Clamor: Extending Functional Cluster Computing Frameworks with Fine-Grained Remote Memory Access

Pratiksha Thaker
Stanford University
prthaker@stanford.edu

Hudson Ayers
Stanford University
hayers@stanford.edu

Deepti Raghavan
Stanford University
deeptir@stanford.edu

Ning Niu
Stanford University
nniu@stanford.edu

Philip Levis
Stanford University
pal@stanford.edu

Matei Zaharia
Stanford University
matei@cs.stanford.edu

Abstract

We propose Clamor, a functional cluster computing framework that adds support for fine-grained, transparent access to global variables for distributed, data-parallel tasks. Clamor targets workloads that perform sparse accesses and updates within the bulk synchronous parallel execution model, a setting where the standard technique of broadcasting global variables is highly inefficient. Clamor implements a novel dynamic replication mechanism in order to enable efficient access to popular data regions on the fly, and tracks fine-grained dependencies in order to retain the lineage-based fault tolerance model of systems like Spark. Clamor can integrate with existing Rust and C++ libraries to transparently distribute programs on the cluster. We show that Clamor is competitive with Spark in simple functional workloads and can improve performance significantly compared to custom systems on workloads that sparsely access large global variables: from 5× for sparse logistic regression to over 100× on distributed geospatial queries.

CCS Concepts

• **Computer systems organization → Processors and memory architectures**; **Distributed architectures**.

Keywords

data analytics, remote memory, fault tolerance

SoCC '21, November 1–4, 2021, Seattle, WA, USA

ACM ISBN 978-1-4503-8638-8/21/11.
https://doi.org/10.1145/3472883.3486996

ACM Reference Format:
Pratiksha Thaker, Hudson Ayers, Deepti Raghavan, Ning Niu, Philip Levis, and Matei Zaharia. 2021. Clamor: Extending Functional Cluster Computing Frameworks with Fine-Grained Remote Memory Access. In *ACM Symposium on Cloud Computing (SoCC '21), November 1–4, 2021, Seattle, WA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3472883.3486996

1 Introduction

In frameworks such as MapReduce [29], Spark [78], and Dask [66], programmers write single-node "driver" programs that periodically launch parallel functional operations such as maps and reduces. The framework automatically ships function closures (a piece of code and variables it depends on) to the workers to run in parallel. This bulk synchronous parallel (BSP) computation model has made large-scale distributed computing accessible to data analytics and machine learning programmers who are not distributed systems experts.

The tasks in BSP programs may depend on global variables constructed in-memory in the driver program, such as indexes, search trees, or lookup tables. Access to these structures is data-dependent, so the framework cannot easily partition them across workers. Instead, workers access these global variables using broadcast [26]: the driver ships the entire variable with the task closure to each worker machine.

In some modern applications, these global variables are large enough to make broadcast inefficient to impossible. For example, training machine learning models requires random access to global arrays of billions or trillions of parameters [48, 56]. Similarly, analyzing scientific datasets requires computing indexes over terabytes of data: some astrophysics applications compute a k-d tree over an entire astronomical dataset in order to perform nearest neighbors searches efficiently [62, 63]. These applications express a common programming pattern: they construct a large, global index that workers read, and possibly make intermediate updates based on the results of that computation. For variables that are gigabytes in size, this incurs both the communication overhead of broadcast as well as the memory overhead of storing the variables.

One common workaround to address this overhead is to make calls to an external application-specific index (such as

a key-value store) [6, 19, 49, 62, 72, 76, 77]. This strategy has two major problems. First, each application has to develop a custom approach to manage data models between the BSP framework and the data store. Second, an external data store loses the fine-grained fault tolerance of a BSP system: an external store like Redis, for example, does not integrate with Spark's lineage-based tracking.

In real-world settings, however, accesses to these large global variables are often *sparse*. Broadcasting the variable is both inefficient and unnecessary, because each task only accesses a small portion of the variable [48]. Sending only the parts of the index that are actually accessed would conserve both communication and RAM.

The key insight of this paper is that the constrained data access model between driver programs and workers in the BSP model enables efficient fine-grained state sharing when accesses are sparse. Shared variables are only writeable by the driver program, and only in between stages of cluster computation (that is, after a barrier). This access model is sufficient to express many practical workloads. While these workloads are also amenable to a solution based on distributed shared memory (DSM), the BSP execution model enables caching optimizations, fine-grained fault tolerance, and straggler mitigation that are not possible in generic DSM systems.

This paper presents Clamor, a cluster computing framework that integrates a BSP programming model with efficient random access to global variables. In order to provide finer-grained access to arbitrary global variables, Clamor allows workers to access the variable at the granularity of pages. Clamor's page abstraction enables accessing and caching just the relevant parts of the variable in local memory, at a granularity suitable to the structure of the underlying data structure.

Integrating fine-grained data access into a BSP computing system requires solving two challenges to reconcile their different access patterns and failure models efficiently.

The first problem is providing lineage-based fault-tolerance, a key feature of Spark. Spark constructs a task dependency graph from a program and uses it to reconstruct failed tasks. Integrating lineage-based fault tolerance with global shared variables is difficult for two reasons. First, Spark tracks coarse-grained lineage for broadcast variables, serializing the entire variable when any part is modified, which results in prohibitive time and storage overhead for the large variables that Clamor targets. Instead, Clamor must incorporate fine-grained history for parts of global variables, not just partitioned datasets. Second, a traditional BSP system can statically construct a lineage graph when a job is submitted because the access pattern is known in advance, while a system supporting fine-grained global variable access must construct the dependency graph on the fly as tasks make random accesses.

The second problem is that page-based access can bottleneck at the driver if it serves "hot" pages accessed by many workers simultaneously. The access pattern is not known until runtime, so replicating pages in advance is not possible. Clamor implements a dynamic replication mechanism that allows workers to serve requests for pages they have cached. This replication leverages the fact that global variables do not change during a BSP stage: workers can freely cache and serve cached pages within a stage.

We implement Clamor in Rust and C++ and provide a C++ API to express programs in the BSP model. Moreover, Clamor's memory allocator is a drop-in replacement for `malloc`, which allows us to use existing third-party libraries, such as the Rust `kdtree-rs` library [42], unmodified in our benchmarks.

We evaluate Clamor on several random-access workloads, including a geographic lookup benchmark and distributed sparse logistic regression training. We show that Clamor can outperform custom systems designed to support these queries in the BSP model, performing up to 5× better on sparse logistic regression and over 100× better on geospatial queries. We also show that Clamor's performance is within 2-3× of hand-optimized MPI on standard batch workloads including distributed k-means clustering.

In summary, our contributions are:

- We identify a class of workloads that require sparse random access to large global variables within the BSP execution model;
- We implement an efficient remote memory system, Clamor, to serve this access pattern, taking advantage of the restricted execution model to implement efficient caching for hot pages;
- We integrate Clamor with lineage-based fault tolerance by tracking fine-grained lineage within large global variables, as well as straggler mitigation made possible by the restricted execution model, and
- We evaluate Clamor on three random-access workloads, as well as standard functional data-parallel workloads, and show that it can outperform custom solutions for these workloads when accesses and updates are sparse.

2 Motivation

Clamor is motivated by large scale data-processing workloads that rely on sparse, data-dependent access to large global variables. Because accesses to these structures are data-dependent, a driver cannot know a priori which part of the index a given worker will need and so cannot partition it. As the index structure grows, broadcasting it in entirety to every worker requires too much RAM or takes too long. This section describes three common workload examples that fit this pattern: geographic lookups, parameter servers, and streaming updates. Existing BSP systems cannot efficiently express their sparse random access patterns, and significant developer effort has gone into building custom solutions to scale these workloads.

Geographic lookups. A common geometric computation is k-nearest neighbors (k-NN): examples include lookups on 2D geographic coordinates [70] and querying massive 3D datasets in astrophysics [62, 63]. k-nearest neighbors queries are efficiently supported by k-d trees[1], a space-partitioning data structure [18]. A k-d tree's key benefit of geometric locality, however, is directly in conflict with how key-value pairs spread data across keys. As a result, cluster frameworks cannot efficiently support queries to a k-d tree. Naïve solutions [5] build the k-d tree on the driver and broadcast it to workers. Other approaches include storing reference locations using the geospatial API of an external Redis server [70] and implementing a custom partitioner for k-d tree-like functionality within a Spark RDD [62].

Parameter servers. Modern machine learning workloads train models with billions or trillions of parameters [48, 56]. Having every worker store every parameter requires huge amounts of RAM and is therefore prohibitively expensive. The parameter server architecture [48] addresses this problem and improves model training efficiency by allowing workers to pull only the parameters they need from an external key-value store and push gradient updates. Each worker executes the same gradient update computation on its local training data. The architecture's ability to scale up to huge models has made it the dominant paradigm for distributed training.

