Scalable Data-structures with Hierarchical, Distributed Delegation

Yuxin Ren
The George Washington University
ryx@gwmail.gwu.edu

Gabriel Parmer
The George Washington University
gparmer@gwu.edu

ABSTRACT

Scaling data-structures up to the increasing number of cores provided by modern systems is challenging. The quest for scalability is complicated by the non-uniform memory accesses (NUMA) of multi-socket machines that often prohibit the effective use of data-structures that span memory localities. Conventional shared memory data-structures using efficient non-blocking or lock-based implementations inevitably suffer from cache-coherency overheads, and non-local memory accesses between sockets. Multi-socket systems are common in cloud hardware, and many products are pushing shared memory systems to greater scales, thus making the ability to scale data-structures all the more pressing.

In this paper, we present the *Distributed, Delegated Parallel Sections* (DPS) runtime system that uses message-passing to move the computation on portions of data-structures between memory localities, while leveraging efficient shared memory implementations within each locality to harness efficient parallelism. We show through a series of data-structure scalability evaluations, and through an adaptation of `memcached`, that DPS enables strong data-structure scalability. DPS provides more than a factor of 3.1 improvements in throughput, and 23x decreases in tail latency for `memcached`.

CCS CONCEPTS

• **Computing methodologies → Parallel algorithms**.

KEYWORDS

concurrent data-structure, NUMA locality, delegation

Middleware '19, December 8–13, 2019, Davis, CA, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-7009-7/19/12…$15.00
https://doi.org/10.1145/3361525.3361537

ACM Reference Format:
Yuxin Ren and Gabriel Parmer. 2019. Scalable Data-structures with Hierarchical, Distributed Delegation. In *20th International Middleware Conference (Middleware '19), December 8–13, 2019, Davis, CA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3361525.3361537

1 INTRODUCTION

Even more than a decade since the multicore revolution started, it is still depressingly difficult to scale basic data-structures to an increasing number of cores and sockets. The fundamental challenge is that shared memory data-structures must maintain consistency between parallel operations, and using locks or various techniques that rely on atomic instructions cause expensive cross-core and -socket memory traffic. The cache-coherency resulting from stores to shared memory – either due to lock's implementation, or to modifications to data-structure – cause inter-core and -socket messages. These memory accesses manifest in overheads that can prevent scalability, and even decrease the aggregate throughput of data-structure operations.

Locks provide mutual exclusion, thus serializing execution from multiple cores and limiting the upper bounds on speedup with increasing numbers of cores. Other approaches such as lock-free data-structures avoid critical sections by instead using retry semantics, and linearizability [24] is used to reason about correctness. In this way, they trade often increased parallelism for complexity. As we'll show later, such approaches scale well within a socket (when sharing only a last-level-cache (LLC)), but succumb to the significant overheads of non-local memory access across sockets.

To avoid the cache-coherency and remote memory overheads, delegation [42] serializes data-structure access *using a "server core"*. All "client cores" use message passing to pass requests to the server, and it performs the data-structure operations on behalf of the clients. As the data-structure is only accessed by the server, no remote memory accesses are made, nor is there contention. In contrast to shared memory data-structures that move *memory to computation*, delegation moves *computation to the memory*. However, this approach is limited by the throughput of the server core, and requires total allocation of the server core whose computation is used for polling. Additionally, the separation of clients and server

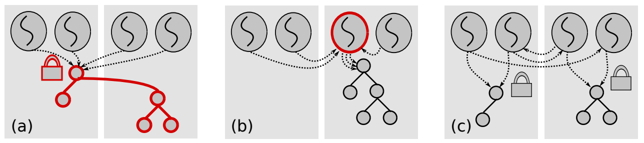


Figure 1: (a) Shared memory, (b) delegation, and (c) DPS-based data-structures. Circles with threads are cores, squares are sockets (*i.e.* memory localities), and trees are data-structures that are resident in the memory for a specific socket, and might be synchronized using locks or non-blocking means.

cores means that data-structures with large working sets only use the server's cache capacity, missing the opportunity to use the aggregate LLC capacity.

In this paper we introduce *Distributed, Delegated Parallel Sections* (DPS), which provides a set of mechanisms and abstractions to *partition* data-structure namespace across a number of *memory localities* (*e.g.* sockets). In doing so, operations use either local accesses, or a form of optimized, possibly asynchronous, delegation between *peer* computations. An underlying set of observations motivate the structure and mechanisms in DPS: First, shared-memory data-structures are often very efficient when parallel accesses share a LLC. Second, using message passing to distribute computation between sockets effectively avoids remote memory accesses. Figure 1 depicts shared memory, delegation-based, and DPS-based data-structures. Darker, red lines denote the bottleneck to scalability. Shared memory structures are shared between sockets, thus resulting in synchronization and memory overheads, while delegation moves all data-structure computation to a server core. DPS partitions the *namespaces* of the items stored in the data-structure, *moves computation to the memory locality* (not server) that contains that item, and uses the *efficient parallelism of shared structures where they are effective*. Any core can perform data-structure operations within its locality, on behalf of others, thus allowing data-structure computations to scale across sockets.

The contributions of this research include (i) the design of DPS that both avoids remote memory accesses, and effectively uses parallelism, (ii) the implementation of DPS provides efficient and scalable distribution of data-structures while enabling locality-aware computation, (iii) the application of DPS to existing state-of-the-art concurrent data-structures, and an evaluation that demonstrates significant gains in performance and scalability for microbenchmarks, concurrent data-structures, and the `memcached` application.

2 BACKGROUND AND MOTIVATION

Shared memory data-structures. To better understand the properties of existing parallel data-structure approaches, we evaluate a number of configurations on a four socket system (see §5 more hardware details). Figure 2 plots performance of the (fine-grained) lock-based (`lb-`) and lock-free (`lf-`) versions of a binary search tree (`bst`) and skiplist (`sl`) from ASCYLIB [3]. The top and bottom graphs show throughput and cache misses per operation, respectively. The left graphs vary the workload's update fraction which increase cache coherency traffic and contention. In both graphs, the data-structure fits into L2 cache. The right graphs vary the data-structure size which increases cache capacity misses, thus increasing non-local memory accesses.

The differences between lock-free and lock-based data-structures have been thoroughly investigated in [14]. A number of trends are shared by both. Increasing the number of necessary stores to the data-structure (due to the synchronization to coordinate them) and the update rate forces cache coherency and (remote) memory access overheads. As data-structure size increases, capacity cache misses cause significant increases in the number of (remote) memory accesses. As the data-structure grows past the size of the LLC capacity supported by *each* socket (as opposed to aggregate LLC capacity), capacity misses significantly decrease throughput.

Delegation-based data-structure serialization. Delegation partitions cores into those running non-data-structure code (clients), and cores performing data-structure operations on behalf of the others (servers). `ffwd` [42] provides an optimized implementation of this. Unfortunately, data-structure accesses are serialized by servers, and as the data-structure operations increase in length, or the number of clients increases, data-structure computation throughput

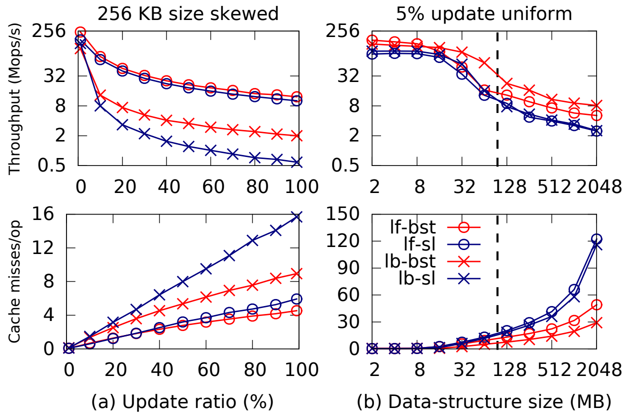


Figure 2: The throughput and cache misses per operation for binary tree and skiplist on a 40 core (80 hyperthread) system. The vertical line represents aggregate LLC capacity.

