Flat Combining Synchronized Global Data Structures

Brandon Holt[1], Jacob Nelson[1], Brandon Myers[1], Preston Briggs[1], Luis Ceze[1], Simon Kahan[1,2] and Mark Oskin[1]

[1] University of Washington
[2] Pacific Northwest National Laboratory
{bholt,nelson,bdmyers,preston,luisceze,skahan,oskin}@cs.washington.edu

Abstract

The implementation of scalable synchronized data structures is notoriously difficult. Recent work in shared-memory multicores introduced a new synchronization paradigm called *flat combining* that allows many concurrent accessors to cooperate efficiently to reduce contention on shared locks. In this work we introduce this paradigm to a domain where reducing communication is paramount: distributed memory systems. We implement a flat combining framework for Grappa, a latency-tolerant PGAS runtime, and show how it can be used to implement synchronized global data structures. Even using simple locking schemes, we find that these flat-combining data structures scale out to 64 nodes with 2x-100x improvement in throughput. We also demonstrate that this translates to application performance via two simple graph analysis kernels. The higher communication cost and structured concurrency of Grappa lead to a new form of distributed flat combining that drastically reduces the amount of communication necessary for maintaining global sequential consistency.

1 Introduction

The goal of partitioned global address space (PGAS) [10] languages and runtimes is to provide the illusion of a single shared memory to a program actually executing on a distributed memory machine such as a cluster. This allows programmers to write their algorithms without needing to explicitly manage communication. However, it does not alleviate the need for reasoning about consistency among concurrent threads. Luckily, the PGAS community can leverage a large body of work solving these issues in shared memory and explore how differing costs lead to different design choices.

It is commonly accepted that the easiest consistency model to reason about is *sequential consistency* (SC), which enforces that all accesses are committed in program order and appear to happen in some global serializable order. To preserve SC, operations on shared data structures should be *linearizable* [14]; that is, appear to happen atomically in some global total order. In both physically shared memory and PGAS, maintaining linearizability presents performance challenges. The simplest way is to have a single global lock to enforce atomicity and linearizability through simple mutual exclusion. Literally serializing accesses in this way is typically considered prohibitively expensive, even in physically shared memory. However, even in fine-grained lock-free synchronization schemes, as the number of concurrent accessors increases, there is more contention, resulting in more failed synchronization operations. With the massive amount of parallelism in a cluster of multiprocessors and with the increased cost of remote synchronization, the problem is magnified.

A new synchronization technique called *flat combining* [12] coerces threads to *cooperate* rather than *contend*. Threads delegate their work to a single thread, giving it the opportunity to combine multiple requests in data-structure specific ways and perform them free from contention. This allows even a data structure with a single global lock to scale better than complicated concurrent data structures using fine-grained locking or lock-free mechanisms.

The goal of this work is to apply the flat-combining paradigm to a PGAS runtime to reduce the cost of maintaining sequentially consistent data structures. We leverage Grappa, a PGAS runtime library optimized for fine-grained random access, which provides the ability to tolerate long latencies

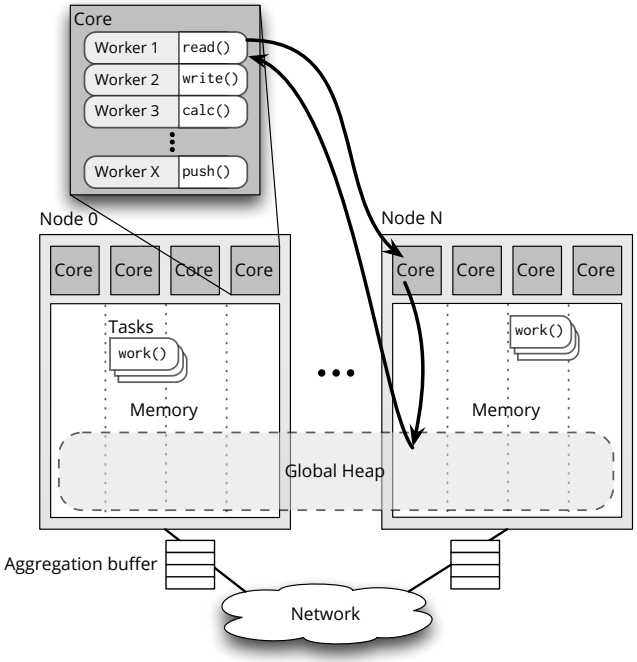


Figure 1: *Grappa System Overview:* Thousands of workers are multiplexed onto each core, context switching to tolerate the additional latency of aggregating remote operations. Each core has exclusive access to a slice of the global heap which is partitioned across all nodes, as well as core-private data and worker stacks. Tasks can be spawned and executed anywhere, but are bound to a worker in order to be executed.

by efficiently switching between many lightweight threads as sketched out in prior work [19]. We leverage Grappa's latency tolerance mechanisms to allow many fine-grained synchronized operations to be combined to achieve higher, scalable throughput while maintaining sequential consistency. In addition, we show how a generic flat-combining framework can be used to implement multiple global data structures.

The next section will describe in more detail the Grappa runtime system that is used to implement flat combining for distributed memory machines. We then explain the flat-combining paradigm in more depth and describe how it maps to a PGAS model. Next, we explain how several data structures are implemented in our framework and show how they perform on simple throughput workloads as well as in two graph analysis kernels.