While developers have built custom systems for parameter serving, there is nevertheless significant interest in building parameter servers on top of existing systems like Spark, reusing familiar infrastructure that users expect to scale [19, 38, 49, 76, 77]. Existing solutions include broadcasting all of the parameters [49, 76] or implementing custom partial broadcast mechanisms [77]. Glint [38] integrates Spark with a key-value store, but sacrifices Spark's lineage-based fault tolerance to do so.

Streaming updates. Many streaming applications require periodic updates to broadcast variables: for example, to update lookup tables based on time-series data ingested from an external queue [14, 20, 65, 69, 73] or update a global machine learning model as new data arrives [21]. Broadcast variables, however, are immutable, so the only way to update the workers is to copy and re-broadcast the whole variable [73]. Broadcasting the entire data structure to workers after a sparse update is extremely wasteful and imposes a high cost. The inefficiency of this solution has led some users to rely on custom solutions based on an external key-value store, which loses both the fault tolerance guarantees as well as the optimized broadcast implementation of Spark [65].

[1]k-d means k-dimensional; the k in k-d is distinct from the k in k-nearest neighbors.

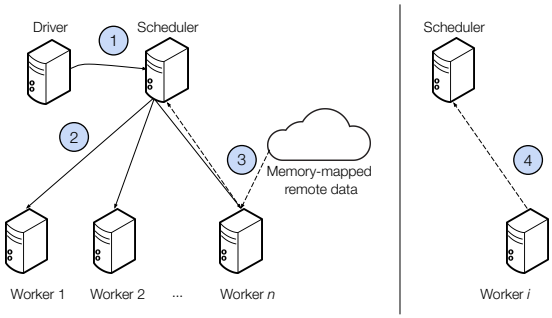


Figure 1: The execution of a Clamor stage. (1) The user provides a driver program that executes on the primary. The driver program submits stages for execution to the scheduler, which (2) dispatches tasks to the workers. (3) The workers may load data from remote storage, and make requests for driver pages to the scheduler while executing tasks. (4) When a worker completes, it registers the completion with the scheduler, sending its result along with the pages it read during execution.

This access pattern is common, and occurs in many other modern data analytics and machine learning workloads, such as lookups into word or graph embeddings for recommendation systems [2, 56, 57], similarity search via locality-sensitive hashing [24], and evaluating large decision forests [1].

2.1 The Case for Fine-Grained Sharing

These three workloads share a common pattern: the centralized driver program builds a large data structure that workers sparsely access. Because these accesses are data-dependent, the program cannot easily partition and ship only the relevant parts of the data structure with the task. Instead, the driver must broadcast the entire variable every time a computation updates it. As a workload is parallelized over more workers, the constant cost of this broadcast takes a larger fraction of execution time. As a result, applications scale by using specialized systems or extensions, such as external services or custom partitioners, costing significant developer effort.

A system that integrates fine-grained global variable access into a BSP programming model could replace all of these systems. While the solution of a key-value store is appropriate for some workloads, such as the parameter server, we aim to support a generic set of variables including the k-d tree and other data structures, such as decision trees, that are not effectively supported by a flat key-value layout. Additionally, since these global variables are constructed on the driver, the driver can become a bottleneck in serving data; thus, the system must implement fine-grained replication that additionally scales with the access pattern of the workload. Finally, we aim to support the fine-grained fault tolerance benefits of systems like Spark while also supporting in-place updates to global variables, which Spark does not support. The key challenge

in doing so is implementing lineage for global variables at a fine granularity rather than serializing and storing the entire variable each time it is updated.

3 Overview

Clamor enables random access to global variables within the restrictions of the BSP computation model. In this section, we review the computational model, describe how we implement random addressing into global variables, and describe the Clamor programming model.

3.1 Cluster and computation model

Figure 1 provides an overview of the Clamor cluster model. Clamor programs run in a cluster consisting of a *primary* node that launches the driver program and *worker* nodes that run parallel computation.[2] The primary additionally runs the *scheduler* that schedules computation on the cluster. The scheduler and driver run on the same node by default, although these functionalities are not coupled and can run on independent nodes. In Clamor, the scheduler handles task scheduling, but also serves requests for parts of driver memory, which we elaborate on in the next subsection.

In the bulk synchronous parallel computation model [71], workers execute computation in parallel in between global synchronization barriers. In this paper, we refer to the computation between two barriers as a *stage*, divided into data-parallel, functional *tasks* of work. Results of a task are only globally visible after the corresponding synchronization barrier.

Within stages, Clamor executes *functional* and data-parallel tasks, a common model for modern data-parallel computation [29, 61, 66, 78]. Thus, global variables passed to a task are immutable during the execution of a stage, and can only be updated by the driver in between stages (i.e., they cannot be directly written to by tasks on workers).

3.2 Enabling global variable access

In order to efficiently enable random access to global variables, Clamor equips the driver with a globally-addressable static array and a transparent memory allocator to use for constructing variables on the global array. Rather than broadcasting an entire global variable to each task, the scheduler ships only the address of the variable in the global array. Workers can use this memory address to randomly access this global variable while executing tasks. The global memory region is divided into contiguous segments which we refer to as *pages*, which we discuss further in Section 4.

When a worker accesses the global array, it makes a request to the scheduler to for the page corresponding to that address, copying the data into the same address region in its local memory (analogous to OS demand paging). Importantly, the scheduler handles both task scheduling as well as page requests, because it must track the pages associated with a particular worker and task in order to track the *lineage* (dependency graph) of data dependencies for a task as the task executes. The lineage can later be used to rerun the task deterministically if the worker fails. We elaborate on this functionality in Section 5.

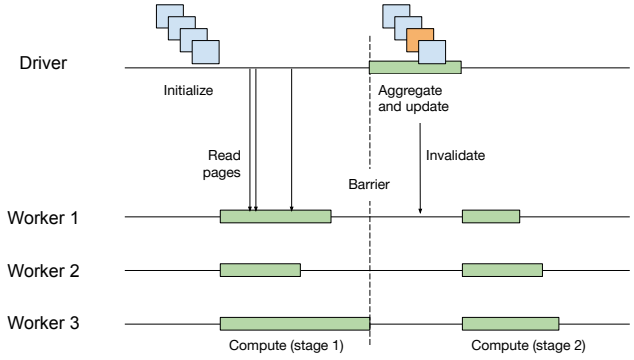


Figure 2: Diagram of Clamor execution. The driver initializes global variables in the shared array. Workers compute in parallel, reading and caching pages as the program accesses addresses in the shared array (requests only shown for worker 1). The driver waits for all workers to complete before aggregating results and writing updates to the global variable. When the driver makes a write to a page, it invalidates any existing readers of the page. The unmodified pages (blue) can remain cached for both stages 1 and 2.

The solution of a globally addressable array has a close relationship to DSM, which also aims to provide random access to memory across a cluster. In contrast to general DSM, the functional data-parallel execution model ensures that only the driver writes to the shared address space in Clamor, and updates to the variable happen only after barriers. These guarantees allow for a number of communication and caching optimizations that make Clamor both generic and efficient for the workloads and execution model it targets. This restricted write model additionally allows Clamor to implement lineage-based fault tolerance, where most DSM systems must rely on coarse-grained checkpointing that is too expensive for short-lived data analytics workloads. Existing BSP systems can only implement lineage for global variables at the granularity of the entire variable, serializing and storing it even when it is large.

We note that although Clamor enables partial broadcast via remote memory, this solution is not mutually exclusive with existing mechanisms to share variables. For instance, when variables are smaller than the page size, it can be more efficient to ship the variable along with the task closure (if the variable is accessed by all tasks) because the page access

[2]The assumption of a centralized driver is standard in comparable frameworks such as Spark [78], Hadoop [68], or Dryad [37].

```
1  val hm = new HashMap(input_data)
2  val bhm = hm.broadcast()
3  val input = sc.textFile("s3://input-file")
4  val values = input.map(x =
       > bhm.get(x)) // Expensive: broadcast entire hash map
5  val sum = inputs.reduce(_+_)
6  values.map(x => x / sum)
```

Listing 1: Pseudocode to look up values in a hash table and then normalize the resulting vector, in Spark. The large table is shipped to all workers in the call to `input.map()`, even though the workers may only make lookups to a few keys.

```
// Hash map lookup function
int64 hash_map_lookup(int64_t key, hash_map* hm) {
  return hm->lookup(key);
};
```

```
// Construct the hash map in the driver.
hash_map* hm = clamor_hash_map(input_data);

DataCollection<int64> inputs("s3://input-file");

// Look up the keys in the hash map.
DataCollection<int64> values
    = inputs->map(hash_map_lookup, hm);

// Sum the values and materialize the result.
int64 sum = values->sum()
                  ->execute();

// Normalize the values in a second stage.
// The resulting vector remains on the cluster
// when the stage completes.
auto result = values->map(divide_by, sum);
```

Listing 2: Pseudocode of a Clamor driver program that looks up values in a hash table and then normalizes the resulting vector. The driver program initializes the hash table using the Clamor memory allocator. This program performs a distributed `map` (bottom) that calls a user-defined function (top) to call the hash table's `get` function using the input keys. Unlike the Spark example, no explicit broadcast is performed. The pages corresponding to the lookup keys are retrieved on the fly.

pattern is completely predictable. While we focus in this paper on Clamor's performance on large variables, these strategies can be useful for keeping communication overheads low for smaller dependencies.

