labelled with merge-interval.

parallel threads can use our mergeable counter. Similarly, the objects that store statistical measures such as sums, min, max etc. that are computed by parallel threads will benefit from the global-local view model. In software transactional memory, we may use mergeable objects to avoid unnecessary aborts where the conflicting updates can be meaningfully merged [6].

6 Evaluation

We evaluated the performance and scalability of the mergeable counter and the hybrid mergeable queue using different micro-benchmarks. As an example of real applications, we employed the hybrid queue in a breadth-first traversal on graphs. We implemented the counter in C++ and the queue in Java. The evaluations are performed on a 12 core CPU (2 NUMA nodes) with 2-way hyper-threading.

Counter. We provide two variants of a mergeable counter and compare them with an atomic counter, implemented using the atomic compare and swap operation. In the first experiment, we allow threads to increment the shared purely mergeable counter until a *target* value is reached. Since threads might not know about non-merged increments from other threads, they typically end up overshooting the target. For this experiment, the *target* is set to 5×10^6 increments. We evaluated several merge-intervals, labelled with how many local increments are allowed between merges. Figure 2 shows that the throughput scales linearly with the number of threads and with the merge-intervals. At the same time, the overshoot increases. However, the percentage of the overshoot is small. (Notice that overshoot is upper bound by the number of threads times the merge-interval, as this reflects the amount of increments not yet accounted for.) The atomic counter never overshoots the *target*, but since threads are always competing on the increment, performance is very low and no speedup is obtained. In contrast, the mergeable counter can scale linearly up to a good fraction of the available concurrency, in particular with merge-interval of $\geq$4096.

While some applications could tolerate an overshoot, in general, applications will require to further bound the overshoot. To address this, we provide a variant

of the mergeable counter that makes a hybrid use of initial weak local increments and later switches to atomic strong increments when approaching the target. The first thread that, upon the periodic merges, detects that it is close to the target, initiates a barrier synchronization to ensure that all threads have switched to strong operations. Figure 3 shows that under this approach, overshoot is eliminated while the performance is mostly identical to the mergeable counter.

Comparison to CRDT. In this experiment, we demonstrate that CRDT designs have significant overhead in performance when used in a shared memory program. We implemented a CRDT counter on the global-local view model, where each local view and global view are a CRDT replica. We implemented the G-counter [19] using (1) a HashMap that maps thread-id to an integer, (2) an array where the array index corresponds to a thread id. Figure 4 shows that the array scales better when the merge-interval is large. However, the size of array must be fixed to the number of threads. The map implementation does not scale well because (1) there is an overhead in accessing the map entries, (2) merge requires an iteration over the entire map resulting in longer critical section. Thus, the cost of merge operation is negating the benefit achieved by the asynchronous local increment.

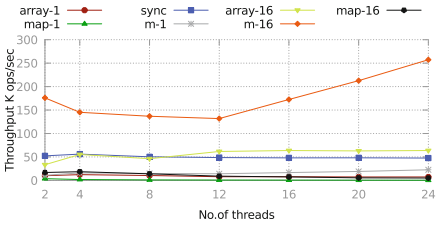


Fig. 4. CRDT counter using array and map, m-mergeable counter with merge-interval 1,16. sync-atomic counter.

Queue. To evaluate the scalability of hybrid mergeable queue (referred to as mergeable queue) in comparison to classical algorithms, we implemented four different queues in Java – (1) a lock-based linearizable queue based on Michael and Scott's 2-lock queue [17] (LL), (2) a lock-based mergeable queue which uses similar 2-lock mechanism (ML), (3) a lock-free linearizable queue adapted from Michael and Scott's lock-free queue [17] (LF) and (4) a lock-free mergeable queue (MLF). Figure 5 shows the time to perform a total of 5×10^6 enqueues and dequeues. We evaluated mergeable queues with different merge intervals m (a merge is performed by a thread after m enqueues). In this experiment, we forced half of the threads to run on one NUMA node and the other half on the second NUMA node. For both lock-based and lock-free versions, the mergeable queue is faster than the linearizable counterpart. Since this is a high-contention workload, the lock-based version performs better than the lock-free version. Unlike the mergeable counter, increasing the merge interval from 8 to 64 does not improve the performance significantly because *dequeue* is always executed synchronously which shadows the performance gain from asynchronous *enqueue*s.