2 Grappa

Irregular applications are characterized by having unpredictable data-dependent access patterns and poor spatial and temporal locality. Applications in this class, including data mining, graph analytics, and various learning algorithms, are becoming increasingly prevalent in high-performance computing. These programs typically perform many fine-grained accesses to disparate sources of data, which is a problem even at multicore scales, but is further exacerbated on distributed memory machines. It is often the case

that naively porting an application to a PGAS system results in excessive communication and poor access patterns [8], but this class of applications defies typical optimization techniques such as data partitioning, shadow objects, and bulk-synchronous communication transformations. Luckily, applications in this class have another thing in common: abundant amounts of data access parallelism. This parallelism can be exploited in a number of different ways to improve overall throughput.

Grappa is a global-view PGAS runtime for commodity clusters which has been designed from the ground up to achieve high performance on irregular applications. The key is latency tolerance—long-latency operations such as reads of remote memory can be tolerated by switching to another concurrent thread of execution. Given abundant concurrency, there are opportunities to increase throughput by sacrificing latency. In particular, throughput of random accesses to remote memory can be improved by delaying communication requests and aggregating them into larger packets.

Highly tuned implementations of irregular applications in shared-memory, PGAS, and message-passing paradigms, typically end up implementing similar constructs. Grappa includes these as part of its core infrastructure and simply asks the programmer to express concurrency which it can leverage to provide performance.

Grappa's programming interface, implemented as a C++11 library, provides high-level operations to access and synchronize through global shared memory, and task and parallel loop constructs for expressing concurrency. In addition, the Grappa "standard library" includes functions to manipulate a global heap, stock remote synchronization operations such as *compare-and-swap*, and several synchronized global data structures. These features make it suitable for implementing some next-generation PGAS languages like Chapel [6] and X10 [7]. The following sections will explain the execution model of Grappa and the current C++ programming interface.

2.1 Tasks and Workers

Grappa uses a task-parallel programming model to make it easy to express concurrency. A *task* is simply a function object with some state and a function to execute. Tasks may block on remote accesses or synchronization operations. The Grappa runtime has a lightweight threading system that uses prefetching to scale up to thousands of threads on a single core with minimal increase in context-switch time. In the runtime, *worker* threads pull these programmer-specified tasks from a queue and execute them to completion. When a task blocks, the worker thread executing it is suspended and consumes no computational resources until woken again by some event.

2.2 Aggregated Communication

The most basic unit of communication in Grappa is an *active message* [24]. To make efficient use of the networks in high-performance systems, which typically achieve maximum bandwidth only for messages on the order of 64 KB, all communication in Grappa is sent via an aggregation layer that automatically buffers messages to the same destination.

2.3 Global Memory

In the PGAS style, Grappa provides a global address space partitioned across the physical memories of the nodes in a cluster. Each core owns a portion of memory, which is divided among execution stacks for the core's workers, a core-local heap, and a slice of the global heap.

All of these can be addressed by any core in the system using a `GlobalAddress`, which encodes both the owning core and the address on that core. Additionally, addresses into the global heap are partitioned in a block-cyclic fashion, so that a large allocation is automatically distributed among many nodes. For irregular applications, this helps avoid hot spots and is typically sufficient for random access.

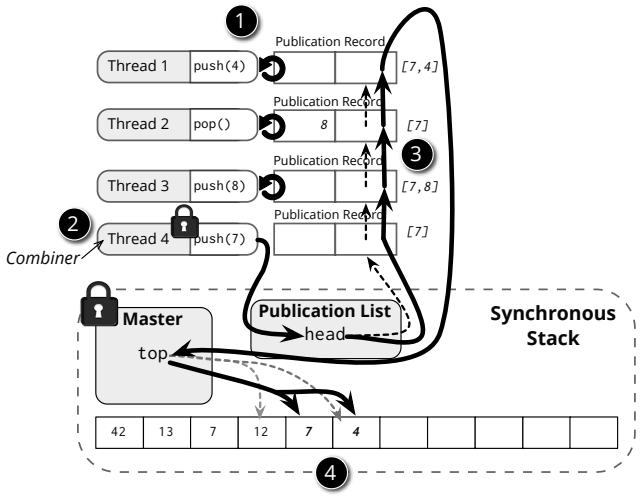


Figure 2: *Flat combining in shared memory.* To access the shared stack, each thread adds its request to the publication list (1). Thread 4 acquires the stack's lock and becomes the combiner (2), while the remaining threads spin waiting for their request to be satisfied. The combiner walks the publication list from the head, matching up Thread 3's push with Thread 2's pop on its own private stack (3). The two remaining pushes are added to the top of the shared stack (4). Finally, the top is updated, and Thread 4 releases the lock and continues its normal execution.

Grappa enforces strict isolation—all accesses must be done by the core that owns it via a message, even between processes on the same physical memory. At the programming level, however, this is hidden behind higher-level remote operations, which in Grappa are called *delegates*. Figure 1 shows an example of a delegate `read` which blocks the calling task and sends a message to the owner, who sends a reply with the data and wakes the caller.

3 Flat Combining

At the most basic level, the concept of flat combining is about enabling cooperation among threads rather than contention. The benefits can be broken down into three components: improved locality, reduced synchronization, and data-structure-specific optimization. We will explore how this works in a traditional shared-memory system, and then describe how the same concepts can be applied to distributed memory.