In the remainder of the paper, we will discuss the details of serving these pages efficiently, including caching and updating pages (§4.1), replication (§4.2), and tracking lineage for fault tolerance (§5).

3.3 Programming model

Clamor provides a simple user-facing C++ API for writing distributed tasks that conform to the functional BSP model. The API includes common operations on vectors such as `map`, `filter`, `reduce`, `join`, and `lookup`. Operations that accept lambda arguments, such as `map`, can be called using user-defined functions. Users instantiate a distributed dataset, `DataCollection<T>`, either from data already located on the driver or from data located in remote storage, e.g., an Amazon S3 bucket. They can then call API functions on the `DataCollection`, as shown in Listing 2.

In Clamor, stages are constructed implicitly: execution proceeds on workers in parallel until an aggregation or a local update on the driver, which runs only once the preceding stage completes. As an example, Listing 2 shows the Clamor code to normalize a distributed vector. The code first computes the sum of the vector; the workers compute local sums and then return those to the driver, which performs the final aggregation. The second stage divides the vector by the sum in parallel on the cluster.

4 Efficiently serving global variables

Clamor targets sparse accesses to global variables, but does not put further restrictions on the variables themselves. In order to enable efficient but generic memory access, Clamor serves memory at the granularity of *pages* (a contiguous memory region that may be larger than a standard OS page; we discuss page sizing further in Sections 6 and 7). The page abstraction allows for more efficient caching, updating, and fault handling at the cost of added communication overhead for accesses that have lower locality. We discuss these tradeoffs further in Section 6.

4.1 Caching and updates

The functional BSP model allows workers to cache driver pages for the duration of a stage, and possibly beyond. Global variables are immutable while workers execute tasks, and can only be written by the driver after a barrier.

The driver program can make updates to global variables after a barrier, in between stages. In the parameter server, for example, the driver updates the weights in between stages of synchronous SGD before launching the next iteration on the cluster. When the driver updates a memory address in the global address space, the scheduler first makes an invalidation RPC to all workers caching the corresponding page to clear their local caches. On the next access to the page, the worker retrieves the most recent version of the page from the driver memory.

Figure 2 illustrates task execution and caching in Clamor. The driver initializes a global variable that the workers read in parallel while executing tasks. The driver must wait for workers to complete the parallel tasks before writing to the global array again or performing an aggregation. While updating the global array, the driver invalidates any readers of the pages that are updated, but pages that are never updated can stay cached at the workers for later stages.

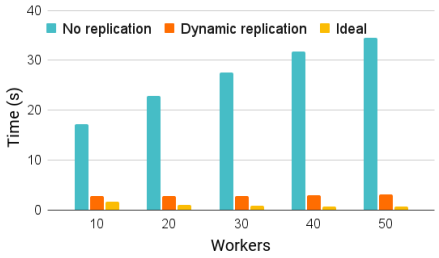


Figure 3: Performance of the dynamic replication mechanism, which improves performance by over 11× on a network-intensive sparse lookup workload.

4.2 Replicating hot pages

As the number of workers grows, the primary can become a bottleneck in serving pages. Static replication is not possible because the access pattern is not known in advance, and the global variables that Clamor supports are too large to replicate in their entirety.

In order to resolve this problem, we introduce a simple but novel *dynamic replication* mechanism, that takes advantage of the caching enabled by the BSP model to efficiently serve pages from multiple machines: the scheduler simply redirects requests to a random worker that has previously requested the page. While the mechanism is simple, it is enabled by the fact that workers can cache pages for the entirety of a stage, and is fast enough to load balance for stages that complete in seconds. This is analogous to the peer-to-peer mechanism implemented to efficiently broadcast entire variables [26], but our fine-grained access model improves over broadcast because only the parts of the variable actually accessed need to be communicated. Moreover, broadcast requires serializing and deserializing the entire variable, while Clamor transparently sends pages.

In Figure 3 we run a simple microbenchmark to illustrate the benefit of dynamic replication, on clusters of machines with 10 Gbit/s Ethernet each and a page size of 256 KB. The microbenchmark makes 10 million Zipf-distributed lookups to a 1 GB hash table on the driver. Because the lookups are Zipf-distributed, most workers request the same pages, but *which* pages are not known in advance, so they cannot be trivially replicated. We plot the time to execute this workload with dynamic replication against the baselines of no replication and the ideal runtime possible if all of the aggregated bandwidth in the cluster were utilized. Without replication, the workload takes 17 seconds on a 10-node cluster and 35 seconds on a 50-node cluster, slowing down as the driver becomes a bottleneck in serving pages. With dynamic replication, the runtime is near constant at 3 seconds, while the ideal runtime is 1.6 seconds on 10 nodes and 0.65 seconds on 50 nodes.

We note that well-known results on the "power of two choices" [54] suggest, in a slightly different service model, that the uniform random strategy would result in load skew at the servers as the cluster size increases; although Figure 3 shows that this skew is not significant for the cluster sizes in our experiments, exploring an implementation of the power of two choices in our dynamic setting would be an interesting direction for future work.

5 Fault tolerance and straggler mitigation

Frameworks that enable remote memory access, including most systems that implement some form of DSM, rely on coarse-grained or whole-system checkpointing for fault tolerance. These approaches work well in a high-performance computing setting, because they can be carefully tuned to supercomputer reliability and highly engineered applications that are developed over years and run for days. For dynamic data analytics workloads, however, their coarse-grained behavior greatly increases tail latencies. Clamor integrates with the fine-grained lineage-based fault tolerance model that users have come to expect from popular functional frameworks like Spark that target short-lived batch analytics workloads. In this section, we review the lineage-based fault tolerance model, describe the challenges of integrating Clamor's global variables with this model, and describe how we address them.

5.1 Lineage-based fault tolerance

Lineage-based fault tolerance tracks the sequence of operations used to construct a result rather than logging the intermediate data itself. When a worker fails, a backup can deterministically reconstruct the lost data from the sequence of operations. Systems like Spark [78] can efficiently implement lineage because of the functional programming model that implements transformations on coarse partitions of a dataset. The coarse-grained access model, however, limits Spark's ability to implement lineage for global variables: it must serialize and store the entire variable in order to include it in a task's lineage, even if the task only accesses a small part of the variable, and if the variable is updated, it must be re-serialized and stored separately, even if the update is very sparse. Clamor aims to preserve the benefits of lineage-based fault tolerance while reducing the overheads that result from variable updates.

5.2 Tracking lineage in Clamor

Because Spark only tracks coarse-grained lineage, the program is sufficient to statically determine the lineage before the program executes. In contrast, Clamor tracks lineage at page granularity, and must do so on the fly as the program executes because the access pattern is not known in advance.

One subtlety that arises in reconstructing tasks in Clamor is if a worker fails in the middle of executing some task t. The lineage of that task is incomplete because it did not finish executing. Thus, we must wait for *all* lost data to be recovered – not just the known dependencies – before relaunching task t

```
KDTree* kd = construct_kdtree(old_locations);

DataCollection<double[2]> queries("s3://input-file");
DataCollection
      <double> distances = queries->map(kdtree_lookup, kd);

kd.erase(old_locations[0]);
kd.insert(new_location);

DataCollection
      <double> new_distances = locs->map(kdtree_lookup, kd);

DataCollection<int32> preferences =
      distances->zip(new_distances)
               ->map(argmin);
```

Listing 3: Pseudocode for a program that computes distances in a k-d tree, updates the k-d tree in place with new locations, recomputes the distances, and then determines which of the two location sets has the closer distances. If a worker fails after the update to the k-d tree, a new worker needs both the old and new versions of the k-d tree in order to re-execute and complete the task.