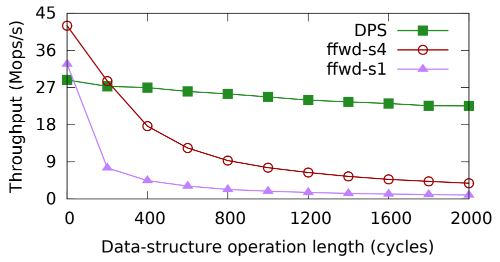


Figure 3: `ffwd` and DPS throughput with variant data-structure operation length on 40 cores (80 hyperthreads).

	complexity	coherence	locality	parallelism
lock-based	easy	large	poor	low
non-blocking	hard	medium	poor	high
delegation	easy	none	good	low
DPS	easy	none	good	highest

Table 1: Comparison of data-structure implementations.

proportionally decreases. Figure 3 shows `ffwd`'s throughput with an increasing data-structure operation length using 1 (`s1`) and 4 (`s4`) servers. Note that using one server is more common as any data-structure with a serial implementation is easily adapted to it.

Discussion. These results demonstrate the limitations of the existing approaches. Importantly, their weaknesses are often due to different factors:

- *Shared data-structures* fail to scale on NUMA machines due to cache-coherency, and capacity misses that cause remote memory accesses.
- *Delegation* is limited by the capacity of a statically fixed set of data-structure servers.

DPS carefully combines both approaches to avoid these weaknesses by partitioning the data-structure item's namespace between localities, and moving associated data-structure operations to those localities. Concurrent data-structures are used *within* a memory locality (which includes multiple cores). This enables a key balance: *concurrent data-structures are used up to, but not past their scalability limits*, and *distribution of computation is used to move computation to the locality charged with the relevant portion of the data-structure.*

Table 1 summarizes this comparison. Non-blocking data-structures tend to be complicated, and the shared memory implementations have non-trivial coherence overheads, and since data-structure memory is spread across sockets, locality is poor. Delegation's parallelism is limited by the server core's capacity.

3 DPS DESIGN

DPS is designed to provide a simple interface that interposes between the client performing operations on the data-structure, and the data-structure operation logic itself. Its runtime transparently moves computation to the locality containing the requested data-structure items, and distributes computation within that locality between multiple cores. Therefore, it harnesses the parallelism of each core in the system yet avoids remote memory accesses.

3.1 Interface

DPS provides a small and simple API:

- dps_t create(ds_init_fn, ds_args, partition_cnt, ns_sz, hash_fn)

Create a data-structure accessible via the DPS interface. The ds_init_fn and ds_args are a data-structure specific initialization function, and arguments to it. The partition_cnt, ns_sz and hash_fn specify the number of partitions the key namespace is spread across, size of the key namespace, and the hash function to distribute keys across the namespace. Though not listed here, there is a corresponding destroy function.

- completion_rec_t execute(ps_t dps, key, op_fn, args. . .)

Execute an operation (op_fn) with the specified arguments on the dps DPS instance. The operation is to be performed on data associated with key that is used to determine the locality on which to execute the operation. This function does *not* necessarily return the value of the data-structure's execution as it might execute asynchronously on a different socket. Instead it returns a pointer to a completion record which is used to determine when the return value is available.

- await_completion(completion_rec_t, *retval)

This function is called on a completion record, and returns a boolean value to indicate if the data-structure's output is available. When the computation is done, the retval is populated with the computation's output.

The separation between execute and await_completion enables DPS to use peer-based inter-socket communication to move computation to the data-structure, and makes the data-structure partitioning explicit. To avoid changing the structure of client code, a synchronous API emulating a simple function call is easily built from this interface by directly following execute with a loop on await_completion. Though we use synchronous semantics for client code in most examples, we will describe an optimization using asynchronous communication implemented on the same API (§4.4). Please note, DPS does not offer synchronization to data-structures it manages. It relies on users to provide their concurrent implementation to execute.

3.2 System Architecture

Figure 4 depicts DPS's system architecture. Data-structure operation requests come from clients into the DPS runtime, which then either executes them locally if they are destined for the requesting core's partition, or pass them to the destination locality via efficient and scalable message passing structures. Any core within a locality that requests execution or awaits a computation completion will *also* process requests being made to its partition. This is the core of the *peer-based distributed delegation* used by DPS, since every core is used for data-structure processing. This avoids constraining the parallelism caused by devoting specific cores solely to data-structure processing (as in delegation).

The design optimizes around a number of goals:

Memory locality. By splitting the data-structure namespace (indexed by *key*) across partitions that each map to the underlying memory organization of the system, computation is performed where it has high locality to the memory, within a single the LLC and socket.

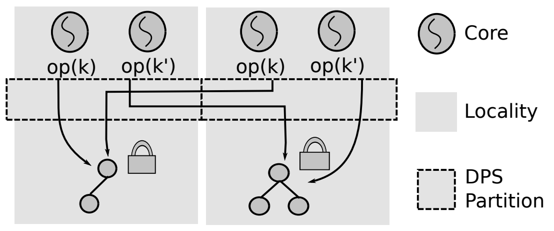


Figure 4: DPS architecture. In DPS, each namespace partition binds a portion of the data-structure to the cores and memory on its associated locality. An operation is executed either locally or remotely according to its key.

Workload- and data-structure-driven parallelism. The parallelism of DPS is driven by the workload, and by the scalability of the data-structure implementations. First, since each core is active in processing data-structure operations within its locality, the amount of parallelism of DPS is bound only by the number of cores in the system. Second, within a locality, all cores access their data-structure using concurrent data-structures that can scale well up to the locality size. The hashing function is chosen to spread computation among partitions as much as possible to leverage inter-partition parallelism and the distributed bandwidth of NUMA memories.

Inter-partition communication efficiency. A message-passing layer transfers data-structure operation requests between partitions. This communication layer is implemented using shared memory structures, and as it is in the fast-path to execute data-structure operations, it must be optimized. Two factors impact cost of delegation path, the costs of synchronization and cache efficiency. The data-structures backing the communication paths focus on using wait-free structures that guarantee each core's progress without spinning, and replicate them to ensure that cache-lines are pair-wise shared between requesting cores and partitions, to avoid globally shared, non-scalable cache-line modifications.

Maximize resource utilization. In previous delegation systems, some cores are reserved for server threads and cannot carry out other work. Additionally, after sending a request to a server, a synchronous client naively spins waiting for a reply. Such spinning wastes CPU cycles and decreases whole system utilization as operation length increases (see Figure 3). In contrast, since each core can execute both client code and data-structure operations in DPS, when execute and await_completion are called, they additionally query for data-structure operation requests from *other* partitions. Thus they overlap data-structure computation in their partition, with their own requests for other partitions. This significantly increases throughput, and enables the peer-based computation of data-structure logic on all cores.

3.3 Assumptions and Consistency

To split a data-structure across multiple localities, DPS makes some assumptions about the properties of the data-structure and execution environment. Each item in the data-structure must be associated with a *key* (*e.g.* a hash). This defines a namespace for the data-structure, which DPS partitions across localities. Though it is possible to perform the partitioning dynamically, and balance between sockets, in this paper we assume a static partitioning.

By leveraging concurrent implementations, DPS preserves any ordering and consistency guarantees of that implementation (*e.g.* linearizability [24]) *within a partition.* However, data-structure operations that touch data spread across *different partitions* are issued to the corresponding localities in parallel. The DPS prototype does not maintain ordering guarantees across all partitions. To discuss the consistency properties of the system, we leverage the taxonomies of consistency from Highly Available Transactions [4] and Adya [50].

Despite the lack of ordering guarantees across partitions, DPS provides "read you writes" and monotonic writes as it focuses on partitioning, not replication of data. Simply put, a thread that writes two values will see (read) those writes in order. However, it does not, by default, provide transaction-level consistency such as read uncommitted. That is to say, dirty writes [6] are possible: two parallel writes across partitions can be committed to partitions in different orders. However, we observe that strong consistency can be provided in a few common situations:

Each operation's modifications are local to a single partition. DPS inherits the *same* consistency properties if *data-structure writes each only modify a single partition.* This is a relatively common-case as data-structure updates often impact only a single data value. Even common APIs for B+-trees (a range-optimized data-structure) often only provide single value insert and delete. A study of key-value stores in [52] shows that 3/4 of the studied systems provide no consistency for multi-key operations, thus making DPS trivially applicable to most studied data-structures. If data-structures must provide atomic updates to *multiple* data-values, the user-provided hash function can partition the data such that all of an operation's updates are localized to a single partition.