3.1 Physically shared memory

Simply by delegating work to another core, locality is improved and synchronization is reduced. Consider the shared synchronous stack shown in Figure 2, with pre-allocated storage and a `top` pointer protected by a lock. Without flat combining, whenever a thread attempts to push something on the stack, it must acquire the lock, put its value into the storage array, bump the top, and then release the lock. When many threads contend for the lock, all but one will fail and have to retry. Each attempt forces an expensive memory fence and consumes bandwidth, and as the number of threads increases, the fraction of successes plummets. Under flat combining, instead, threads add requests to a *publication list*. They each try to

acquire the lock, and the one that succeeds becomes the combiner. Instead of retrying, the rest spin on their request waiting for it to be filled. The combiner walks the publication list, performs all of the requests, and when done, releases the lock. This allows the one thread to keep the data structure in cache, reducing thrashing between threads on different cores. It also greatly reduces contention on the lock, but introduces a new point of synchronization—adding to the publication list. However, if a thread performs multiple operations, it can leave its publication record in the list and amortize the synchronization cost. This publication list mechanism can be re-used in other data structures, saving each from needing to implement its own clever synchronization.

The above example of delegation is compelling in itself. However, the crux of prior work is that data structure-specific optimization can be done to perform the combined operations more efficiently. As the combiner walks the publication list, it merges each non-empty publication record into a combined operation. In the case of the stack example shown in Figure 2, as it walks the list, Thread 4 keeps track of the operations on its own temporary stack. When it encounters Thread 2's pop, it recognizes that it can satisfy that pop immediately with the push it just processed from Thread 3, so it fills both of their records and allows them to proceed. After traversing the rest of the publication list, the thread applies the combined operation to the actual data structure, in this case, the two unmatched pushes are added to the top of the stack. In the case of the stack, combining came in the form of matched pushes and pops, but many data structures have other ways in which operations can be matched.

3.2 Grappa

In a PGAS setting, and in Grappa in particular, the cost of global synchronization and the amount of concurrency is even greater than in shared memory. With thousands of workers per core, in a reasonably sized cluster there are easily millions of workers. This presents an opportunity for flat combining to pay off greatly, but also poses new challenges.

To illustrate how flat combining can be applied to Grappa, we must first describe what a global data structure looks like. Figure 3 shows a simple PGAS translation of the shared-memory stack from earlier. A storage array is allocated in the global heap, so its elements are striped across all the cores in the system. One core is designated the *master* to enforce global synchronization, and holds the elements of the data structure that all concurrent accessors must agree on, in this case, the `top` pointer.

All tasks accessing the stack hold a `GlobalAddress` to the master object, and invoke custom *delegate methods* that, like the `read` delegate described earlier, block the task until complete. Example code to do a `push` is shown in Figure 5. The task must send a message to the master to acquire the lock. If successful, it follows the `top` pointer, writes its new value to the end of the stack, returns to bump the `top` pointer and release the lock, and finally sends a message back to wake the calling worker. All others block at the first message until the lock is released. Grappa's user-level threading allows requests to block without consuming compute resources. However, all workers on each core perform this synchronization and serialize on a single core, causing that core to become the bottleneck.

Centralized Combining. A first thought might be to directly apply the idea of flat combining to the serialization at the *master*. The worker that acquires the lock can walk through the requests of all the other workers waiting to acquire the lock and combine them. In the case of the Stack, this would mean matching pushes and pops, applying the remainder, and sending messages back to all remote workers with results, before starting another round of combining. This approach reduces traffic to the data structure storage, but a single core must still process every request, so it cannot scale if every other core is generating requests at the same rate.

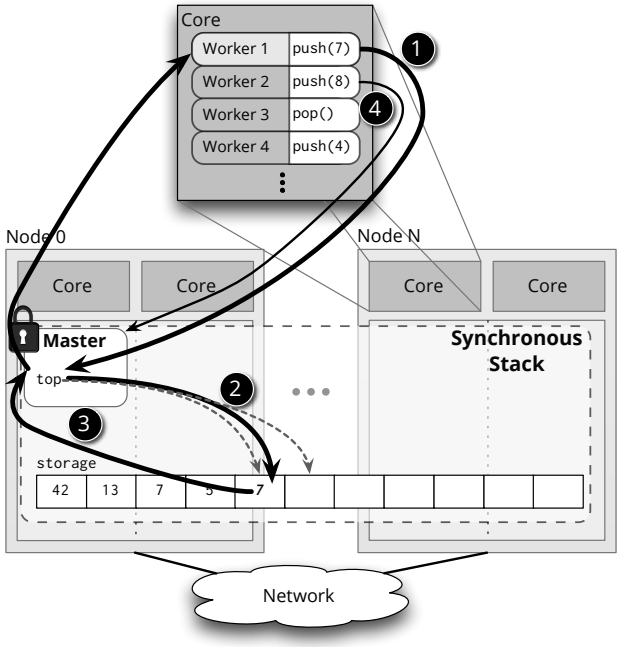


Figure 3: *Stack without combining.* To do its push, Worker 1 sends a message to synchronize with the master on Core 0 (1), which sends another message to write the value to the top of the stack (2), bumps the synchronized `top` pointer (3), and finally continues. Worker 2, and workers on other cores, must block at the master and wait for Worker 1 to complete its push before doing their operations (4).

Distributed Combining. Instead of all workers sending independent synchronization messages and putting the burden all on one core, each core can instead do its own combining first then synchronize with the master in bulk. Distributing the combining to each core allows the majority of the work to be performed in parallel and without communication.

In distributed flat-combining, each core builds its own publication list to track all the operations waiting to be committed to the global data structure. In Grappa, for each global data structure, a local *proxy* object is allocated from the core-local heap to intercept requests. Conceptually, workers add their requests to a local publication list in the proxy, one is chosen to do the combining, and the rest block until their request is satisfied. However, because Grappa's scheduler is non-preemptive, each worker has atomicity "for free" until it performs a blocking operation (such as communication). This means that an explicit publication list with clever synchronization is unnecessary. Instead, workers merge their operations directly into the local proxy, and block, except in restricted cases where they are able to satisfy their requirements immediately without violating ordering rules for consistency (discussed in the next section). The proxy structure is chosen to be able to compactly aggregate operations and efficiently perform matching in cases where it is allowed. Figure 4 shows how pushes and pops are matched on the proxy's local stack.