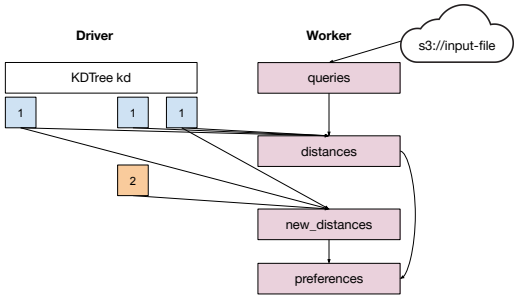


Figure 4: Lineage graph for the query in Listing 3. The colored boxes on the left side represent pages of the global memory region, where the k-d tree is located. The driver updates one page after the first stage completes. If a worker fails and loses its partitions of `distances` and `new_distances`, a new worker needs the version of the k-d tree both from stage 1 (to recompute `distances`) and stage 2 (to recompute `new_distances`) in order to complete the task.

in case task t depends on some of that data. (An optimization, which we implement, is to speculatively launch t when its known dependencies are available, and abort if some future dependency is still missing.) This subtlety does not arise in Spark because dependencies can be statically computed from the program.

5.3 Page versioning

In contrast to standard BSP systems where broadcast variables are immutable, in Clamor, the driver program makes in-place updates to global variables. In order for backup workers to deterministically reconstruct old stages, they may need access to previous versions of a driver page that have since been modified.

We illustrate this problem using the simplified example in Listing 3. This program executes distributed lookups into a k-d tree constructed on the driver. The driver then updates the k-d tree, and then queries the updated k-d tree using the same query set, aggregating the results with an `argmin` between the two queries. (For instance, this query could be used to determine the best set of locations for public water fountains based on their proximity to a query set of foot traffic data.)

Figure 4 shows the lineage graph for this query. The driver updates the k-d tree in between the computation of `distances` and `new_distances`, but `preferences` is computed last and depends on both of these results. If a worker fails after the driver update, a backup cannot recompute the lost data from `distances` without having access to the previous version of the k-d tree.

In order to remedy this, Clamor tracks versions of pages when they are updated. When the driver submits a new stage, all pages in the global address space are marked as copy-on-write. When the driver next writes the page, the previous version of the page is logged alongside the sequential index of the stage prior to the barrier. When a worker reconstructing a task from stage i accesses a page, the driver returns the latest version of the page with stage ID less than or equal to i. This protocol allows Clamor to track changes to the global variable at an intermediate granularity between recording every update independently and serializing the entire variable at each update.

In the example in Figure 4, only a single driver page is updated, so only that page needs to be versioned in order to support failure recovery.

In the current implementation, the versions are recorded in the scheduler RAM, as scheduler failures are not in scope for our fault tolerance model; however, page versions can trivially also be written to and retrieved from stable storage.

5.4 Straggler mitigation

The BSP model additionally allows Clamor to integrate speculative execution to mitigate slowdown from stragglers during a stage. Since global variables are immutable while the stage executes, and results of a stage are not globally visible until the end of the stage, the scheduler is free to launch replicas of tasks without causing write conflicts. Execution on the workers is deterministic, so once any one replica returns a result for a given task, the scheduler can abort the remaining replicas. We evaluate the performance of straggler mitigation in Section 7.4.

6 Implementation

We implemented a prototype of Clamor in C++ and Rust. Clamor's task manager and client library are implemented in 6850 lines of C++, and enabling integration with Weld [61] (described in § 6.1) required an additional 1400 lines of Rust

to implement the compiler pass that parallelizes Weld IR expressions across distributed workers. Clamor uses the gRPC library [36] for communication between nodes.

6.1 Weld compiler pass

Clamor represents parallel and functional programs using the Weld intermediate representation [61], a low-level representation of data-parallel computation that is expressive enough to capture parallel, functional, relational, and linear algebra computations. The Weld IR can support standard `map` and `reduce` operators, but provides a more general API to express data-parallel `for` loops and a compiler that performs optimizations such as loop fusion that Clamor can also take advantage of. Although Weld is a relatively low-level representation, it is versatile and expressive enough to allow for higher-level APIs to be built on top of it. We envision users ultimately interfacing with Clamor through such APIs and through Weld's integration with existing, popular data science libraries.

In order to support distributed computation in Weld, we introduce a compiler pass to the Weld compiler that looks for top-level `for` loops over vectors and transforms them into a loop that partitions data and distributes data-parallel tasks across workers. This also allows Clamor to take advantage of Weld's loop fusion optimizations to automatically reduce the number of stages between aggregations on the driver.

We note that while Weld provides an expressive and general representation for the data-parallel computation that Clamor targets, it is not a fundamental part of Clamor's architecture and could also be replaced by other representations that express a subset of these computations (such as simpler `map` and `reduce` operators).

6.2 Shared address space

Clamor implements a shared address space as a static C array. Workers use `mprotect` to raise segmentation faults when accessing pages in the shared buffer and handle segmentation faults by requesting the corresponding page permission from the task manager via RPC.

In order to ensure that addresses are consistent across machines, Clamor requires ASLR to be disabled; however, to mitigate security concerns, address translation can be implemented with minimal overhead: when a worker traps a page fault corresponding to an absolute address, it can compute the offset of the address relative to the start of the shared buffer and request the page using the relative address (which is consistent across workers) rather than the absolute address.

6.3 Page size and metadata overhead

Clamor stores about 400 bytes of metadata per page (including locks, page state, reads and writes for lineage tracking, and source data information for pages backed by data files).

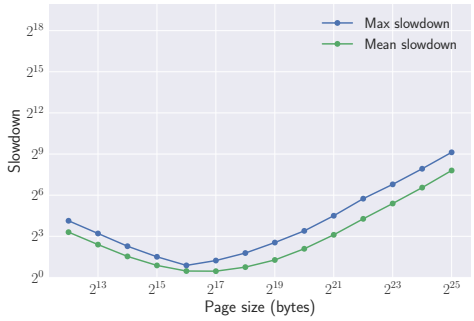
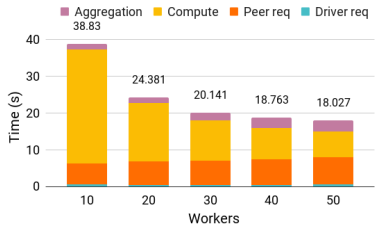


Figure 5: Mean and maximum slowdown relative to optimal page size for a range of transfer sizes between 4 KB and 32 MB.

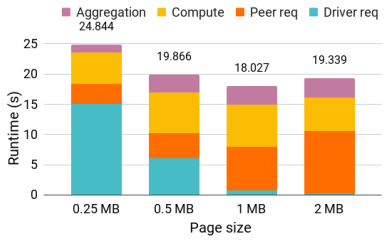
In order to determine a reasonable default page size for Clamor, we measured the time required to make data transfers for a range of page sizes using gRPC, the RPC framework used in Clamor. Specifically, we measured total transfer sizes (on nodes with 10 Gbit/s bandwidth) logarithmically distributed between 4 KB (the size of a typical OS page) and 32 MB, and varied the page size in the same range. For example, a transfer of 16 MB with a page size of 1 MB would require 16 RPCs to complete. The optimal page size for that transfer should be 16 MB, which requires only one RPC. For each page size, we measured the mean and maximum slowdown across transfer sizes, relative to the page size that would be optimal for that transfer size (i.e. page size equal to transfer size). The result is in Figure 5. In this range of transfer sizes, a page size of 128 KB minimizes the mean slowdown (1.37×), while 65 KB minimizes the maximum slowdown (1.83×). While this experiment suggests that 65 or 128 KB are reasonable default page sizes, we note that it does not take into account the effects of dynamic replication. The hit rate for dynamic replication improves as the page size increases, and we find in our evaluation that this time can often dominate the runtime, making larger pages more effective for many workloads.

7 Evaluation

Experimental setup. Unless otherwise stated, our experiments ran on `m5.8xlarge` Amazon EC2 machines with 16 physical cores, 128 GB of RAM, and 10 Gbit/s of bandwidth. Spark was configured to run one executor per node with 8 cores per executor, and Clamor runs 8 parallel workers per node. We compare against Spark version 2.4.3 and OpenMPI version 2.1.1. Clamor uses Weld version 0.3.0 compiled with LLVM 6.0 and Rust 1.43.0. For measurements that include data download time, data is downloaded from an Amazon S3 bucket located in the same region as the cluster (us-east-1, N. Virginia). (Measurements do not include data download time unless otherwise stated.)



(a) Scaling for k-d tree lookups with 1 MB pages. Compute time decreases linearly, but the time to aggregate results increases as the number of workers increases. "Peer req" refers to time spent retrieving pages from peers, while "Driver req" refers to time spent retrieving pages directly from the driver.