Application-assisted serialization. A large body of highly concurrent data-structures [8, 12, 28, 30, 32, 43] do not provide transaction-level consistency solely by their implementation, and need application assistance to support transactions. For example, all data-structure operations submitted to the data-structure can be tagged with a monotonically increasing timestamp that the backing data-structure uses to serialize updates. This avoids dirty writes [5], and provides *read uncommitted* consistency. Anna [52] uses this technique to order updates. Silo [49], as another example, supports transaction by adding additional transaction version and locks on top of Masstree [32]. The downside is that data-structures must be designed to adhere to the application specific logic (such as version or timestamp ordering),

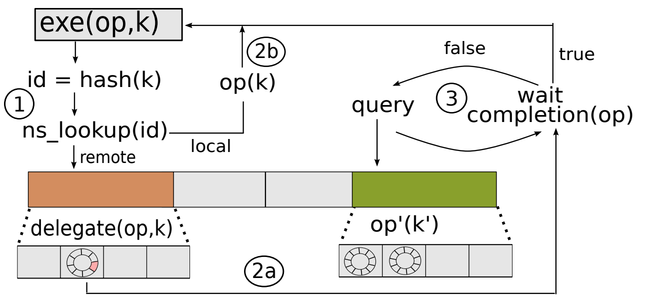


Figure 5: The work-flow of DPS on a single core. The hashing value of the operating key is used for namespace look-up and to determine the target locality (1). If a remote locality is determined, the operation is delegated to the destination (2a); otherwise, the operation is executed locally (2b). While waiting for delegation reply, the thread queries for and executes operations moved to its own locality (3).

but these technique are widely used in practical systems to provide read uncommited consistency.

Relaxed semantics and weak consistency are common [1, 19, 22] and desirable for such concurrent data-structures as they can provide increased performance. Motivated by the inherent difficulty in providing strong consistency (*e.g.* linearizability) with high availability in a distributed environment (see the CAP Theorem [7] and HAT [4]), §4.4 discusses DPS extensions which further relax consistency to further increase performance.

3.4 Generality

DPS trivially supports operations that perform data-structure reads, and single-partition updates. In this paper, we mainly evaluate DPS support for these data-structures. However, some data-structure operations require context across the entire data-structure. For example, sorting and priority queues encode an ordering constraint across all data items, and stack and queue operations encode relative insertion-time constraints. DPS broadcasts an operation across all partitions and determine on which partition to perform further operations. For example, to support a priority queue in §5, DPS peeks at the head of each partition's queue, and dequeues from the one with the highest priority.

4 DPS IMPLEMENTATION

DPS is implemented as a run-time library on top of ParSec [51], which provides memory reclamation, locking support and namespace management. The DPS runtime partitions namespace, delegates data-structure operations to proper partitions and serves remote computation requests. Figure 5 gives the overview of the work-flow on a single core. When DPS handles a data-structure operation, it first looks up the operating key in the data-structure's namespace to determine the operation's destination partition (§4.1) (step (1)). If the determined (orange, left) partition is not local, DPS migrates the operation to the remote partition (§4.2) (step (2a)). Instead of busy waiting for the response, DPS queries and processes data-structure operations within its (green, right) partition that are requested by others (§4.3, see step (3)). If instead the operation is local (or if a query determines another core has performed a delegation), it is carried out immediately on the data-structure (step (2b)).

4.1 Namespace Partition and Lookup

A DPS namespace partition consists of a subset of the data-structure key space. A data-structure node belongs to a partition when its key falls into the range of that partition. Each namespace partition is bound to a subset of cores that all share a single NUMA node that we call a "locality" (as in "memory locality"). Only threads running within that locality can create or manipulate portions of data-structure belonging to the corresponding partition. Moreover, memory are by default allocated on NUMA node local to the allocating threads. Thus, DPS creates a one-to-one mapping between a namespace partition and a designated NUMA memory locality and threads running within that NUMA node.

The size and count of partitions are assigned by clients. Usually one should choose the locality size smaller than the scalability "knee" of the managed concurrent data-structure. A DPS partition structure holds both its metadata (such as its id and size), and data-structure-specific data for the partition (such as the data-structure root and lock). DPS exposes additional interfaces to set and retrieve the user specified data. Furthermore, the lookup in ParSec namespaces is synchronization-free, incurring negligible overhead in DPS.

In order to convert arbitrary key (*e.g.* strings) into the flat scalar ParSec namespace, DPS first hashes the key into an integer using a (user-provided) hash function. This function gives applications the ability to control the namespace mapping. For instance, users can use an uniform hash to evenly distribute hot keys or use a consistent hash to preserve some locality in the original key space.

4.2 Task Delegation

If a key's partition is the local (*i.e.* the current core is in its locality), DPS executes the operation locally via function call. Otherwise, it delegates the operation to the remote locality via message passing. For simplicity, the current DPS implementation packs both delegation request and completion record into the same message. Each message occupies one cache line and contains a pointer to the target partition, a function pointer, a return value, a toggle bit and up to 4 arguments. The toggle bit is set by threads when they send new requests and is unset by the remote partition when it finishes processing the request.

Each (core, locality) pair maintains a dedicated fixed size ring buffer of messages, which is allocated on the locality's NUMA node. Thanks to the toggle bit in request structure, we do not need to compare `head` and `tail` of the ring buffer to check its status. When a sending thread finds an entry whose toggle bit is set, the ring buffer is full, and it waits for remote partition to finish. Similar logic applies to the receive-side, which is discussed next.

4.3 Overlapped Delegation Request and Data-Structure Processing

To better utilize parallel resources, DPS processing in a locality neither reserves cores nor busy waits. All threads in the DPS runtime are identical peers. They both delegate operations to remote localities, and serve requests delegated to the current locality. DPS accomplishes this by internally switching the role of a thread. Specifically, as shown in Figure 5, after a thread sends a request message, it goes to check its assigned ring buffers (§4.2), and handles requests if there are any. As a result, a thread alternatively checks for its response, and for requests from remote localities within a loop until the response is available. However, if multiple cores in the same locality concurrently retrieve delegation requests from the same ring buffer synchronization (and the associated overhead) is necessary. To avoid synchronization around the ring buffer, we organize request ring buffers of the same locality into a flat array, and this array is also divided across cores in the partition. In this way, different cores access disjoint parts of the array of ring buffers, and need no synchronization. DPS exposes a parameter to determine the number of checks performed on the ring buffer for each of its own requests. This enables users to control the trade-off between latency of local and remote operations. Though that parameter is static in our current implementation, a back-off strategy can adjust it adaptively based on the workload.

4.4 Extensions and Optimizations

The basic implementation of DPS is simple, but we've extended and optimized it in several important ways:

Range operations. Operations over a range of keys are challenging as the key range can be scattered over multiple partitions. To support range operations, DPS offers an additional API, which broadcasts range operations to all localities, and requires users to provide an aggregate function to merge the results. Through this simple mechanism, DPS supports a large range of operations. For example, to support `findMin` method in priority queue, we use an aggregation function to return the object with the smallest key among all localities' output. However, the range operation is not linearizable, because the execution of aggregation and parallel data-structure operations in each partition are not atomic.

Local execution. DPS migrates computations to a remote locality and in doing so it pays the cost of message passing. If the cost of message passing is larger than the cost of memory locality, then delegation will decrease performance. Thus, it is valuable to control whether an operation is executed locally or remotely. An additional DPS API allows local execution of specific operations. As DPS works over the already linearizable concurrent data-structure, the computational locality does not impact operation's correctness. However, memory modifications and coordination across sockets are expensive (motivating DPS in the first place). Thus, we mainly use local execution of operations for read-only data-structure operations, especially with lock-free data-structures. With this, DPS enables the flexible use of computation placement to best trade delegation versus shared memory coordination.

Asynchronous execution. Computation *outside* of data-structure processing can increase DPS latency (we study this effect in §5.1). This latency can be hidden when an operation has no return values, it does not necessarily need to wait for a response. In such a case, DPS includes an asynchronous API, which returns immediately after delivering requests for remote execution. However, asynchronous executions accumulate requests and can fill up the ring buffer. In that case, the thread waits for an available request slot, while performing operations delegated to it. Asynchronous requests reduce cache-coherency traffic, and further overlap client and data-structure computation. `await_completion` needs to be inserted properly as a barrier to enforce ordering of dependent asynchronous operations. DPS with asynchronous execution can be easily integrated into an event-driven programming model, which we leave as future work.