Breadth-First Traversal. A standard breadth-first traversal algorithm using queues can be parallelized using concurrent queues. We evaluated four versions

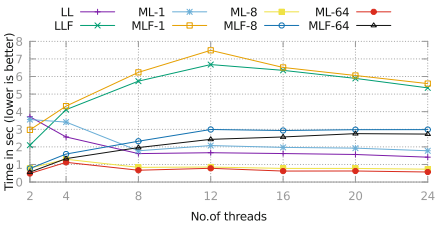


Fig. 5. Queue. linearizable lock based (LL), lock-free (LF). mergeable lock based (ML), lock-free (MLF) 1,8,64-merge interval.

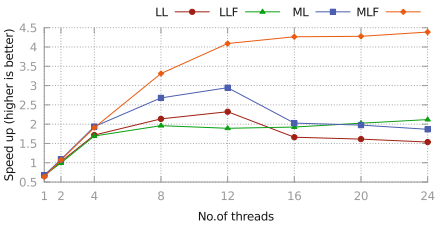


Fig. 6. Breadth-first traversal. linearizable lock based (LL), lock free (LLF). mergeable lock-based (ML), lock-free (MLF).

of the algorithm using different queue implementations, that traversed randomly generated graphs of size of 2×10^6 vertices and 2×10^7 edges. Unlike the micro-benchmark for the queue, there is no fixed merge interval. The threads merge their local queue at the end of processing each level. Figure 6 shows the speedup of each version compared to a single-threaded implementation. Mergeable queues scale better than their linearizable counterparts. The speedup of the lock-free mergeable queue is significantly higher than that of the others, and scales almost linearly until 16 threads. Beyond 16 threads, the number of vertices processed by each thread at each level is reduced, as they are divided among the threads, leading to smaller merge frequencies. We believe the sudden drop in the speedup of lock-based queues after 12 threads is due to the additional cost in synchronization to the second NUMA core. This is a low-contention workload because a significant amount of time is spent in processing the nodes rather than updating the queue.

7 Conclusion

Incorporating more information about the respective datatype semantics is crucial for datatype designs that are more parsimonious regarding synchronization. CRDTs succeeded in capturing datatypes with clear concurrency semantics and are now common components in industry. However, they do not migrate trivially to shared-memory architectures due to high computational costs from merge functions, which becomes apparent once network communication is removed.

In this paper, we define the *global-local view* model as base for a framework that allows capturing the semantics of multi-view datatypes. The *global-local view* distinguishes between local fast state and distant shared state where operations need to be synchronized. This distinction allows the datatype designer to explore the trade-offs in the design when using weak or strong operations. Our approach enables speedups in order of magnitudes while preserving the datatypes' target behavior. It is quite possible that further increments of the number of components involved will lead to a hierarchical model with more levels than the current binary, local vs global, scheme.

Data Availability Statement and Acknowledgements. The work presented was partially supported by EU H2020 LightKone project (732505), and SMILES Research Line within project "TEC4Growth - Pervasive Intelligence, Enhancers and Proofs of Concept with Industrial Impact /NORTE-01- 0145-FEDER-000020" financed by the North Portugal Regional Operational Programme (NORTE 2020), under the PORTUGAL 2020 Partnership Agreement, and through the European Regional Development Fund (ERDF).

The datasets and code generated during and/or analysed during the current study [5] are available in the figshare repository: https://doi.org/10.6084/m9.figshare.6383807