(b) k-d tree performance on 50 workers varying the page size. As the page size increases, the hit rate of dynamic replication increases and more pages can be retrieved from peers, but when the page size is larger than necessary, the excess communication overhead of page retrieval limits scaling.

Figure 6: Scaling for k-d tree lookup workload with varying page size.

7.1 Random access workloads

In this section, we evaluate three workloads involving transparent random access to a global variable constructed in the driver to show the performance benefits of Clamor in its target setting: sparse random access to large global variables.

7.1.1 k-d tree We constructed a k-d tree of 7 million nodes of locations across the United States downloaded from OpenStreetMap [7] and equip each with metadata of 2048 bytes (approximately the size of the metadata provided for businesses in the Yelp open dataset [8]) and a numeric rating 1-5. The resulting k-d tree is about 15 GB in size. We use the Rust library `kdtree-rs` [42] unmodified except for the Clamor memory allocator to construct the k-d tree. Serializing the entire tree using Rust's `serde` library takes 89 seconds, and deserializing takes 40 seconds, an overhead that Clamor bypasses. The total time required to run the workload in our experiments was not greater than 38 seconds, which is faster than the 89 seconds required just to serialize the k-d tree for a broadcast solution.

We evaluate a read-only workload of 1 billion queries to the k-d tree that computes the 10 nearest neighbors of each query point, sorts by rating, and finds the average distance to the top-rated neighbor across all query points. The query points are localized to the San Francisco bay area, a small region roughly between 36 and 39 degrees latitude and -123 and -120 degrees longitude.

Scaling. To understand the scale-out behavior of Clamor, in Figure 6 we plot the runtime of this workload on clusters of sizes between 10 and 50 nodes each running 8 worker processes.

In Figure 6a, we plot the runtime of this workload with large 1 MB pages. "Peer req" refers to time spent retrieving pages from other workers, while "Driver req" refers to time spent retrieving pages from the driver directly. The computation time decreases linearly as we increase the number of workers, and the network overhead of retrieving pages stays approximately constant as the number of workers increases due to dynamic replication. As the number of workers increases to 50 nodes, the benefits of scaling out decrease as the computation time approximately equals the network overhead of retrieving k-d tree pages.

In Figure 6b, we explore the effects of varying page size on the workload performance. We fix the cluster size at 50 workers and measure the runtime as the page size varies between 0.25 MB and 2 MB. At the small page size of 0.25 MB, the hit rate for dynamic replication is low and most requests must be served by the driver, creating a bottleneck. As the page size increases, the hit rate improves, but when the page size is too large at 2 MB, the workers incur an excess communication cost that limits scaling.

Comparison to Redis. Redis features an API specialized to geospatial queries [3], where users can add locations associated with values. We evaluated the benefit of Clamor over a custom Redis solution, by storing the 7 million input nodes in a Redis server using the geospatial API. We then ran a subset of the queries from 1 or more Redis clients (each running 8 threads). We found that Redis was too slow to complete the entire 1 billion point workload in a reasonable amount of time, and instead benchmarked a very small subset of the full benchmark. To complete only 10,000 queries, 1/1000 of the queries in the Clamor benchmark, a single Redis client takes about 50 minutes – 125× slower than the time to run Clamor on the *full* query set. Moreover, the Redis implementation does not benefit from scale-out: running this small benchmark on cluster sizes between 10 and 50 nodes results in a roughly constant runtime between 43 and 50 minutes.

We note two main differences that improve Clamor's performance compared to Redis. First, Redis does not build an actual k-d tree index over the points, instead storing them as a sorted set. Redis does not natively support nearest-neighbor queries: queries must provide a manual bounding box hint in order for nearest-neighbor searches to be efficient. In order to match Clamor's top-10 workload, we manually found the minimum size of bounding box such that all queries in the workload returned 10 results. Importantly, because Clamor builds a spatial

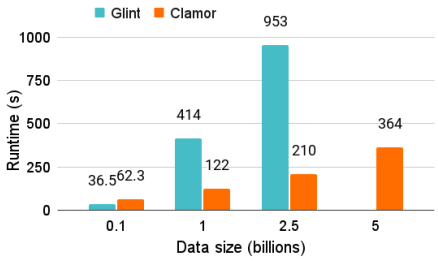
index over the queries, it does not need any manual hints and still achieves orders of magnitude better performance. Second, in Clamor, the nearest neighbor search is performed on the workers. In contrast, the single-node Redis server has to serve all of the client's bounding-box queries, limiting scalability.

7.1.2 Parameter server We constructed synthetic data sets following the schema for the Criteo ad click prediction benchmark [4] to benchmark sparse logistic regression. Each data point has 39 features with values uniformly generated according to the number of unique values in each integer valued Criteo data feature (bucketing the domain of integer values into 5 million buckets). We do not replicate the categorical values in the criteo dataset. For sparse logistic regression, the data points are one-hot encoded, resulting in extremely sparse accesses to the 1 billion global parameters.

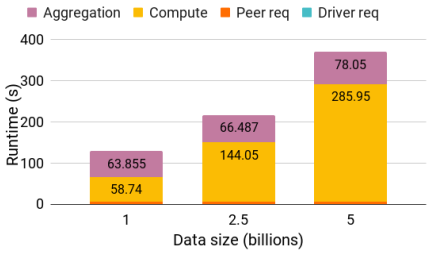
We perform 10 iterations of synchronous stochastic gradient descent (SGD) in each system and measure the average runtime per iteration. We compare Clamor's implementation of sparse logistic regression to two existing implementations of the parameter server in a BSP system: a naïve Spark implementation for which all updates to the global weights array are made by collecting to the driver node using broadcast, and a Spark implementation that uses Glint [38] as a parameter server to store the global weights.

Scalability. We first evaluate the behavior of these systems on a 50 node cluster, performing SGD with a 1 billion weight parameter vector and increasing numbers of data points. We start with a small 96 million data point set, and use it to generate larger synthetic datasets, as realistic workloads will have many more data points than parameters. The naïve Spark implementation using broadcast failed to complete after 7 hours on even the smallest dataset. (This is consistent with observations by the Spark MLlib developers that logistic regression fails to scale beyond tens of millions of parameters [52]). Therefore our evaluation compares Clamor and Glint. Figure 7a shows the performance of Glint and Clamor for increasing dataset sizes. This evaluation uses a 2 MB page size for Clamor, based on some tuning. Notably, Glint fails to scale to 5 billion data points because of an out of memory error. We were able to reduce this memory overhead by reducing the partition size, but a run with smaller partitions failed to complete in over 10 hours. Although Clamor performs worse than Glint for very small datasets (smaller than the number of parameters), it scales much better than Glint for realistic dataset sizes, performing 3.3× better on 1 billion points and 4.5× better on 2.5 billion points.

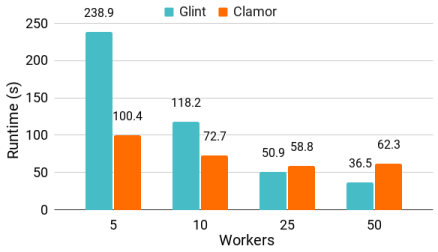
In order to understand this scaling behavior, we plot a breakdown of the Clamor runtimes in Figure 7b. As the data size increases, the compute time scales up linearly, while the aggregation time remains roughly constant because the features



(a) Performance of Glint and Clamor as the data size increases on a cluster of 50 nodes. For a workload of 5 billion data points, Glint fails to complete with an out of memory error while Clamor still runs in 6 minutes per iteration.



(b) Breakdown of runtime for Clamor varying data size on 50 nodes. As the data size increases, the aggregation time remains roughly constant and the compute time dominates, reducing the relative overhead of aggregation.



(c) Scaling for logistic regression on Glint and Clamor with 1 billion weights and a small number of data points (96 million).

Figure 7: Performance of Glint and Clamor on sparse logistic regression workloads.

are sparse and only the final weight update, aggregated over the batch, is returned to the driver.

Finally, we evaluate the per-iteration runtime of these implementations on clusters between 5 and 50 nodes with a 1 billion weight parameter vector using only the small dataset of 96 million points, as using the larger dataset is impractical for smaller numbers of nodes and Glint fails to complete on very large datasets. The results are in Figure 7c. As the number of nodes increases, the performance of Clamor degrades due to the weight aggregation on the driver, both due to the increasing network overhead of retrieving weights from workers as well as lock contention on weights.

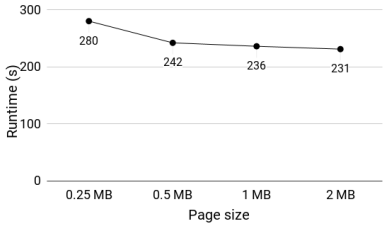


Figure 8: Clamor page size variation for parameter server. Because feautres vary uniformly in their domain, larger page sizes have a higher peer hit rate; however, because the features are sparse, the relative performance gains are negligible after 0.5 MB.