After all local combining has been done, one requesting worker on the core is chosen to commit the combined operation globally. In Figure 4, Worker 4 becomes the combiner and performs much the same synchronization as in the uncombined case, but is able to push multiple elements with one synchronization. The global lock must still be acquired, so concurrent combined requests (from different

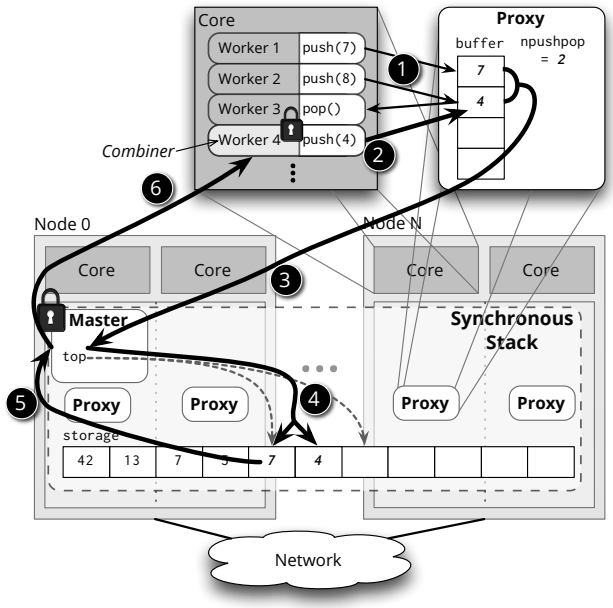


Figure 4: *Stack with distributed combining.* Worker 3's pop matches with Worker 2's push, requiring no global communication (1). After combining locally, Worker 1 and 4's pushes remain, so Worker 4 becomes the core's combiner (2), sends a message to synchronize with the master (3), adds both new values to global stack (4), bumps the top pointer and releases the lock on master (5), and finally wakes Worker 1 (6).

cores) must block and serialize on the master, but the granularity of global synchronization is coarser, reducing actual serialization.

Centralized combining could be applied on top of distributed combining, combining the combined operations at the master node. However, in our experiments, performing the additional combining on the master did not significantly affect performance, so it is left out of our evaluation for space.

Memory Consistency Model. In the context of Grappa, sequential consistency guarantees that within a task, operations will be in task order and that all tasks will observe the same global order. The Grappa memory model is essentially the same as the C++ memory model [5, 4, 15], guaranteeing sequential consistency for data-race free programs. To be conservative, delegate operations block the calling worker until they have become globally visible, ensuring they can be used for synchronization. As such, delegate operations within a task are guaranteed to be globally visible in program order and all tasks observe the same global order. Operations on synchronized data structures must provide the same guarantees. Because it is not immediately obvious that this distributed version of flat combining preserves sequential consistency, we now argue why it does.

To behave in accordance with sequential consistency, operations on a particular data structure must obey a consistent global order. One way to provide this is to guarantee *linearizability* [14] of operations, which requires that the operation appear to take effect globally at some instantaneous point during invocation of the API call. This ensures that a consistent total order can be imposed on operations on a single data structure. For operations to be globally linearizable, first of all the execution of local combined

operations must be serializable; this is unchanged from shared-memory flat-combining, and is trivially true due to the atomicity enabled by cooperative multithreading. Second, combined operations must be committed atomically in some globally serializable order. When committing, combined operations are serialized at the particular core that owns the synchronization (the *master* core for the Stack). Whenever a global commit starts, a fresh combiner must be created for subsequent operations to use. If they were to use the old combiner's state to satisfy requests locally, it would violate global ordering because the local object would not reflect other cores' updates. For this serial "concatenation" of serialized batches to be valid, the order observed by workers during local combining must be the same as what can be observed globally as operations are committed.

In the case of a stack or queue, this guarantee comes from applying a batch of push or pop operations atomically in a contiguous block on the global data structure. Matching pushes and pops locally for the Stack is one exception to the rule that operations must block until globally committed. Because a pop "destroys" the corresponding push, these operations together are independent of all others and can conceptually be placed anywhere in a global order. It is trivial to respect program order with this placement, so they can be safely matched locally.

Combining set and map operations exposes more nuances in the requirements for consistency. Insert and lookup operations performed by different tasks are inherently unordered unless synchronized externally. Therefore, a batch of these operations can be committed globally in parallel, since they are guaranteed not to conflict with each other. Note that if an insert finds that its key is already in the combiner object, it does not need to send its own message. However, it must still block until that insert is done to ensure that a subsequent operation it performs cannot be reordered with it, preserving program order. Similarly, lookups can piggy-back on other lookups of the same key.

Using intuition from store/write buffers in modern processors, it is tempting to allow a lookup to be satisfied locally by inspecting outstanding inserts. However, this would allow the local order in which keys were inserted to be observed. Then to preserve SC, that same order would need to be respected globally, forcing each batch to commit atomically with respect to other cores' batches. Enforcing this would be prohibitively expensive, so a cheaper solution is chosen: lookups only get their values from the global data structure, therefore batches can be performed in parallel.

These requirements only guarantee linearizability of operations on a single data structure. To guarantee sequential consistency with respect to all accesses, data-race freedom must be guaranteed by the program, as in the C++ memory model for shared memory.