References

1. Antidotedb. http://syncfree.github.io/antidote/
2. Riak KV: a distributed key-value database. http://basho.com/products/riak-kv/
3. Afek, Y., Korland, G., Yanovsky, E.: Quasi-linearizability: relaxed consistency for improved concurrency. In: Lu, C., Masuzawa, T., Mosbah, M. (eds.) OPODIS 2010. LNCS, vol. 6490, pp. 395–410. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17653-1_29. http://dl.acm.org/citation.cfm?id=1940234.1940273
4. Aiyer, A., Alvisi, L., Bazzi, R.A.: On the availability of non-strict quorum systems. In: Fraigniaud, P. (ed.) DISC 2005. LNCS, vol. 3724, pp. 48–62. Springer, Heidelberg (2005). https://doi.org/10.1007/11561927_6
5. Akkoorath, D., Brando, J., Bieniusa, A., Baquero, C.: Code to run experiments for euro-par 2018 paper: Global-local view: Scalable consistency for concurrent data types (2018). https://doi.org/10.6084/m9.figshare.6383807
6. Akkoorath, D.D., Bieniusa, A.: Transactions on mergeable objects. In: Feng, X., Park, S. (eds.) APLAS 2015. LNCS, vol. 9458, pp. 427–444. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-26529-2_23
7. Akkoorath, D.D., Bieniusa, A.: Highly-scalable concurrent objects. In: Proceedings of the 2nd Workshop on the Principles and Practice of Consistency for Distributed Data. PaPoC 2016, pp. 13:1–13:4. ACM, New York (2016). https://doi.org/10.1145/2911151.2911158
8. Boyd-Wickizer, S., et al.: An analysis of linux scalability to many cores. In: Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation. OSDI 2010, pp. 1–16. USENIX Association, Berkeley (2010). http://dl.acm.org/citation.cfm?id=1924943.1924944
9. Burckhardt, S., Baldassin, A., Leijen, D.: Concurrent programming with revisions and isolation types. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications. OOPSLA 2010, pp. 691–707. ACM, New York (2010). https://doi.org/10.1145/1869459.1869515
10. Burckhardt, S., Leijen, D., Protzenko, J., Fähndrich, M.: Global sequence protocol: a robust abstraction for replicated shared state. In: Boyland, J.T. (ed.) 29th European Conference on Object-Oriented Programming (ECOOP 2015). Leibniz International Proceedings in Informatics (LIPIcs), vol. 37, pp. 568–590. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2015). https://doi.org/10.4230/LIPIcs.ECOOP.2015.568
11. Frigo, M., Leiserson, C.E., Randall, K.H.: The implementation of the Cilk-5 multithreaded language. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI 1998, pp. 212–223. ACM, New York (1998). https://doi.org/10.1145/277650.277725

12. Haas, A., et al.: Local Linearizability for Concurrent Container-Type Data Structures. In: 27th International Conference on Concurrency Theory (CONCUR 2016). Leibniz International Proceedings in Informatics (LIPIcs), vol. 59, pp. 6:1–6:15 (2016). https://doi.org/10.4230/LIPIcs.CONCUR.2016.6
13. Hart, T.E., McKenney, P.E., Brown, A.D.: Making lockless synchronization fast: performance implications of memory reclamation. In: Proceedings of the 20th International Conference on Parallel and Distributed Processing. IPDPS 2006, p. 21. IEEE Computer Society, Washington, D.C. (2006). http://dl.acm.org/citation.cfm?id=1898953.1898956
14. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3), 463–492 (1990). https://doi.org/10.1145/78969.78972
15. Kogan, A., Herlihy, M.: The future(s) of shared data structures. In: Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing. PODC 2014, pp. 30–39. ACM, New York, (2014). https://doi.org/10.1145/2611462.2611496
16. Matveev, A., Shavit, N., Felber, P., Marlier, P.: Read-log-update: a lightweight synchronization mechanism for concurrent programming. In: Proceedings of the 25th Symposium on Operating Systems Principles. SOSP 2015, pp. 168–183. ACM, New York (2015). https://doi.org/10.1145/2815400.2815406
17. Michael, M.M., Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing. PODC 1996, pp. 267–275. ACM, New York (1996). https://doi.org/10.1145/248052.248106
18. Narula, N., Cutler, C., Kohler, E., Morris, R.: Phase reconciliation for contended in-memory transactions. In: Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation. OSDI 2014, pp. 511–524. USENIX Association, Berkeley (2014). http://dl.acm.org/citation.cfm?id=2685048.2685088
19. Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: Conflict-free replicated data types. In: Défago, X., Petit, F., Villain, V. (eds.) SSS 2011. LNCS, vol. 6976, pp. 386–400. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24550-3_29. http://dl.acm.org/citation.cfm?id=2050613.2050642
20. Shavit, N.: Data structures in the multicore age. Commun. ACM **54**(3), 76–84 (2011). https://doi.org/10.1145/1897852.1897873
21. Sstrik, M.: ZeroMQ. In: The Architecture of Open Source Applications, vol. 2 (2012)
22. Viotti, P., Vukolić, M.: Consistency in non-transactional distributed storage systems. ACM Comput. Surv. **49**(1), 19:1–19:34 (2016). https://doi.org/10.1145/2926965