Liveness and responsiveness. If a thread is blocked, preempted or doing computation outside DPS, it cannot respond to delegation requests in a timely manner. To improve responsiveness, as shown in §5.1, applications can use the asynchronous API to hide the latency caused by non-DPS operations. To ensure liveness, one can also devote one core in each locality only for delegation handling. For this purpose, DPS exposes an interface which is used to check all request ring buffers in the requesting locality and process any pending delegations. As the designated core causes concurrent access to request ring buffers, each ring buffer is extended with a lock which rarely contended only when the designated core polls incoming ring-buffers.

4.5 Porting Code to DPS

Overall, porting data-structure to DPS is straightforward and requires little effort. Global variables need to be turned into partition-wide variables, and are retrieved through DPS partitions. Additionally, the underneath memory allocator

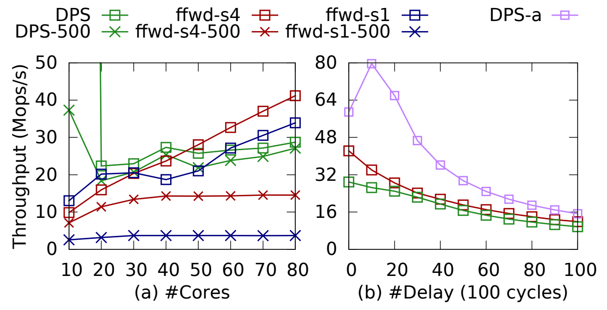


Figure 6: Delegation performance

is required to promote NUMA-local memory allocation. Except to `memcached`, we port all data-structures used in §5 with less than 50 lines modification. Because `memcached` uses a large number of global variables, porting `memcached` involves more effort: 1598 lines of code are modified. However, all those modifications are mechanical, and no core `memcached` logic (such as slab allocator and LRU lists) is changed. To ease data-structure partitioning, DPS provides macros to define and use partition-wide variables (similar to per-cpu variables in the Linux kernel).

5 EVALUATION

We use a system with 48 GB RAM and four Intel Xeon E7-4850 sockets. Each socket is a NUMA node with 10 cores, a shared 24 MB L3 cache, and a private per-core L2 (L1) cache of 256 (64) KB. Cache lines are 64 bytes, and our processor fetches cache-lines from memory as 128 byte aligned regions. Each core operates at 2.0 GHz and has two hyperthreads, leading to 80 hyperthreads in total. All experiments are running under 64 bit Ubuntu 14.04 with Linux kernel 4.4. All threads are pinned to hyperthreads. For tests with n "cores", n is the number of hyperthreads. For a value of n, we allocate a minimal number of sockets with a single hyperthread per core, then ($n > 40$) add hyperthreads across a minimal number of sockets. We report the aggregate throughput across all testing threads, and each result is the average value of 3 repetitions of 10 second runs. All micro benchmarks (§5.1) choose keys uniformly, while §5.2 and §5.3 discuss skewed workloads . DPS fixes its locality size to 10 hyperthreads, which are spread across the minimal number of cores and sockets. Unless otherwise stated, DPS runs in synchronous mode without the local execution optimization. The default NUMA memory allocation policy is `node local`.

5.1 Micro-benchmarks

Delegation Overhead. We first perform micro-benchmarks to compare delegation overhead of DPS with ffwd [42]. For ffwd, we measure its performance using both a single server (`s1`) and four servers (`s4`), the maximal number of servers it currently supports. To focus on the overhead of the delegation mechanisms, the operation just spins for a specified amount of time. Figure 6(a) shows the aggregate throughput with two operation lengths: empty and 500 cycles.

First, we examine empty operations. At very low core counts, DPS's throughput is high as all execution is via function call within a locality. When core count is smaller than 60, DPS has higher throughput than `ffwd-s1`, because DPS harnesses more parallelism. However, with more cores, the benefits of ffwd's batching (very short) delegation requests becomes more apparent, resulting in higher throughput than DPS. `ffwd-s4` has lower throughput than DPS with less than 40 cores, because ffwd incurs more cache traffic than DPS. ffwd statically binds one server to one socket which means that it incurs cache traffic across *all* sockets, even when client threads need fewer sockets. After all sockets are involved, `ffwd-s4` starts to get higher throughput than DPS. To analyse the batch optimization impact on ffwd performance, let us consider the number of cache transactions in the delegation path. A ffwd client has two cache-coherency operations per request (one for sending and one for receiving). A server has one cache miss for receiving each request, but only one cache coherency operation for sending a *batch* of (up to 15) responses. When able to maximally batch replies, ffwd imposes 46 cache operations per batch, while the same number of operations in DPS causes 60, a 30% increase. These results confirm that ffwd's implementation is highly optimized.

However, as operations length increase, the advantages of ffwd diminish. For operation lengths of 500 cycles, neither `ffwd-s1` nor `ffwd-s4` are competitive with DPS. Servers in ffwd quickly saturate, resulting in flatlined performance. With an increasing operation duration (Figure 3 in §2), ffwd's throughput degrades steeply due to the delegation server serialization (for both 1 and 4 servers). In contrast, DPS performs better with a modest operation length (400 cycles), as its hierarchical design enables it to utilize all available cores. Furthermore, the performance decrease in DPS is very small, illustrating the effectiveness of overlapping computation in different localities.

Delegation Responsiveness. In DPS, the processing of delegated operations is carried out when waiting for response (*i.e.* threads need to issue data-structure operations using DPS). Thus, as computation outside of the data-structure increases, DPS risks poor responsiveness. Figure 6(b) shows the throughput of empty operations, as we increase the delay between consecutive two operation. In addition to basic DPS and ffwd, we also include the result of DPS with the asynchronous optimization from §4.4 (`DPS-a`). Although ffwd performs better than basic DPS for empty operations, it is not competitive with the asynchronous DPS. The results demonstrate that the asynchronous execution in DPS successfully hides the delays between invocations.

Working set and cache contention. This experiment performs a synthetic benchmark that allows us to alter working

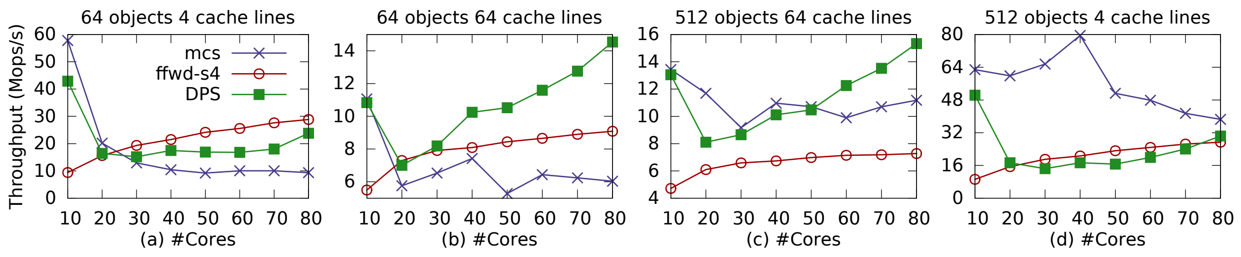


Figure 7: Micro-benchmarks: throughput of atomic read-write object with various workloads

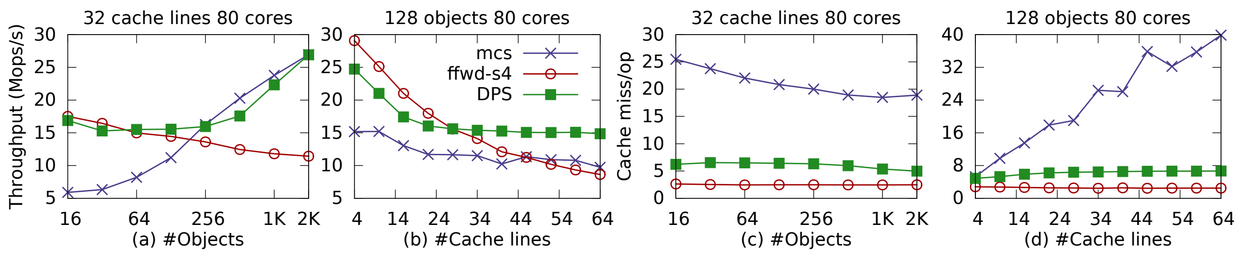


Figure 8: Micro-benchmarks: throughput and cache misses per operation of atomic read-write object

