BuDDI: A Declarative Bloom Language for CALM Programming

Rolando Garcia
University of California, Berkeley
rogarcia@berkeley.edu

Giulia Guidi
University of California, Berkeley
Lawrence Berkeley National Laboratory
gguidi@berkeley.edu

Abstract

Coordination protocols help programmers of distributed systems reason about the effects of transactions on the state of the system, but they're not cheap. Coordination protocols may involve multiple rounds of communication, which can hurt system responsiveness. There exist many efforts in distributed computing for managing the coordination-performance trade-off. More recent is a line of work that characterizes the class of workloads for which coordination is not necessary for consistency: namely, logically monotonic programs [9]. In this paper, we present a case study of logical monotonicity in workloads typical to computational biology. We leverage the Bloom language to write efficient distributed programs, and compare their performance to equivalent programs written in UPC++, a popular language for writing distributed programs. Additionally, we leverage Bloom's analysis tools to identify points-of-coordination, and use our own experience using Bloom to recommend some higher-level abstractions for users without strong distributed computing backgrounds.

1 Introduction

The rapid rise of cloud computing in recent years has brought exciting new challenges in the area of programming languages. The overarching question is how we can make cloud programming easier, and thus make the cloud accessible to a wider range of applications. The Bloom language is one of the leading efforts in this direction [2]. Bloom is a language for disorderly distributed programming based on the principles of the CALM (consistency as logical monotonicity) theorem to guide coordination. The CALM theorem states that coordination avoidance is possible when programs are associativity, commutative, and idempotent. The ability to implement an application according to these properties would naturally lead to a coordination-free implementation that would take greater advantage of cloud computing.

We validate some of the assumptions of Bloom by asking the following questions: First, how common are logically monotonic programs in the wild? Specifically, in the area of computational biology. And second, does merely knowing where the coordination-points lie in a non-monotonic program help us rewrite the program to move coordination off the critical path or otherwise improve performance? In this work, we borrow an application from computational biology (k-mer counting) and implement it in Bud (Bloom Under Development) and compare such an implementation with a standard UPC++ implementation. In addition, we reflect on the design of the probabilistic data structure *count min sketch* to reach a more space-efficient implementation.

Our work demonstrates that the assumptions of Bloom do in fact hold in practice, and Bloom is suitable for scientific computing. Additionally, we leverage our experience using Bloom to propose some changes and a higher-level abstraction, one that is more use-friendly to programmers without strong distributed computing backgrounds. In sum, the second core contribution of this paper is "BuDDI", an enhanced version of Bud using Distributed Data Independence (DDI). BuDDI hides the challenges of distributed computing such as load balancing, data locality, and fault tolerance from the programmer so that even a novice programmer can take advantage of distributed computing. This is critical as the explosion of data from many domains in recent years makes distributed computing a necessity rather than a commodity. BuDDI uses the concept of global state and achieves the same goal of Bud in terms of coordination with comparable performance, while logically using global tables instead of local tables.

The paper is organized as follows. Section 2 describes the key global table concept in BuDDI and how it influences the design of BuDDI. In Section 3 we describe another key construct, the iterator, while in Section 4 we argue for a new design of the `Nil` concept for cloud programming. Section 5 illustrates in detail the implemen-

tation of the k-mer counting case study as well as the design of the count min sketch, which is a probabilistic data structure used for k-mer counting. Finally, Section 6 summarizes our conclusions and future work.

2 Global Tables in BuDDI

In Bud, each worker can only read from and write to its own tables; communication between workers is explicitly managed through channels. By restricting tables to strictly local visibility, Bud ensures that communication is only explicit when instructed by the developer, and is always asynchronous. That is, a table update never triggers expensive consensus or communication protocols. From our perspective, these design decisions make the Bud programmer responsible for both:

1. The **efficient performance** of distributed computing: load balancing, straggler mitigation, data locality, resource disaggregation, auto-scaling, etc.

2. The **intended behavior** of distributed computing: fault tolerance, fault detection, reliable communication, causal delivery, etc.

By allowing the compiler and runtime to manage the data placement and communication of tuples stored in global tables, we significantly reduce the burden on the programmer. By supporting global tables, we empower the compiler and runtime to manage caching, replication, sharding, and other data allocation decisions. With a global table abstraction, the runtime environment can use online statistics and metadata to reorganize data as needed for latency, throughput, or reliability. In this section we will show that it is possible to provide a global table abstraction without sacrificing the consistency and performance expected from Bloom languages. In this work, we rely heavily on the work of CRDTs (conflict-free replicated data types) for inspiration [11].

2.1 Keep CALM and Merge Lattices

In BuDDI, the global tables are CRDTs [11]. This means that when a programmer inserts a tuple into a global table, the insertion succeeds immediately and the new tuple is shared asynchronously. This ensures that BuDDI workers remain responsive even in the presence of network partitions. The nature of CRDTs ensures that concurrent updates are eventually resolved. A BuDDI programmer statically specifies the CRDT type of global tables, which can be:

- **G-Set**: Grow-only set.
- **2P-Set**: Two-phase set comprised of positive and negative sets, each of which is a `G-Set`.

- **LWW-Set**: Last-writer-wins set represented as a timestamped `2P-Set`.

- **MV-Set**: Multi-value set, a dynamo-esque `2P-Set`.

By statically defining legal operations on global tables in advance, the developer enables the compiler and runtime to avoid unwanted coordination. `G-Sets` are coordination free on non-monotonic queries, and may wait to receive all tuples before computing a non-monotonic query. `2P-Sets` may have to wait to receive all tuples in both the positive and negative sets to service a monotonic read. The same coordination requirement applies when expressing equivalent computation in Bud, such as in the shopping cart example at checkout [2].