4 Grappa FC Framework

To leverage the flat-combining paradigm in Grappa, we implemented a generic framework to improve performance for a number of global data structures. The FC framework handles the common problems of managing the combiners, handling worker wake-ups, and maintaining progress. When hooking into the framework, each data structure need only define how to combine operations and globally commit them.

As mentioned before, the Grappa FC framework takes a different approach than the original flat-combining work for expressing how operations combine. For each global structure instance, a *proxy* object is created on each core and each worker merges its request into that structure before blocking or becoming the combiner. Each global data structure must define a *proxy* that has *state* to track combined requests, *methods* to add new requests, and a way to *sync* the state globally. An example proxy declaration for the GlobalStack is shown in Figure 5.

The FC framework is responsible for ensuring that all of the combined operations eventually get committed. There are a number of ways progress could be guaranteed, but one of the simplest is to ensure that as long as there are any outstanding requests, at least one worker is committing a combined

```
// Global master state
class GlobalStack {
  GlobalAddress<T> top;
  Grappa::Mutex lock;
};
// Push *without* combining
void push(GlobalAddress<GlobalStack> master, T e){
  // from stack's master core, perform write to top and increment atomically
  delegate::call(master.core(), [master,e]{
    master->lock.acquire();
    delegate::write(master->top, e);
    master->top++;
    master->lock.release();
  }); // blocks until response arrives
}
// Definition of proxy
class GlobalStackProxy : Grappa::FCProxy {
  GlobalAddress<GlobalStack> master;
  // Local state for tracking requests
  T  pushed_values[1024];
  T* popped_results[1024];
  int npush, npop;

  // Combining Methods
  void push(T val);
  T pop();

  // Global sync (called by FC framework)
  void sync() override {
    if (npush > 0) {
      // on master: acquire lock, return top ptr
      auto top = delegate::call(master.core(),[=]{
        master->lock.acquire();
        return master->top;
      });
      // copy values to top of stack
      Grappa::memcpy(top, pushed_values, npush);
      // bump top and release lock
      delegate::call(master.core(),[master,npush]{
        master->top += npush;
        master->lock.release();
      });
    } else if (npop > 0) {} // elided for space...
  }
};
```

Figure 5: Snippet of code from GlobalStack.

operation. When that combined operation finishes, if there are still outstanding requests that have built up in the meantime, another blocked worker is chosen to do another combined synchronization.

While a combining operation is in flight, new requests continue to accumulate. The framework transparently wraps the proxy object so when one sync starts, it can direct requests to a fresh combiner. This is done by hiding instances of proxy objects behind a C++ smart-pointer-like object which provides the pointer to the current combiner, and whenever it detects that it should send, allocates a new combiner and points subsequent references to that.

4.1 Global Stack and Queue

Figure 5 shows an excerpt from the definition of the proxy object for the Grappa GlobalStack. The proxy tracks pushes in the `pushed_values` array. When `pop` is called, if there are pushes, it immediately takes the top value and wakes the last pusher. Otherwise, it adds a pointer to a location on its stack where the `sync` operation will write the result. Because of local matching, when a Stack proxy is synchronized, it will have either all pops or all pushes, which makes the implementation of sync straightforward. Note that the operation to synchronize a batch of pushes looks almost identical to the code to do a single push from Figure 5.

The GlobalQueue has nearly the same implementation as the stack, but is unable to match locally.

4.2 Global Set and Map

The Grappa GlobalSet uses a simple chaining design, implemented with a global array of cells (allocated from the global heap), which are partitioned evenly among all the cores, and indexed with a hash function. Our flat-combining version supports both `insert` and `lookup`. To track all of the keys waiting to be inserted, we use the hash set implementation from the C++11 standard library (`std::unordered_set`). As with pops, lookups must provide pointers to stack space where results should be put, which is done with a hash map (again from the C++ standard library) from keys to lists of pointers. As discussed in Section 3.2, matching lookups with inserts locally would force a particular sequential order. Instead, we only allow matching inserts with inserts and lookups with lookups, allowing `sync` to simply issue all inserts and lookups in parallel and block until all have completed.

Our implementation of the GlobalMap matches that of the Set but of course stores values.

5 Evaluation

To evaluate the impact flat combining has, we ran a series of experiments to test the raw performance of the data structures themselves under different workloads, and then measure the impact on performance of two graph benchmarks. Experiments were run on a cluster of AMD Interlagos processors. Nodes have 32 2.1-GHz cores in two sockets, 64GB of memory, and 40Gb Mellanox ConnectX-2 InfiniBand network cards connected via a QLogic switch.

5.1 Data Structure Throughput

First we measured the raw performance of the global data structures on synthetic throughput workloads. In each experiment, a Grappa parallel loop spawns an equal number of tasks on all cores. Each task randomly chooses an operation based on the predetermined "operation mix," selecting either a push or pop for the Stack and Queue, or an insert or lookup for the Set and Map.

Queue and Stack. Figure 6 shows the results of the throughput experiments for the global Stack and Queue. Results are shown with flat combining completely disabled, only combining at the master core ("centralized"), and combining locally ("distributed").

Despite Grappa's automatic aggregation, without combining, both the stack and queue completely fail to scale because all workers' updates must serialize. Though centralized combining alleviates some of the serialization, its benefit is limited because all operations involve synchronization through a single core. However, with local flat combining, synchronization is done mostly in parallel, with less-frequent bulk synchronization at the master.