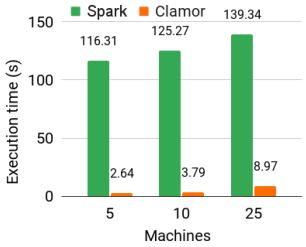


Figure 9: Hash table update scaling for 10 iterations of lookups, with one update in between each iteration.

Performance tuning. Performing this evaluation also highlighted some of the usability advantages of Clamor. We first wrote a naïve Glint implementation that pulled from the parameter server on every read to a weight. This naïve implementation was very slow, with a workload of just 40 thousand data points and 1 million weights taking over 10 minutes per iteration. To speed it up, we implemented optimizations that included manually caching weights on each node, an optimization not required in Clamor because Clamor naturally caches and replicates weights during execution. We also found that data partition sizes had a significant impact on performance of our Glint implementation, so we manually optimized the partition size for each experiment run and compared against the fastest time.

We ran Clamor in these experiments with a page size of 2 MB. In order to determine the page size, we ran an experiment with 1 billion weights and a 2.5 billion point synthetic dataset on a fixed size cluster of 50 workers, varying the page size between 0.25 MB and 2 MB. In Figure 8, we plot these results. A very small page size of 0.25 MB is inefficient because it has a low hit rate for dynamic replication, and above 0.5 MB a larger page size is beneficial but the marginal gain of a larger page size decreases as the features are too sparse for this workload to benefit from additional locality. We set the page size to the optimal 2 MB based on this experiment.

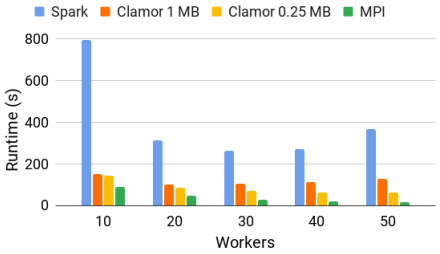


Figure 10: End-to-end runtime for 10 iterations of k-means on 100 GB of data.

7.1.3 Streaming updates In systems such as Spark and Dask, global variables cannot be updated once they have been broadcast to workers [79]. Clamor naturally supports this functionality by allowing the driver to update shared variables, and workers download just the updated pages when they are accessed.

In figure 9, we simulate a streaming updates workload by performing iterations of 10 million Zipf-distributed hash table lookups interspersed with updates to a single key between each iteration. Clamor is up to 44× faster than Spark in this workload, even compared to a Java hash map specialized to `long` pairs, because Spark must re-broadcast a 1 GB variable on each iteration while Clamor workers only update a 1 MB page.

7.2 Batch processing workloads

We additionally evaluate Clamor on a representative batch data-parallel analytics workload that is supported by existing BSP frameworks: k-means clustering. This workload does not perform sparse random access, and our goal is to demonstrate that Clamor's performance is reasonable compared to systems (such as Spark) that primarily target these "dense" access patterns. We compared Clamor with Spark and MPI running k-means for 10 iterations on 100 GB of data on clusters between 10 and 50 machines each running 8 worker processes. This dataset was generated using Gaussian-distributed points around 100 ground truth cluster means with dimension 10 ($k = 10$, $d = 10$). In this experiment, we compared against the same program implemented in Spark and MPI. Spark was configured to have one executor with 8 cores per machine, with the executor allotted the entire machine RAM; MPI was configured to run 8 processes per machine. We run Clamor with both 1 MB and 0.25 MB pages to compare the effect of page sizes on this workload.

The results are in Figure 10. Clamor with a page size of 0.25 MB outperforms the page size of 1 MB by up to 2×, largely because the workload does not make accesses to a global shared variable and thus the main communication bottleneck lies in the aggregation of means (which is a small 10-dimensional vector). With a page size of 0.25 MB, Clamor performs up to 5.8× better than Spark (on the largest cluster size, where

Spark requires 367 seconds and Clamor takes 63 seconds). On the other hand, Clamor is between 1.8× and 3.4× slower than MPI across cluster sizes.

The slowdown relative to MPI came from two places: first, the compute time per iteration in Clamor was 2 seconds slower than MPI's iteration time, which is likely due to a slightly sub-optimal Clamor implementation that assigns clusters to points and then makes a second pass over the cluster assignments to compute means; this can be resolved by optimizing the implementation. Second, aggregates on the driver in Clamor happen in serial rather than in parallel, resulting in higher latency when aggregating the results of work on the cluster. Again, this slowdown is not fundamental, and further implementation work to add parallel or tree-structured aggregation would reduce this aggregation overhead.

7.3 Fault recovery

A key feature of modern BSP frameworks is the ability to recover gracefully from worker failures. In this section, we show that, in spite of the more complex memory model, Clamor retains the fault recovery guarantees of frameworks like Spark.

In Figure 11, we run 10 iterations of k-means on 100 GB of data, on a 25-node cluster, and compare normal execution times to execution times under a failure in the 4th iteration. Each iteration consisted of 250 tasks. The iteration times include the time required to fetch data from S3.

In the first iteration, workers download data from S3 in parallel. Subsequent iterations take about 21s each in the normal case. When a worker fails in iteration 4, the tasks required to reconstruct the data on that worker, including the initial download, are reassigned to other workers. The 112-second execution time for iteration 4 includes the time for workers to complete their assigned tasks from iteration 4 and then download and recompute the data lost from the failed worker. Once the data is reconstructed, iteration time returns to 21s. We can compare these numbers to a comparable experiment in the Spark paper [78]: each iteration in that experiment takes 60s, and the execution time increases to 81s after a failure. The primary reason for this relatively larger slowdown is that Clamor must retrieve lost input data from Amazon S3, while Spark inputs were stored near the executors in a distributed file system. Qualitatively, the recovery behavior of both systems is similar; executors return to normal execution times on the iteration following the failure and recovery.

These results demonstrate that fine-grained, lineage-based recovery is possible even in a relaxed setting in which dependencies are not known in advance, as long as the dependencies can be expressed as a DAG.

7.4 Straggler mitigation

Clamor implements straggler mitigation by launching replicas for tasks within a stage that are lagging behind the median

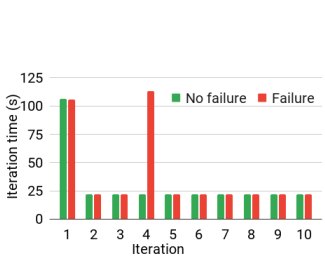


Figure 11: Fault recovery in Clamor. A worker fails in the 4th iteration.

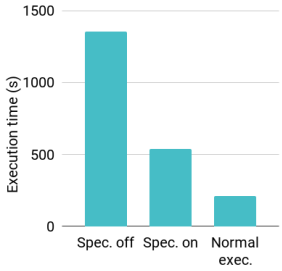


Figure 12: Straggler mitigation for 10 iterations of k-means on 100 GB of data and 10 nodes.

observed time to return a result. This feature is difficult to implement in generic DSM systems, but is possible in Clamor because of the write restrictions in the BSP model. Figure 12 shows Clamor performance with and without speculation enabled in a setting where one node in a 10-node cluster is experiencing heavy CPU contention while running 10 iterations of k-means over 100 GB of data. Each node runs 8 workers, and each iteration has 80 partitions. With no contention, execution takes roughly 3.5 minutes; with this contention and no speculation enabled, execution slows down over 6× to 22 minutes.

With speculation enabled, the overall execution time reduces to roughly 9 minutes (less than 3× slower than executing under no contention). The remaining slowdown comes from two sources: first, the delay in observing that a node is slow and launching replicas, and second, the additional delay incurred from moving data from the slow node to the replicas. Because there are exactly as many partitions as workers, a worker must completely finish executing its assigned task before taking on the work of a straggler, so a slowdown of 2× is inevitable in this experiment.

8 Related Work

Cluster programming models. A number of frameworks in recent years provide users with access to distributed computation via restricted dataflow models. MapReduce [29] and Dryad [37] provide an abstraction for massively parallel and distributed computation over datasets located in stable storage. Other frameworks similarly restrict the dataflow model, for example for graph computation [16, 35, 50] and stream processing [22, 33]. Spark [78] extends the abstractions of MapReduce and Dryad for fault-tolerant computation over in-memory datasets, providing a restricted form of DSM. A number of cluster computing frameworks aim to provide transparent access to distributed computation for non-expert users. Dask [66] provides a distributed interface for computations supported by the Pandas [51] data analysis library.

Ray [55] primarily targets fine-grained, low-latency computation for machine learning workloads. PyWren [41] massively parallelizes work using serverless computation.