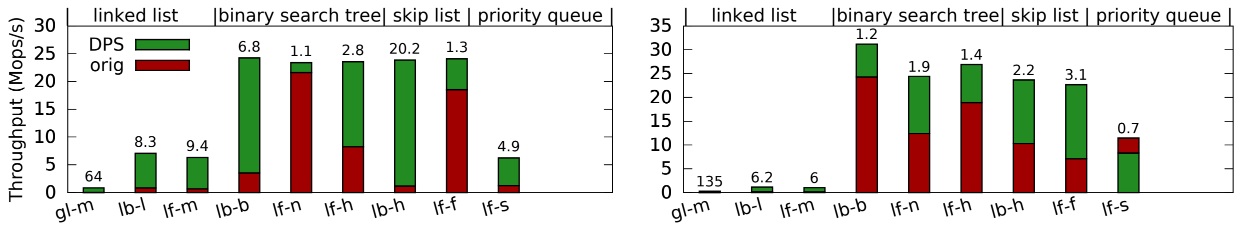
set (by increasing the number of objects), and the number of modified cache-lines (thus the amount of cache-coherency overhead). Each thread uniformly at random chooses an object and performs the data-structure operation. Ffwd deploys four servers and statically partitions the data-structure across servers. Each client delegates its operation to the appropriate server. We compare against locking, where each object is protected with a separate MCS lock [34]. The same locking implementation is used in DPS to synchronize multiple threads within a locality, and the index of the object is used as the namespace key. Figure 7 compares the throughput of these techniques under various setups.

Figure 7(a) has 64 objects, each with 4 cache-line modifications. With fewer cores, the contention is minor, as the number of objects are sufficiently larger than the number of cores. In such a case, the fine-grain locking can take the advantage of parallelism, and achieves the highest throughput. DPS also gets benefits from parallelism, but its interposition on data-structure processing adds a slight overhead over MCS. As we increase the core count, contention also increases, continuously decreasing MCS throughput. This contention also limits DPS's scalability, but its ability to localize cache traffic within a socket enables it to perform better than the locking method. In this case, ffwd steadily gains in throughput with increasing workload since operations are short.Though not the best option for these settings, DPS remains more consistently competitive across core counts than the other options.

When more capacity or coherency cache misses are produced, DPS presents a substantial performance boost. Figure 7(b) extends the object size to 64 cache lines, pressuring cache capacity.With only 64 locks, cache-coherency overheads prevent MCS from scaling. ffwd benefits from avoiding contention, but longer operations prevent it scaling at a rate of DPS. 512 objects are used in Figure 7(c). This decreases lock contention for both MCS and each locality in DPS, increasing throughput relative to ffwd. For higher core counts, the preservation of memory locality in DPS enables it to scale well versus MCS. However, when cache-coherency overheads shrink in Figure 7(d) (only 4 cache-lines are modified), MCS locks suffer little overhead. In Figure 7(d), operations are short, thus ffwd increases in throughput (tracking DPS with the exception of for low core counts).

To better understand the system factors that impact which approach is most scalable, we measure their throughput on 80 cores as we vary the number of objects (Figure 8(a)), and cache-lines (Figure 8(b)). In Figure 8(a), the object size is fixed to 32 cache lines. For a small number of objects, MCS contends a small number of locks while DPS and ffwd effectively avoid expensive contention. With an increasing number of objects, ffwd's performance degrades due to less effective use of L1 and L2 caches while DPS and MCS effectively use the decreased lock contention, and distributed caches to scale well. Figure 8(b) uses 128 objects, and varies the modified working set. Ffwd does well with short data-structure operations (few cache-line modifications), but degrades as they increase. MCS locks struggle to maintain sufficient parallelism with only 128 striped locks. Both DPS and MCS decrease in performance due to increased critical section length (even if it is restricted to a locality in DPS).

Figures 8(c) and (d) plot the LLC cache-miss rates for the same setups. These confirm that (1) locking does significantly increase the number of cache misses (vertical offset of MCS versus other techniques); (2) shared memory structures that modify multiple cache-lines entail unavoidable overheads (increasing MCS line in 8(b)); and (3) the ffwd implementation that batches replies is able to decrease the number of cache misses compared to DPS. The *design* of



(a) skewed 4K nodes 50% update (high contention) (b) uniform 32k/2M nodes 5% update (large working set)

Figure 9: Throughput improvements (the number on each bar) on existing concurrent data-structure implementations (80 cores). For the linked list, we compare a global MCS lock protected list (`gl-m`), lock-based lazy list [20] (`lb-l`) and the Michael lock-free list [36] (`lf-m`). For `bst`, we study a lock-based implementation from Bronson et al. [8] (`lb-b`) and two lock-free algorithms, from Natarajan et al. [38] (`lf-n`) and Howley et al. [25] (`lf-h`), respectively. The skip list implementations include the lock-based Herlihy list [23] (`lb-h`) and the optimized Fraser lock-free list [45] (`lf-f`). We also include a lock-free priority queue implementation from Shavit et al [44] (`lf-s`).

DPS	ffwd-s4	MCS (local)	MCS (interleave)
929.385	844.839	365.922	907.051

Table 2: Micro-benchmarks: throughput (ops/s) of atomic read-write object with 5GB working set.

DPS compensates for the additional overhead over ffwd by increasing available parallelism.

To further assess the use of DPS with large working sets, Table 2 reports the throughput on 80 cores with a 5GB working set, with 512 10MB-sized objects. MCS performs worst under the default `node local` NUMA memory allocation policy, as a single NUMA node is saturated. Using the `interleave` NUMA policy increases throughput by a factor of 2.5. Despite ffwd's NUMA awareness, it's throughput suffers due to the lack of sufficient parallelism. Though this workload is memory-bound in the data-structure, DPS best uses NUMA locality and effectively uses parallelism.

5.2 Data structures

This section presents an exhaustive evaluation of the state-of-the-art concurrent data-structure algorithms and parallelism techniques, and compares them with DPS. These experiments study four data-structures, a sorted singly linked list (`ll`), a binary search tree (`bst`), a skip list (`sl`) and a priority queue (`pq`). All these data-structures represent a set of nodes, each of them having a unique key and value. `ll`, `bst` and `sl` have three main operations, `lookup`, `insert` and `remove`. `pq` provides two extra operations, `findMin` and `removeMin`. All data-structure implementations and the benchmark framework are taken from ASCYLIB library [3]. Our benchmarks explore a broad range of parameter settings, including workload type (uniform or skewed), update percentage, data-structures size and core count. Each benchmark populates the data-structure with the given size, and a key range doubling the initial size. In each thread's iteration, the benchmark picks a key from the key range based on the specified distribution, then determines if the operation is an update operation according to the update percentage. Update operations include half insertions, half removals.

The DPS versions use one of the existing concurrent data-structures within each locality shown in Figure 9, which depicts the performance of existing version and the DPS result (overlaid). The `findMin` operation in `pq` is implemented using DPS range operations (§4.4). The system configurations shown here are those that existing implementations struggle most with: high update workload (Figure 9(a)), and large working sets (Figure 9(b)). In these setups, DPS delivers performance improvements for all data-structures, except for `pq` in Figure 9(b), by enabling better memory locality and avoiding cache-line contention bottlenecks. For instance, under high contention, DPS improves lock-based `bst` and `sl` up to 6 and 20 times, respectively (Figure 9(a)). With large working set, DPS improves lock-free algorithms of `bst` and `sl` by 1.4 and 3 times, respectively (Figure 9(b)). With a low update ratio (Figure 9(b)), the most visited node in `pq` is its head, thus, leading to few cache misses. While, DPS fails to improve `pq` in this case due to message passing overhead.

Two important observations: First, DPS enables some of the simpler and less scalable algorithms to approach the performance of their sophisticated counterparts. For example, with DPS the naive `gl-m` list is on par with the complicated Michael list. Second, DPS provides more predictable performance. For instance, `lf-n` and `lf-f` different across workloads (a) and (b). However, DPS confines remote memory accesses using message passing, and localizes cache contention to a single LLC, which provide more consistent performance across workloads.

To understand the details behind the bars in Figure 9, we separately evaluate the data-structures. We omit the details behind the priority queue in the interest of brevity. In addition to the data-structures listed above, we compare with the ffwd delegation system, OPTIK [18], a design pattern for optimistic concurrency and RLU [33].