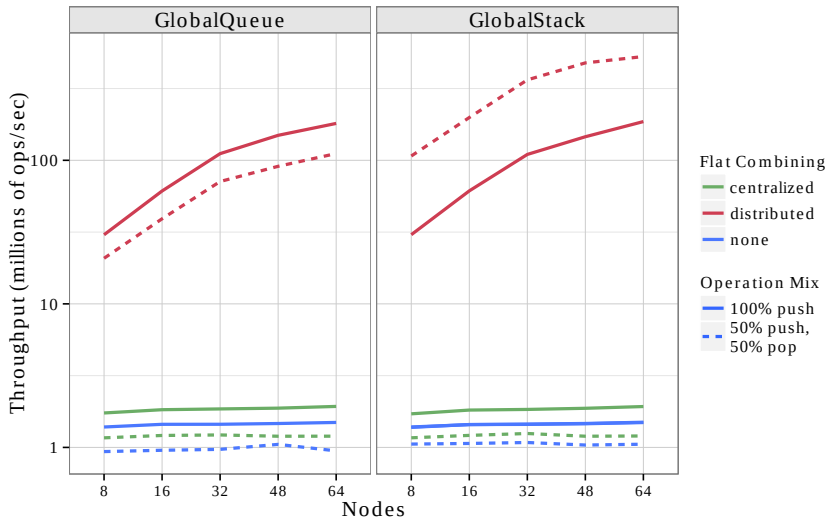


Figure 6: *Queue and Stack performance.* Results are shown on a log scale for a throughput workload performing 256 million operations with 2048 workers per core and 16 cores per node. Local flat combining improves throughput by at least an order of magnitude and allows performance to scale. Matching pushes and pops enables the stack to perform even better on a mixed workload.

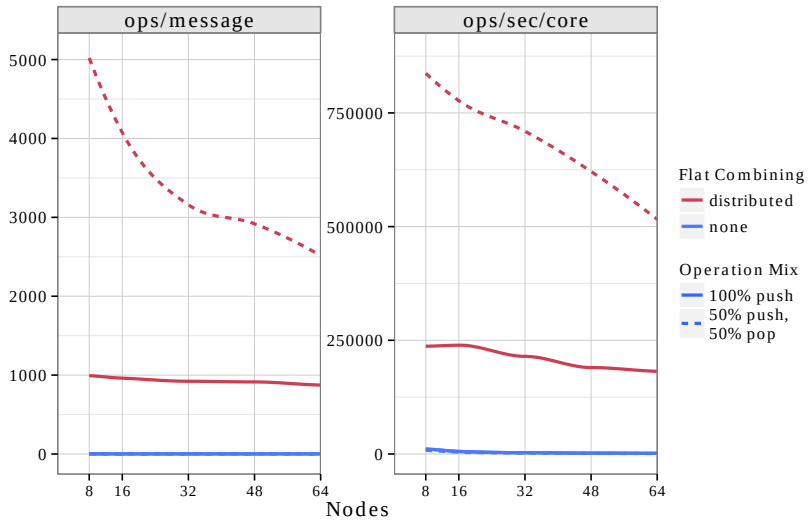


Figure 7: *GlobalStack Statistics.* Measured number of combined messages sent by the Stack with a fixed combining buffer of 1024 elements. Matched pushes and pops result in ops/message being greater than the buffer size.

On the mixed workload, the stack is able to do matching locally, allowing it to reduce the amount of communication drastically, greatly improving its performance. Figure 7 corroborates this, showing that the amount of combining that occurs directly correlates with the observed throughput.

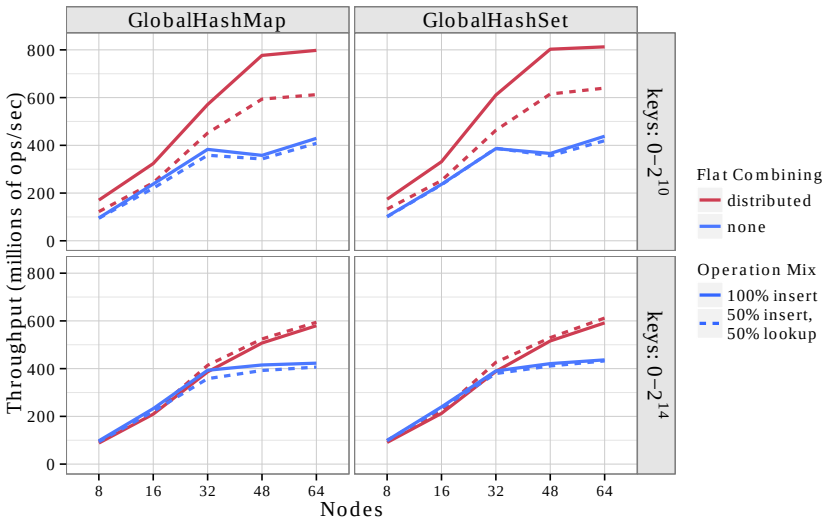


Figure 8: *GlobalHashSet and GlobalHashMap.* Results are shown for a throughput workload inserting and looking up 256 million random keys in a particular range into a global hash with the same number of cells, with 2048 workers per core and 16 cores per node.