Like Clamor, Piccolo [64] offers a MapReduce-like interface for parallel workloads. Clamor allows for transparent global addressing, while Piccolo requires code modifications to use a shared key-value table API; and Piccolo uses global checkpointing for fault tolerance rather than fine-grained lineage. Similarly, Jet [34] provides a distributed key-value store abstraction for streaming workloads, but cannot implement the caching optimizations possible in Clamor with the BSP programming model. Some distributed workloads, such as k-means, have distributed algorithms that may perform better than our BSP implementation but would lose the fault tolerance and caching benefits [15, 32, 39].

Clamor builds on these models, retaining the best of the performance and fault tolerance enabled by coarse-grained data dependencies while enabling fine-grained data access in key use cases.

Distributed shared memory. The Clamor global address space is partly inspired by a long line of research [59] that aims to make the memory of a cluster transparently available to applications, starting with classic work including IVY [46], Munin [17], and TreadMarks [13] and extending to the present [13, 40, 43, 47]. Co-arrays [60] and partitioned global address spaces (PGAS) [16, 23, 25, 27, 28, 31, 45, 74, 75, 80] are examples of DSM abstractions that leverage independence assumptions on data to reduce synchronization overheads. These systems largely aim to support transparent distriubted shared memory for generic programs. In contrast, Clamor focuses on a subset of applications that require fine-grained data access in a restricted programming model, which allows it to implement beneficial features of functional analytics frameworks that are difficult to implement in general DSM. For example, DSM frameworks typically rely on checkpointing-based fault tolerance (if they implement fault tolerance at all), while Clamor's restricted programming model enables fine-grained, lineage-based fault tolerance as well as straggler mitigation.

More recent models of DSM restrict the workloads they support in order to simplify synchronization, programming, and performance optimization. Grappa [58] takes the position that many common data analytics tasks have poor locality, and argues that workload parallelism can be used to hide the communication overhead of such workloads. However, Grappa does not implement fault tolerance as they determine that restarting a job entirely is cheaper for their target workloads, and Grappa cannot easily integrate straggler mitigation. Pocket [44] provides facilities for sharing ephemeral data between serverless tasks. While Pocket enables a form of distributed shared memory, it does not provide a shared address space or transparent random addressing as Clamor does.

Other systems [11, 12, 30, 53] take advantage of increasing network throughput to implement fine-grained remote memory accesses using RDMA. Clamor's functional task model could also be implemented using RDMA and is not fundamentally in conflict with these systems, although functional tasks enable reasoning about data locality that are difficult in general-purpose RDMA-based systems.

Load balancing and replication. Replication is a common problem for load balancing in distributed systems, and systems implement different replication strategies depending on their target workload. Orchestra [26] is a heavily optimized peer-to-peer mechanism to broadcast global variables in Spark, but does not need to support dynamic load balancing as in Clamor. Slicer [10] supports dynamic sharding, but on time scales of hours rather than minutes or seconds. Centrifuge [9] uses consistent hashing, which in Clamor would incur the overhead of copying each page to several replicas in order to load balance even when those pages may not be accessed. Clay [67] solves a complex optimization problem in order to plan database migrations, again based on statistics over time.

9 Conclusion

We have presented Clamor, a system that enables fine-grained remote memory access for large, sparsely-accessed global variables in data-parallel functional analytics workloads. Clamor takes advantage of sparse access patterns and the restricted BSP execution model to make remote memory access efficient by caching hot pages, and tracks fine-grained lineage within global variables for efficient fault tolerance. Overall, Clamor is able to outperform custom systems that use remote key-value stores for fine-grained access and performs on par with frameworks like Spark on traditional data-parallel workloads, making fine-grained, generic global variable access practical for many relevant modern workloads.

10 Acknowledgements

We thank Shoumik Palkar, Dan Ports, and Shivaram Venkataraman for valuable conversations regarding applications of Clamor. Alex Aiken, Peter Kraft, Deepak Narayanan, John Ousterhout, Keshav Santhanam, Keith Winstein, Gina Yuan, and Irene Zhang provided helpful feedback on previous drafts of our work. We also thank our shepherd, Yu Hua, and the anonymous SoCC reviewers for their detailed comments. This research was supported in part by affiliate members and other supporters of the Stanford DAWN project—Ant Financial, Facebook, Google, and VMware—as well as Toyota Research Institute, Cisco, SAP, and the NSF under CAREER grant CNS-1651570. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. Toyota Research Institute ("TRI") provided funds to assist the authors with their research but this article solely reflects the opinions and conclusions of its authors and not TRI or any other Toyota entity.

References

[1] [n.d.]. Distributed XGBoost with Dask. https://xgboost.readthedocs.io/en/latest/tutorials/dask.html.

[2] [n.d.]. Faiss. https://github.com/facebookresearch/faiss.

[3] [n.d.]. Geospatial. https://redislabs.com/redis-best-practices/indexing-patterns/geospatial/.

[4] 2015. Download Criteo 1TB Click Logs dataset. https://ailab.criteo.com/download-criteo-1tb-click-logs-dataset/.

[5] 2017. Using KDTrees in Apache Spark. http://www.trailofpapers.net/2017/01/using-kdtrees-in-apache-spark.html.

[6] 2018. Parameter Server. https://github.com/dask/dask-ml/issues/171.

[7] 2021. OpenStreetMap. https://openstreetmap.org.

[8] 2021. Yelp open dataset. https://www.yelp.com/dataset.

[9] Atul Adya, John Dunagan, and Alec Wolman. 2010. Centrifuge: Integrated Lease Management and Partitioning for Cloud Services.. In *NSDI*, Vol. 10. 1–16.

[10] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, et al. 2016. Slicer: Auto-sharding for datacenter applications. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 739–753.

[11] Marcos K Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, et al. 2018. Remote regions: a simple abstraction for remote memory. In *2018 USENIX Annual Technical Conference (USENIX ATC) 18)*. 775–787.

[12] Marcos K Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. 2017. Remote memory in the age of fast networks. In *Proceedings of the 2017 Symposium on Cloud Computing*. ACM, 121–127.

[13] Cristiana Amza, Alan L Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. 1996. Treadmarks: Shared memory computing on networks of workstations. *Computer* 29, 2 (1996), 18–28.

[14] Carlos Azevedo. 2019. Broadcasting updates on spark jobs. https://stackoverflow.com/q/57860398.

[15] Maria Florina Balcan, Steven Ehrlich, and Yingyu Liang. 2013. Distributed k-means and k-median clustering on general topologies. *arXiv preprint arXiv:1306.0604* (2013).

[16] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2012. Legion: Expressing locality and independence with logical regions. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–11.

[17] John K Bennett, John B Carter, and Willy Zwaenepoel. 1990. *Munin: Distributed shared memory based on type-specific memory coherence*. Vol. 25. ACM.

[18] Jon Louis Bentley. 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM* 18, 9 (1975), 509–517.

[19] Badri Bhaskar. 2016. Scaling machine learning to billions of parameters. https://databricks.com/session/scaling-machine-learning-to-billions-of-parameters.

[20] broadcast-gist [n.d.]. Spark streaming broadcast variable wrapper. https://gist.github.com/mcnamaras/040a362ca8100347e1a6.

[21] Clement Carreau. 2017. How to update a ML model during a spark streaming job without restarting the application? https://stackoverflow.com/q/43387114.

[22] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R Henry, Robert Bradshaw, and Nathan Weizenbaum. 2010. FlumeJava: easy, efficient data-parallel pipelines. In *ACM Sigplan Notices*, Vol. 45. ACM, 363–375.

[23] Barbara Chapman, Tony Curtis, Swaroop Pophale, Stephen Poole, Jeff Kuehn, Chuck Koelbel, and Lauren Smith. 2010. Introducing OpenSHMEM: SHMEM for the PGAS community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*. 1–3.

[24] Moses S Charikar. 2002. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*. 380–388.

[25] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph Von Praun, and Vivek Sarkar. 2005. X10: an object-oriented approach to non-uniform cluster computing. *Acm Sigplan Notices* 40, 10 (2005), 519–538.

[26] Mosharaf Chowdhury, Matei Zaharia, Justin Ma, Michael I Jordan, and Ion Stoica. 2011. Managing data transfers in computer clusters with orchestra. *ACM SIGCOMM Computer Communication Review* 41, 4 (2011), 98–109.

[27] Cristian Coarfa, Yuri Dotsenko, John Mellor-Crummey, François Cantonnet, Tarek El-Ghazawi, Ashrujit Mohanti, Yiyi Yao, and Daniel Chavarría-Miranda. 2005. 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*. 36–47.