Linked List. In this benchmark, DPS is integrated with the ParSec linked list, which uses ParSec quiescence for memory reclamation [51] and an MCS lock to serialize writers. Ffwd uses a single server (as in [42]) and is built on top of a lazy linked list. In ffwd, clients traverse the list locally, and

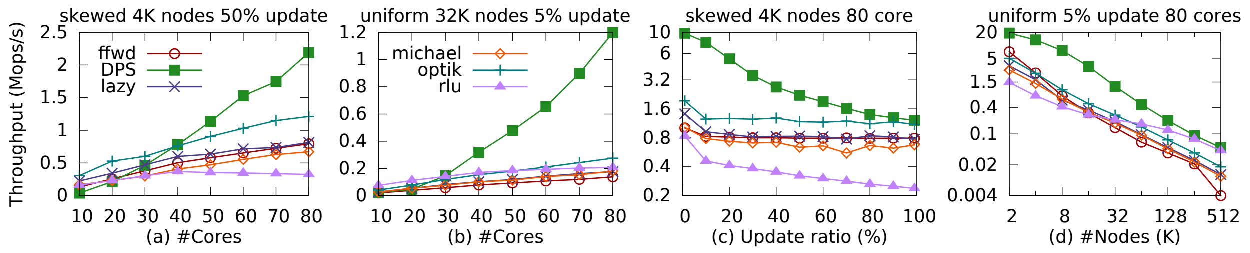


Figure 10: Throughput of a sorted singly linked list on different workloads.

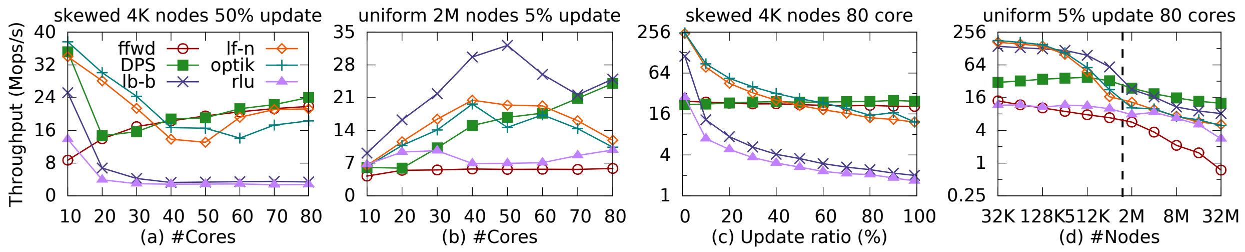


Figure 11: Throughput of the binary search tree on different workloads.

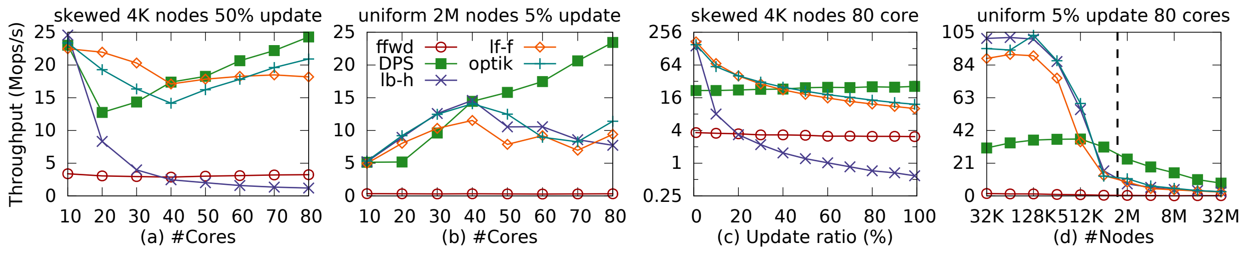


Figure 12: Throughput of skip lists for different workloads.

delegate node modifications to the server. Thus, ffwd relies on the underlying concurrent lazy algorithm to synchronize clients and server. OPTIK-based linked list uses fine-grained OPTIK locks with the node caching optimization.

Figure 10 shows the throughput of linked list on various workloads. Figure 10(a) and (b) show its scalability under high contention, and large working set cases, respectively. As the core count grows, the increased inter-socket cache contention limits the scalability of all locality-unaware techniques. Only DPS avoids most non-local memory access and cache-coherency traffic, and with 80 cores DPS has 4.3 times improvements over the next best implementation (OPTIK).

To better understand the impact of memory locality issues, Figures 10(c) and (d) show the performance as we vary the update ratio and the list length with 80 cores (note the log scales). The throughput of all methods drop with an increasing update ratio. DPS and RLU degrade quicker than others, because of the single writer lock used in ParSec linked list and the blocked quiescence detection in `rlu_synchronize`. The performance of the linked list decreases quickly (Figure 10(d)), due to its $O(n)$ complexity. An interesting observation is that ffwd drops from the second best to the worst due to increased operation latency. It is worth noting that the partitioning of the link-list's namespace across localities in DPS has another side-effect: each partition manages a shorter list. This is an additional factor in DPS performance.

Binary search tree. `bst` is an interesting case as efficient, specialized implementations exist. We use `lb-b` and `lf-n` as representatives of lock-based and lock-free algorithms, respectively, Ffwd uses four servers, and each server manages a sharding of the tree. In that case, all read and update operations are delegated to servers, that each use a barebones `bst` implementation. OPTIK reuses the BST-TK implementation from ASCY [14], which is also the internal data-structure used by DPS.

Figure 11 depicts the throughput of the various approaches. Under high contention (Figure 11(a)), there are three trends: (1) OPTIK and `lf-n` impose little overhead for few cores, and flatline when saturating physical cores, (2) DPS and ffwd scale well as they avoid remote memory accesses, and (3) `lb-b` and RLU require more complex updates (rotations for `lb-b` and quiescence for RLU), thus scale poorly. Of note, DPS does take advantage of local computation for 10 cores. The scalability of OPTIK and `lf-n` indicate that the localities for DPS might benefit from being larger in this case.

Figure 11(b) focuses on a large working set, with a read-heavy workload. `lb-b` has the highest throughput, due to an optimistic read-path (no stores), but also since it maintains a balanced tree (max depth 25, versus 48, 60 for DPS, OPTIK). ffwd has difficulty scaling, as larger the tree causes longer operations that saturate delegation servers, whereas DPS scales as it leverages more parallelism. Figure 11(c) and (d) vary the degree of cache coherency traffic and working set, and they reinforce the conclusions above. In Figure 11(d), the vertical line represents the size of aggregate LLC cache. DPS and ffwd maintain performance invariant on update ratios, though for read-only workloads their delegation infrastructure adds significant overhead relative to the other approaches. For large working sets, when the data-structure size exceeds the LLC capacity, DPS makes use of distributed caches and local memory accesses while all shared memory implementations suffer remote memory accesses, and ffwd suffers from longer operation lengths due to cache thrashing.

In summary: `lf-n` and OPTIK perform especially well for update-intensive workloads. `lb-b` maintains a balanced tree, and has very good performance with few updates. While RLU's performance compared to many of the customized implementations is lacking, it offers a much simpler interface and broader applicability. ffwd generally does well when delegated operations are short, but is quickly overloaded with longer operations. DPS is particularly effective with a large core count as it both harnesses parallelism, well-utilizes LLCs, and avoids remote memory traffic. However, at a small scale, interposition of the runtime on local operations can cause significant overhead for small update ratios.

Skip list. Figure 12 depicts all `sl` results. We omit RLU as we could not find a public `sl` implementation. Overall, `sl` behaves very similar to `bst` but for a few trends. First, ffwd performs consistently worse as its provided implementation only supports one server. Concurrent `sl` implementations are less efficient than their `bst` counterpart in general. Thus, the relative performance of DPS tends to be more favorable. Unlike `lb-b` `bst`, the `sl` read-heavy, large working set workloads aren't as scalable. Thus, even for a read-heavy workload with a large working set, DPS achieves at least a 3.2 times higher throughput than the others.

5.3 Application study: `memcached`

To evaluate DPS in real-world applications, we use `memcached` (https://memcached.org/), a popular in-memory key-value cache infrastructure in data centers [39]. At first glance, `memcached` is a rather counter-intuitive use case for DPS. The underlying data-structure of `memcached` is a hash table protected by per-bucket locks. It isn't clear that DPS could improve `memcached`. First, fine-grained locking enables large hash tables to effectively use parallelism, marginalizing lock contention. Second, `memcached` is designed for a read-most workload. Regardless, we find that `memcached` still suffers from significant remote memory access overhead due to its large memory footprint. Third, `memcached` contains complicated connections between its hash table, LRU list, and the backend memory allocator. Only optimizing its hash table is not sufficient to scale the whole system. As reported in §4.5, this did not prohibit the porting effort.