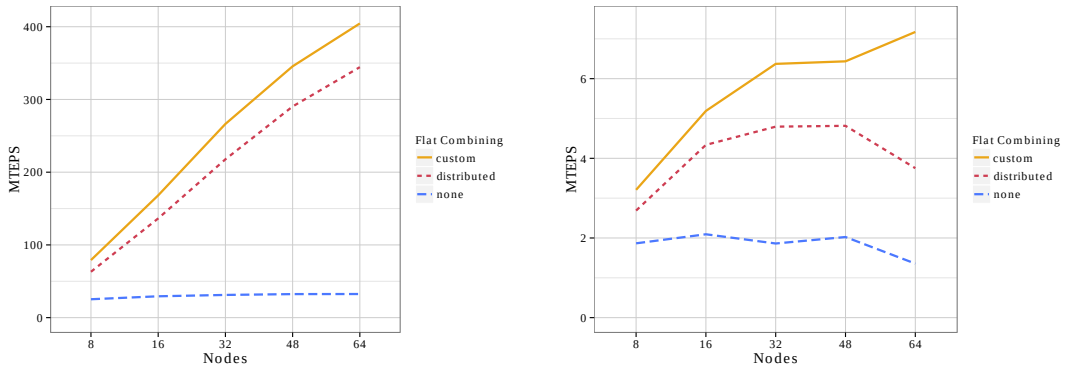
The queue benefits in the same way from reducing synchronization and batching, and its all-push workload performs identically to the stack's. However, the queue is unable to do matching locally, and in fact, the mixed workload performs worse because the current implementation serializes combined pushes and combined pops. This restriction could be lifted with more careful synchronization at the master core.

HashSet and HashMap. Figure 8 shows the throughput results for the Set and Map. Both data structures synchronize at each hash cell, which allows them to scale fairly well even without combining. However, after 32 nodes, scaling drops off significantly due to the increased number of destinations. Combining allows duplicate inserts and lookups to be eliminated, so performs better the smaller the key range. This reduction in message traffic allows scaling out to 64 nodes.

5.2 Application Kernel Performance

The Grappa data structures are synchronized to provide the most general use and match the expectations of programmers and algorithms. In these evaluations, we compare the flat-combining structures against custom, tuned versions that leverage relaxed consistency needs of the applications.

Breadth-First Search. The first application kernel is the Graph500 Breadth-First-Search (BFS) benchmark [11]. This benchmark does a search starting from a random vertex in a synthetic graph and builds a search tree of parent vertices for each vertex traversed during the search. The BFS algorithm contains a global queue which represents the frontier of vertices to be visited in each level. Our implementation employs the direction-optimizing algorithm by Beamer et al. [2]. The frontier queue is write-only in one phase and read-only in the next, making it amenable to relaxed consistency. We compare performance of BFS using the Grappa FC queue described above with a highly tuned Grappa implementation that uses a custom asynchronous queue.



(a) *BFS* with the direction-optimizing algorithm. (b) *Connected Components* using Kahan's algorithm.

Figure 9: Performance of graph kernels on a Graph500-spec graph of scale 26 (64 million vertices, 1 billion edges). Performance for both is measured in Millions of Traversed Edges Per Second (MTEPS).

Figure 9a shows the scaling results. Using the simple queue without flat combining is completely unscalable, but with FC, it tracks the tuned version. This illustrates that providing a safe, synchronized data structure for initially developing algorithms for PGAS is possible without sacrificing scaling.

Connected Components. Connected Components (CC) is another core graph analysis kernel that illustrates a different use of global data structures in irregular applications. We implement the three-phase CC algorithm [3] which was designed for the massively parallel MTA-2 machine. In the first phase, parallel traversals attempt to claim vertices and label them. Whenever two traversals encounter each other, an edge between the two roots is inserted in a set. The second phase performs the classical Shiloach-Vishkin parallel algorithm [22] on the reduced graph formed by the edge set from the first phase, and the final phase propagates the component labels out to the full graph. Creation of the reduced edge set dominates the runtime of this algorithm, but includes many repeated inserts at the boundary between traversals, so is a prime target for the flat-combining Set. Similar to BFS, the Set is write-only first, then read-only later, so further optimizations involving relaxation of consistency can be applied. Therefore, we compare our straightforward implementation using the generic Set with and without flat combining against a tuned asynchronous implementation.

The results in Figure 9b show that none of these three scale well out to 64 nodes. However, performing combining does improve the performance over the uncombined case. The tuned version outperforms the synchronous version because it is able to build up most of the set locally on each core before merging them at the end of the first phase. An implementation that did not provide synchronized semantics could potentially relax consistency in a more general way.

6 Related Work

Though much attention has gone to lock-free data structures such as the Treiber Stack [23] and Michael-Scott Queue [18], a significant body of work has explored ways of scaling globally locked data structures using combining to reduce synchronization and hot-spotting. Techniques differ mainly in the structures used to do combining: fixed trees [25], dynamic trees (or "funnels") [20, 17], and randomized trees [1]. In particular, MAMA [17], built for the MTA-2, leveraged funnels to better utilize hardware optimized for massive concurrency. On the other hand, *flat* combining [12] observed that in multicore systems,

hierarchical schemes have too much overhead, and a cleverly implemented *flat* publication list can perform better. Follow-on work introduced parallel flat-combining [13], in which multiple publication lists are combined in parallel and their remainders serialized. Flat combining was also applied in the NUMA (non-uniform memory access) domain with scalable hierarchical locks [9], which improves on prior work on hierarchical locking by leveraging flat-combining's publication mechanism which amortizes the cost of synchronization.

Our work extends the flat-combining paradigm further, to the PGAS domain, where only software provides the illusion of global memory to physically distributed memories. The relatively higher cost of fine-grained data-driven access patterns in PGAS makes flat combining even more compelling. In addition, the physical isolation per node combined with Grappa's engineered per-core isolation guarantees, create a novel situation for combining, which led to our *combining proxy* design.