[28] Mattias De Wael, Stefan Marr, Bruno De Fraine, Tom Van Cutsem, and Wolfgang De Meuter. 2015. Partitioned global address space languages. *ACM Computing Surveys (CSUR)* 47, 4 (2015), 62.

[29] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[30] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. 2014. FaRM: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. 401–414.

[31] Tarek El-Ghazawi and Lauren Smith. 2006. UPC: unified parallel C. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. 27–es.

[32] Dan Feldman, Melanie Schmidt, and Christian Sohler. 2020. Turning big data into tiny data: Constant-size coresets for k-means, PCA, and projective clustering. *SIAM J. Comput.* 49, 3 (2020), 601–657.

[33] Yuan Yu Michael Isard Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, and Pradeep Kumar Gunda Jon Currey. 2009. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. *Proc. LSDS-IR* 8 (2009).

[34] Can Gencer, Marko Topolnik, Viliam Ďurina, Emin Demirci, Ensar B Kahveci, Ali Gürbüz Ondřej Lukáš, József Bartók, Grzegorz Gierlach, František Hartman, Ufuk Yılmaz, et al. 2021. Hazelcast Jet: Low-latency Stream Processing at the 99.99 th Percentile. *arXiv preprint arXiv:2103.10169* (2021).

[35] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. 2014. Graphx: Graph processing in a distributed dataflow framework. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 599–613.

[36] grpc [n.d.]. gRPC. https://grpc.io/.

[37] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS operating systems review*, Vol. 41. ACM, 59–72.

[38] Rolf Jagerman, Carsten Eickhoff, and Maarten de Rijke. 2017. Computing web-scale topic models using an asynchronous parameter server. In *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 1337–1340.

[39] Ruoming Jin, Anjan Goswami, and Gagan Agrawal. 2006. Fast and exact out-of-core and distributed k-means clustering. *Knowledge and Information Systems* 10, 1 (2006), 17–40.

[40] Kirk Lauritz Johnson. 1996. High-performance all-software distributed shared memory. (1996).

[41] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the cloud: Distributed computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing*. ACM, 445–451.

[42] kdtree [n.d.]. kdtree-rs. `https://github.com/mrhooray/kdtree-rs`.

[43] Pete Keleher, Alan L Cox, and Willy Zwaenepoel. 1992. *Lazy release consistency for software distributed shared memory*. Vol. 20. ACM.

[44] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic ephemeral storage for serverless analytics. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 427–444.

[45] Vivek Kumar, Yili Zheng, Vincent Cavé, Zoran Budimlić, and Vivek Sarkar. 2014. HabaneroUPC++: a Compiler-free PGAS Library. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. 1–10.

[46] Kai Li. 1988. IVY: A Shared Virtual Memory System for Parallel Computing. *ICPP (2)* 88 (1988), 94.

[47] Kai Li and Paul Hudak. 1989. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems (TOCS)* 7, 4 (1989), 321–359.

[48] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. 2014. Scaling distributed machine learning with the parameter server. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 583–598.

[49] Qiping Li. 2016. A Prototype of Parameter Server. `https://issues.apache.org/jira/browse/SPARK-6932`.

[50] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 135–146.

[51] Wes McKinney. 2010. Data Structures for Statistical Computing in Python . In *Proceedings of the 9th Python in Science Conference*. 51 – 56.

[52] Xiangrui Meng. 2014. `https://issues.apache.org/jira/browse/SPARK-4590`

[53] Christopher Mitchell, Yifeng Geng, and Jinyang Li. 2013. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store.. In *USENIX Annual Technical Conference*. 103–114.

[54] Michael Mitzenmacher. 2001. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems* 12, 10 (2001), 1094–1104.

[55] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, William Paul, Michael I Jordan, and Ion Stoica. 2017. Ray: A Distributed Framework for Emerging AI Applications. *arXiv preprint arXiv:1712.05889* (2017).

[56] Dheevatsa Mudigere, Yuchen Hao, Jianyu Huang, Andrew Tulloch, Srinivas Sridharan, Xing Liu, Mustafa Ozdal, Jade Nie, Jongsoo Park, Liang Luo, et al. 2021. High-performance, Distributed Training of Large-scale Deep Learning Recommendation Models. *arXiv preprint arXiv:2104.05158* (2021).

[57] Maxim Naumov, John Kim, Dheevatsa Mudigere, Srinivas Sridharan, Xiaodong Wang, Whitney Zhao, Serhat Yilmaz, Changkyu Kim, Hector Yuen, Mustafa Ozdal, et al. 2020. Deep learning training in facebook data centers: Design of scale-up and scale-out systems. *arXiv preprint arXiv:2003.09518* (2020).

[58] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. 2014. Grappa: A latency-tolerant runtime for large-scale irregular applications. In *International Workshop on Rack-Scale Computing (WRSC w/EuroSys)*.

[59] Bill Nitzberg and Virginia Lo. 1991. Distributed shared memory: A survey of issues and algorithms. *Computer* 24, 8 (1991), 52–60.

[60] Robert W Numrich and John Reid. 1998. Co-Array Fortran for parallel programming. In *ACM Sigplan Fortran Forum*, Vol. 17. ACM New York, NY, USA, 1–31.

[61] Shoumik Palkar, James J Thomas, Anil Shanbhag, Deepak Narayanan, Holger Pirk, Malte Schwarzkopf, Saman Amarasinghe, Matei Zaharia, and Stanford InfoLab. 2017. Weld: A common runtime for high performance data analytics. In *Conference on Innovative Data Systems Research (CIDR)*.

[62] Julien Peloton. 2019. Accelerating Astronomical Discoveries with Apache Spark. `https://databricks.com/session_eu19/accelerating-astronomical-discoveries-with-apache-spark`.

[63] S Plaszczynski, J Peloton, C Arnault, and JE Campagne. 2019. Analysing billion-objects catalogue interactively: Apache Spark for physicists. *Astronomy and Computing* 28 (2019), 100305.

[64] Russell Power and Jinyang Li. 2010. Piccolo: Building Fast, Distributed Programs with Partitioned Tables.. In *OSDI*, Vol. 10. 1–14.

[65] rebroadcast [n.d.]. Broadcast variables can be rebroadcast? `http://apache-spark-user-list.1001560.n3.nabble.com/Broadcast-variables-can-be-rebroadcast-td22908.html`.

[66] Matthew Rocklin. 2015. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th Python in Science Conference*. Citeseer.

[67] Marco Serafini, Rebecca Taft, Aaron J Elmore, Andrew Pavlo, Ashraf Aboulnaga, and Michael Stonebraker. 2016. Clay: Fine-grained adaptive partitioning for general database schemas. *Proceedings of the VLDB Endowment* 10, 4 (2016), 445–456.

[68] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*. Ieee, 1–10.

[69] Andrew Stubbs. 2015. How can I update a broadcast variable in spark streaming? `https://stackoverflow.com/q/33372264`.

[70] Parthiv Sukumar. 2017. How Spark and Redis help derive geographical insights about customers. `https://build.hoteltonight.com/how-spark-and-redis-help-derive-geographical-insights-about-customers-be7e32c1f479`.

[71] Leslie G Valiant. 1990. A bridging model for parallel computation. *Commun. ACM* 33, 8 (1990), 103–111.

[72] Yeshwanth Vijayakumar. 2020. How Adobe Does 2 Million Records Per Second Using Apache Spark! `https://databricks.com/session_na20/how-adobe-does-2-million-records-per-second-using-apache-spark`.

[73] Yifan Wang. 2015. Call broadcast() in each interval for spark streaming programs. `https://issues.apache.org/jira/browse/SPARK-6404`.

[74] Katherine Yelick, Dan Bonachea, Wei-Yu Chen, Phillip Colella, Kaushik Datta, Jason Duell, Susan L Graham, Paul Hargrove, Paul Hilfinger, Parry Husbands, et al. 2007. Productivity and performance using partitioned global address space languages. In *Proceedings of the 2007 international workshop on Parallel symbolic computation*. ACM, 24–32.

[75] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, et al. 1998. Titanium: a high-performance Java dialect. *Concurrency and Computation: Practice and Experience* 10, 11-13 (1998), 825–836.

[76] Reza Zadeh. 2016. Early investigation of parameter server. `https://issues.apache.org/jira/browse/SPARK-4590`.

[77] Reza Zadeh. 2016. Large linear model parallelism via a join and reduceByKey. `https://issues.apache.org/jira/browse/SPARK-6567`.

[78] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2–2.

[79] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *HotCloud*.

[80] Yili Zheng, Amir Kamil, Michael B Driscoll, Hongzhang Shan, and Katherine Yelick. 2014. UPC++: a PGAS extension for C++. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 1105–1114.