We compare five `memcached` implementations. (1) stock `memcached` (version 1.5.4). (2) ffwd `memcached` (using version 1.4.6), where all `get` and `set` operations are delegated to a single server without any synchronization (as in [42]). (3) ParSec `memcached` (using version 1.4.22), a highly customized implementation, which replaces slab allocator, LRU list and hash table in `memcached` with its own implementation [51]. (4) DPS `memcached`, which partitions not only the hash table, but also all associated data-structures. It also asynchronously delegates `set` requests to remote partitions, while `get` requests remain synchronous delegations. and (5) DPS ParSec, where DPS is applied on top of ParSec `memcached`. To leverage the optimized read operation in ParSec, DPS ParSec always locally execute `get` requests (§4.4), while still asynchronously delegates `set` requests. Similar to [51], we use YCSB [13] to generate testing traces, following a Zipfian distribution. Each trace has 10 million requests (16 Byte key), and are partitioned across all testing threads. `memcached` is pre-populated with 1 million items. To focus on data-structure scalability, we invoke `memcached` directly, instead of generating requests in remote (networked) clients.

Throughput Discussion. Figure 13 varies core count, set ratio, and value size. Figure 13(a) shows the performance under a typical `memcached` workload – low `set` ratio with small value size. The performance of stock `memcached` is good, benefiting from the inherent parallelism in the hash table, and the read-most workload. Regardless, the stock `memcached` does cause significant remote memory accesses. In contrast, DPS `memcached` largely avoids such remote memory accesses, resulting in a throughput improvement of over 200% over the stock `memcached`. Thanks to the highly customized implementation which avoids stores on the `get` path, ParSec `memcached` greatly outperforms the others. However, DPS ParSec still improves on it by increasing the locality of modifications. DPS ParSec combines the advantages of both DPS and ParSec, namely locally executed `get`s, and asynchronously delegated `set`s.

In contrast, Figure 13(b) investigates large value sizes combined with a high update ratio workload. Though this is not a common workload, it presents a chance to understand each technique more comprehensively. Under such severe workload, even ParSec `memcached` fails to scale beyond 40 cores. Of significant note is that DPS produces the same throughput as ParSec on 80 cores, without significantly reimplementing `memcached`. This is surprising as a relatively straightforward data-structure, when accessed using DPS is competitive with a highly specialized and complicated one. Figure 13(c) and

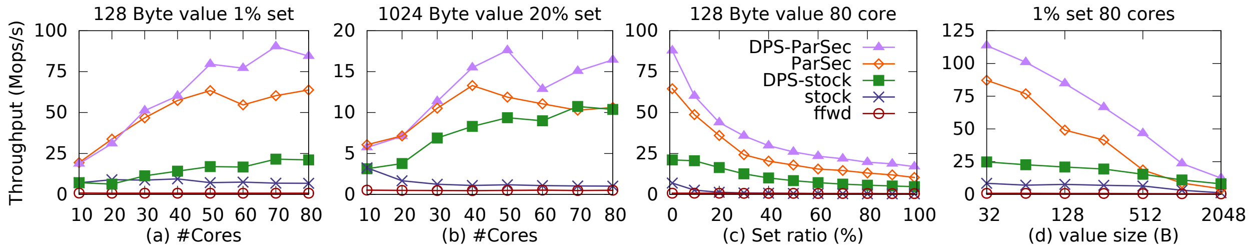


Figure 13: Memcached throughput with different workload

(d) further investigate memory locality effects. With increasing update ratio, throughput consistently and predictably decreases. Interestingly, as the data-structure size increases (in (d)), the local modifications of DPS have better throughput than ParSec `memcached`. As the value size grows in (d), locality-unaware methods drop in throughput rapidly. DPS ParSec behaves similarly, as its local `get` execution also access remote memory. DPS `memcached` is less sensitive to value size, and eventually achieves better throughput than ParSec `memcached`.

In all system setups, ffwd `memcached` doesn't provide high throughput. It does have a higher throughput with a high set ratio (> 50%) than stock `memcached` (*e.g.* 63% higher throughput with 99% set). As reported in [42], when compared to an old version of `memcached` that uses a global lock, it consistently achieves higher throughput (3x higher). However, the throughput of ffwd `memcached` is limited by the single ffwd server. In contrast, DPS approaches have much higher throughput as they utilize full system parallelism, but require application modifications (detailed in § 4.5). Though we use an older version of `memcached` (from [42]), the global lock it employs is not prohibitive with the serialized server.

Latency Discussion. `memcached` tail latency is another important metric. Due to space limitation, we omit its result graph. With fewer than 30 cores, both ffwd and DPS `memcached` have larger latency due to their delegation overhead. However, with more cores, the latency of stock `memcached` increases quickly because of its lock acquisition overhead and poor cache locality. DPS has significantly smaller latency than ffwd due to request's serialization in ffwd. ParSec and DPS ParSec always have the lowest latency, thanks to the optimized read operation in ParSec. On 80 cores, DPS based implementations have lower latency by a factor of 23 and 1.6 than the stock and ParSec `memcached`, respectively.

6 RELATED WORK

Concurrent data-structures. There is a large body of research designing efficient concurrent data-structures [1, 2, 8, 20, 23, 25, 36–38, 45, 48]. ASCY [14] and Synchrobench [17] provide surveys of concurrent data-structures. Due to stores to shared synchronization variables, those implementations usually hit their scalability limit when inter-socket cache coherency are involved. DPS uses those concurrent data-structures within its localities to avoid their scalability limits. CPHash [35] is a partitioned hash table, that uses message passing to transfer operations among partitions. [19, 22, 41] use multiple data-structure instances to distribute concurrent access. Though DPS exploits similar ideas, DPS is more general as is not confined to a specific data-structure.

Delegation and combining. Delegation systems [9, 27, 40, 42] are designed for improving memory locality and eliminating synchronization cost. Combining is a specific delegation implementation [16, 21, 31, 47]. Most of them only allow one server to run each time and do not utilize efficient concurrent data-structure implementations inside their server, whereas DPS employs multiple partitions to run in parallel. Furthermore, DPS focuses on optimizing memory locality, providing no means of synchronization. The separation between memory locality optimization and data-structure synchronization gives significant performance advantages over previous delegation approaches.

NUMA-aware techniques. Cohort locks [15] presents a hierarchical approach to construct NUMA-aware locks. Calciu et al. designs a NUMA-friendly stack [10] by eliminating reciprocal operations (`push` and `pop`). Shoal [26] automatically replicates, distributes or partitions arrays across NUMA domains. NR [11] provides a black-box approach to transfer sequential data-structures into NUMA-aware concurrent implementations, which replicates data-structures on each NUMA node and uses a shared log to synchronize between replicas. Lepers at el. [29] present a thread and memory placement on asymmetry NUMA system, and FlexSC [46] uses asynchronous message passing to enhance system call locality. DPS shares the broad ideas of NUMA-awareness with those techniques, but focuses on a hierarchical combination of message-passing and shared memory structures.

7 CONCLUSIONS

We introduce the *Distributed, Delegated Parallel Sections* (DPS) runtime that uses a peer-computation system to distribute data-structure operations to specific *localities*. It partitions the data-structure namespace to maintain NUMA memory locality, and limits parallelism within a locality, enabling scalable use of concurrent data-structures. DPS uses message passing to move operations to their data's locality, and efficient shared memory structures within. We've demonstrated the utility and scalability of the system across microbenchmarks, data-structures, and via large gains over stock `memcached` with more than a factor of 3.1 improvements in throughput, and 23x decreases in tail latency.

REFERENCES

[1] Dan Alistarh, Justin Kopinsky, Jerry Li, and Nir Shavit. 2015. The SprayList: A scalable relaxed priority queue. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'15), San Francisco, CA, USA.*

[2] Maya Arbel and Hagit Attiya. 2014. Concurrent Updates with RCU: Search Tree As an Example. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing (PODC '14).*

[3] ASCYLIB [n. d.]. ASCYLIB (with OPTIK) concurrent data-structure library: https://github.com/LPD-EPFL/ASCYLIB.

[4] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2013. Highly Available Transactions: Virtues and Limitations. *Proc. VLDB Endowement* 7, 3 (Nov. 2013).