In the distributed memory and PGAS domains, implementing scalable and distributed synchronization is the name of the game. Besides Grappa, other work has proposed leveraging latency tolerance in PGAS runtimes, including MADNESS [21] which proposes ways of using asynchronous operations in UPC, and Zhang et al. [26] who use asynchronous messages in a UPC Barnes-Hut implementation to overlap computation with communication and allow for buffering. Jose et al. introduced UPC Queues [16], a library which provides explicit queues as a replacement for locks to synchronize more efficiently and allow for buffering communication. This work demonstrates a similar use case for synchronized queues as in our work, and uses them with the Graph500 benchmark as well. In contrast, Grappa performs much of the same message aggregation invisibly to the programmer, and the FC framework can be used to implement many data structures.

7 Conclusion

Coming from the multi-core domain, the flat-combining paradigm provides a new perspective to global synchronization, bringing with it both challenges and new opportunities. We have shown that the additional concurrency that comes with a latency-tolerant runtime, rather than compounding the problem, provides a new opportunity for reducing communication by combining locally. In PGAS implementations there is typically a large discrepancy between the first simple description of an algorithm and the final optimized one. Our distributed flat-combining framework allows easy implementation of a library of flat-combined linearizable global data structures, allowing even simple applications that use them to scale out to a thousand cores and millions of concurrent threads.

References

[1] Y. Afek, G. Korland, M. Natanzon, and N. Shavit. Scalable producer-consumer pools based on elimination-diffraction trees. In *Euro-Par 2010-Parallel Processing*, pages 151–162. Springer, 2010.

[2] S. Beamer, K. Asanovi, and D. Patterson. Direction-optimizing breadth-first search. In *Conference on Supercomputing (SC-2012)*, November 2012.

[3] J. W. Berry, B. Hendrickson, S. Kahan, and P. Konecny. Software and algorithms for graph queries on multithreaded architectures. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–14. IEEE, 2007.

[4] H.-J. Boehm. A Less Formal Explanation of the Proposed C++ Concurrency Memory Model. C++ standards committee paper WG21/N2480 = J16/07-350, `http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2007/n2480.html`, December 2007.

[5] H.-J. Boehm and S. V. Adve. Foundations of the C++ concurrency memory model. In *ACM SIGPLAN Notices*, volume 43, pages 68–78. ACM, 2008.

[6] B. Chamberlain, D. Callahan, and H. Zima. Parallel programmability and the Chapel Language. *International Journal of High Performance Computing Application*, 21(3):291–312, Aug. 2007.

[7] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 519–538, New York, NY, USA, 2005. ACM.

[8] C. Coarfa, Y. Dotsenko, J. Mellor-Crummey, F. Cantonnet, T. El-Ghazawi, A. Mohanti, Y. Yao, and D. Chavarría-Miranda. An evaluation of global address space languages: Co-Array Fortran and Unified Parallel C. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 36–47. ACM, 2005.

[9] D. Dice, V. J. Marathe, and N. Shavit. Flat-combining NUMA locks. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 65–74, New York, NY, USA, 2011. ACM.

[10] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick. *UPC: Distributed Shared Memory Programming*. John Wiley and Sons, Inc., Hoboken, NJ, USA, 2005.

[11] Graph 500. `http://www.graph500.org/`, July 2012.

[12] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 355–364. ACM, 2010.

[13] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Scalable flat-combining based synchronous queues. In *Distributed Computing*, pages 79–93. Springer, 2010.

[14] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[15] ISO/IEC JTC1/SC22/WG21. ISO/IEC 14882, Programming Language - C++ (Committee Draft). `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2800.pdf`, 2008.

[16] J. Jose, S. Potluri, M. Luo, S. Sur, and D. Panda. UPC Queues for scalable graph traversals: Design and evaluation on Infiniband clusters. In *Fifth Conference on Partitioned Global Address Space Programming Models (PGAS11)*, 2011.

[17] S. Kahan and P. Konecny. MAMA!: A memory allocator for multithreaded architectures. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, pages 178–186, New York, NY, USA, 2006. ACM.

[18] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 267–275. ACM, 1996.

[19] J. Nelson, B. Myers, A. H. Hunter, P. Briggs, L. Ceze, C. Ebeling, D. Grossman, S. Kahan, and M. Oskin. Crunching large graphs with commodity processors. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Parallelism*, HotPar'11, pages 10–10, Berkeley, CA, USA, 2011. USENIX Association.

[20] N. Shavit and A. Zemach. Combining funnels: A dynamic approach to software combining. *Journal of Parallel and Distributed Computing*, 60(11):1355–1387, 2000.

[21] A. Shet, V. Tipparaju, and R. Harrison. Asynchronous programming in UPC: A case study and potential for improvement. In *Workshop on Asynchrony in the PGAS Model*. Citeseer, 2009.

[22] Y. Shiloach and U. Vishkin. An O(N log(N)) parallel max-flow algorithm. *Journal of Algorithms*, 3(2):128–146, 1982.

[23] R. K. Treiber. *Systems programming: Coping with parallelism*. International Business Machines Incorporated, Thomas J. Watson Research Center, 1986.

[24] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: A mechanism for integrated communication and computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ISCA '92, pages 256–266, New York, NY, USA, 1992. ACM.

[25] P.-C. Yew, N.-F. Tzeng, and D. H. Lawrie. Distributing hot-spot addressing in large-scale multiprocessors.

Computers, IEEE Transactions on, 100(4):388–395, 1987.

[26] J. Zhang, B. Behzad, and M. Snir. Optimizing the Barnes-Hut algorithm in UPC. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 75. ACM, 2011.