[5] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2014. Coordination Avoidance in Database Systems. *Proceedings of the VLDB Endowment* 8, 3 (Nov. 2014).

[6] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A Critique of ANSI SQL Isolation Levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data (SIGMOD '95).*

[7] E. Brewer. 2010. Towards robust distributed systems. Keynote at PODC'10.

[8] Nathan G Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. 2010. A practical concurrent binary search tree. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'10), Bangalore, India.*

[9] Irina Calciu, Dave Dice, Tim Harris, Maurice Herlihy, Alex Kogan, Virendra Marathe, and Mark Moir. 2013. Message Passing or Shared Memory: Evaluating the Delegation Abstraction for Multicores *(OPODIS 2013).*

[10] Irina Calciu, Justin Emile Gottschlich, and Maurice Herlihy. 2013. Using Elimination and Delegation to Implement a Scalable NUMA-Friendly Stack.. In *Presented as part of the 5th USENIX Workshop on Hot Topics in Parallelism, San Jose, CA, USA.*

[11] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K Aguilera. 2017. Black-box concurrent data structures for NUMA architectures. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17), Xi'an, China.*

[12] Sang Kyun Cha, Sangyong Hwang, Kihong Kim, and Keunjoo Kwon. 2001. Cache-conscious concurrency control of main-memory indexes on shared-memory multiprocessor systems. In *VLDB*, Vol. 1. 181–190.

[13] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *SoCC.*

[14] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 2015. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15).*

[15] David Dice, Virendra J Marathe, and Nir Shavit. 2012. Lock cohorting: a general technique for designing NUMA locks. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'12), New Orleans, Louisiana, USA.*

[16] Panagiota Fatourou and Nikolaos D Kallimanis. 2012. Revisiting the combining synchronization technique. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '12).*

[17] Vincent Gramoli. 2015. More than you ever wanted to know about synchronization: synchrobench, measuring the impact of the synchronization on concurrent algorithms. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'15), San Francisco, CA, USA.*

[18] Rachid Guerraoui and Vasileios Trigonakis. 2016. Optimistic concurrency with OPTIK. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'16), Barcelona, Spain.*

[19] Andreas Haas, Michael Lippautz, Thomas A Henzinger, Hannes Payer, Ana Sokolova, Christoph M Kirsch, and Ali Sezgin. 2013. Distributed queues in shared memory: multicore performance and scalability through quantitative relaxation. In *Proceedings of the ACM International Conference on Computing Frontiers (CF'13), Ischia, Italy.*

[20] Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N Scherer, and Nir Shavit. 2005. A lazy concurrent list-based set algorithm. In *Proceedings of the 9th International Conference on Principles of Distributed Systems (OPODIS'05), Pisa, Italy.*

[21] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. 2010. Flat Combining and the Synchronization-parallelism Tradeoff. In *Proceedings of the 22Nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '10).* ACM, New York, NY, USA, 355–364. https://doi.org/10.1145/1810479.1810540

[22] Thomas A Henzinger, Christoph M Kirsch, Hannes Payer, Ali Sezgin, and Ana Sokolova. 2013. Quantitative relaxation of concurrent data structures. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'13), Rome, Italy.*

[23] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. 2007. A simple optimistic skiplist algorithm. In *Proceedings of the 14th International Conference on Structural Information and Communication Complexity (SIROCCO'07), Castiglioncello, LI, Italy.*

[24] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (1990), 463–492.

[25] Shane V Howley and Jeremy Jones. 2012. A non-blocking internal binary search tree. In *Proceedings of the 24th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'12), Pittsburgh, Pennsylvania, USA.*

[26] Stefan Kaestle, Reto Achermann, Timothy Roscoe, and Tim Harris. 2015. Shoal: smart allocation and replication of memory for parallel programs. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference (ATC'15), Santa Clara, CA, USA.*

[27] David Klaftenegger, Konstantinos Sagonas, and Kjell Winblad. 2014. Brief announcement: Queue delegation locking. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA) (SPAA '14).*

[28] Philip L Lehman et al. 1981. Efficient locking for concurrent operations on B-trees. *ACM Transactions on Database Systems (TODS)* 6, 4 (1981), 650–670.

[29] Baptiste Lepers, Vivien Quéma, and Alexandra Fedorova. 2015. Thread and Memory Placement on NUMA Systems: Asymmetry Matters.. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference (ATC'15), Santa Clara, CA, USA.*

[30] Justin J Levandoski, David B Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for new hardware platforms. In *29th IEEE International Conference on Data Engineering (ICDE'13).*

[31] Jean-Pierre Lozi, Florian David, Gaël Thomas, Julia Lawall, and Gilles Muller. 2012. Remote core locking: migrating critical-section execution to improve the performance of multithreaded applications. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12).*

[32] Yandong Mao, Eddie Kohler, and Robert Morris. 2012. Cache Craftiness for Fast Multicore Key-Value Storage. In *Proceedings of the ACM EuroSys Conference (EuroSys 2012).* Bern, Switzerland.

[33] Alexander Matveev, Nir Shavit, Pascal Felber, and Patrick Marlier. 2015. Read-log-update: A Lightweight Synchronization Mechanism for Concurrent Programming. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*.

[34] John M. Mellor-Crummey and Michael L. Scott. 1991. Algorithms for Scalable Synchronization on Shared-memory Multiprocessors. *ACM Trans. Comput. Syst.* (1991).

[35] Zviad Metreveli, Nickolai Zeldovich, and M Frans Kaashoek. 2012. Cphash: A cache-partitioned hash table. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'12), New Orleans, Louisiana, USA*.

[36] Maged M Michael. 2002. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'02), Winnipeg, Manitoba, Canada, August 11-13*.

[37] Adam Morrison and Yehuda Afek. 2013. Fast Concurrent Queues for x86 Processors. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*.

[38] Aravind Natarajan and Neeraj Mittal. 2014. Fast concurrent lock-free binary search trees. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'14), Orlando, Florida, USA*.

[39] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. 2013. Scaling Memcache at Facebook. In *NSDI*.

[40] Darko Petrović, Thomas Ropars, and André Schiper. 2014. Leveraging Hardware Message Passing for Efficient Thread Synchronization. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '14)*.

[41] Hamza Rihani, Peter Sanders, and Roman Dementiev. 2015. Brief announcement: Multiqueues: Simple relaxed concurrent priority queues. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'15), Portland, Oregon, USA*.

[42] Sepideh Roghanchi, Jakob Eriksson, and Nilanjana Basu. 2017. ffwd: delegation is (much) faster than you think. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17), Shanghai, China*.

[43] Jason Sewall, Jatin Chhugani, Changkyu Kim, Nadathur Satish, and Pradeep Dubey. 2011. PALM: Parallel architecture-friendly latch-free modifications to B+ trees on many-core processors. *Proceedings of the VLDB Endowment* 4, 11 (2011), 795–806.

[44] N. Shavit and I. Lotan. 2000. Skiplist-based concurrent priority queues. In *Proceedings 14th International Parallel and Distributed Processing Symposium (IPDPS'00)*.

[45] Nir N Shavit, Yosef Lev, and Maurice P Herlihy. [n. d.]. Concurrent lock-free skiplist with wait-free contains operator. May 3, 2011. US Patent 7,937,378.

[46] Livio Soares and Michael Stumm. 2010. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. In *OSDI*.

[47] M Aater Suleman, Onur Mutlu, Moinuddin K Qureshi, and Yale N Patt. 2009. Accelerating critical section execution with asymmetric multi-core architectures. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*.

[48] Josh Triplett, Paul E. McKenney, and Jonathan Walpole. 2011. Resizable, Scalable, Concurrent Hash Tables via Relativistic Programming. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*.

[49] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP'13)*.

[50] Paolo Viotti and Marko Vukolić. 2016. Consistency in Non-Transactional Distributed Storage Systems. *Comput. Surveys* 49, 1, Article 19 (June 2016).

[51] Qi Wang, Tim Stamler, and Gabriel Parmer. 2016. Parallel Sections: Scaling System-Level Data-Structures. In *Proceedings of the ACM EuroSys Conference*.

[52] Chenggang Wu, Jose M. Faleiro, Yihan Lin, and Joseph M. Hellerstein. 2018. Anna: A KVS for Any Scale. In *34th IEEE International Conference on Data Engineering, (ICDE)*.