The reader should note that tuples in global tables (i.e. CRDTs) are **not necessarily replicated by default**. The decision whether to replicate (or cache or shard) depends on the programmer's service-level goals and is a customizable target for the compiler and runtime to hit.

2.2 Compile-Time Coordination

Whenever a Bud programmer organizes data such that the tuples are hash or range partitioned along the GROUP BY columns, the workers may compute aggregates without communication. We refer to this communication-avoiding coordination strategy as "compile-time coordination" because the programmer implicitly and statically informs every worker that data stored at other workers is irrelevant for the computation of the aggregate. We will next show how the BuDDI compiler can manage data allocation to exploit compile-time coordination automatically in global tables.

Explicit hash or range partitioning of data might be an easy task for Bud programmers, but fixed data partitioning logic prevents the runtime from adapting on the fly to changes in the number of workers, per-worker processing rate, and network conditions. In addition, the distribution of a data set is not always known in advance by the developer. This could lead to serious load balancing problems if the data is skewed. Global tables allow the BuDDI compiler to parse the query to make data distribution decisions and inject switch operators into the IR so that the runtime can dynamically detect and adapt to changes. For example, the data may be initially partitioned in a range, but after a skew is observed, the operators in the IR can switch to round-robin partitioning and inform the system of the need for communication-based coordination. Such adjustments are not easy to express with only local tables and channels, but they follow naturally from the semantics of global tables based on CRDTs.

In Section 5.2, we show a k-mer counting implementation in BuDDI that uses a global table of type `G-Set`, and relies solely on compile-time coordination.

2.3 One-Shot Fixed Point Computation

Consider how a Bud programmer would remove items from a shopping cart:

$$\text{shopping\_cart} := \text{shopping\_cart} - \text{bad\_items} \quad (1)$$

Because `shopping_cart` appears on the left hand side and right hand side of the assignment, this is a recursive query that requires stratification: the query is evaluated repeatedly until a fixed-point is reached [8]. It is possible to statically determine the need for stratification by detecting cycles in the dataflow graph. As we will show in this section, it is not necessary to execute all recursive queries to a fix-point. Some recursive queries, such as the one in Eq. 1, can be evaluated in one-shot.

It is possible to evaluate the recursive query in Eq. 1 in one shot because the same query could be written using two `G-Set`:

$$\text{shopping\_cart} := \text{added\_items} - \text{bad\_items} \quad (2)$$

The query is no longer recursive, and it is now statically possible to determine that the query may be safely evaluated in one-shot, without the need for stratification. In fact, this query is a manual implementation of a `2P-Set`, where wanted items are added to the "pos" set and unwanted items are added to the "neg" set. In BuDDI, the same query would be expressed using a `2P-Set`:

$$
\begin{aligned}
&\text{shopping\_cart} := \text{2P\_Set(`pos', `neg')} \\
&\ldots \qquad\qquad\qquad\qquad\qquad\qquad\qquad (3) \\
&\text{query(shopping\_cart)}
\end{aligned}
$$

If the semantics of a table provide the proper fit for a `2P-Set`, the programmer benefits from the following advantages when entering the table as a `2P-Set`:

1. One-Shot Fixed-Point Computation: The runtime evaluates the contents of the table safely in one step.

2. Async Garbage Collection: The runtime knows that it may safely delete elements from the pos-set iff it deletes the corresponding item from the neg-set.

3. Compile-Time Coordination: The compiler knows that it **can** make data allocation decisions such that tuples in the same group are on the same worker in both the pos-set and the neg-set. In other words, the runtime could check the neg-set only at the local worker without communication and know if the tuple in the pos-set was deleted.

2.4 Zero-Knowledge Overwrite

As our reader probably already knows, the neg-sets of 2P-Set are tombstones. Once a tuple is added to the neg-set, the same tuple can never be added to the 2P-Set again. Such a restriction violates the semantics of the set and makes it difficult or impossible to implement stateful applications. A workaround for achieving stateful execution with CRDTs is to append a universally unique identifier (uuid) to each tuple. This way, an object that has been removed from the 2P-Set can be added back with a new uuid. From now on we will refer to a 2P-Set which models the set semantics as `True-Set`. `True-Set` supports add, remove, add after remove, and update.

An advantage of the uuid approach is that it is possible to add tuples to a `True-Set` without coordination, since the probability of collisions guaranteed by cryptographically secure random number generators is negligible. This quality we call *zero-knowledge insertion*, because a worker may insert tuples into `True-Set` without knowing the content stored in replicas.

However, the uuid approach does not support zero-knowledge deletion or update. To update the contents of an object stored in a `True-Set`, the worker must read the contents of the replicas to learn the uuid of the object. Blindly deleting the object could result in an anomaly where the new tuple is deleted because the messages may be reordered. Consequently, the delete operation must be parameterized by the uuid of the intended object and the request for the uuid must be served synchronously.

Using timestamps instead of uuid allows the support of updates and deletions with zero knowledge (in addition to insertions) for `True-Set`. The `True-Set` implements a variant of Last-Writer-Wins (LWW) or Multi-Value-Set (MV) to control concurrency. For example, a `True-Set` implementing LWW would report the object with the latest timestamp. Outdated versions of the object with old timestamps would be ignored or garbage collected. A worker could delete an object from the `True-Set` without a synchronous read by inserting the object with a later timestamp into the tombstone. If the timestamp of the object in the tombstone is greater than all the timestamps of the versions of that object in the pos-set, then the object is considered deleted.

Clock synchronization is not exactly a coordination-free affair, but such is the cost of sacrificing commutativity in CRDTs and imposing order in the cloud.

3 Iterators

As global tables are to local tables, so are iterators to channels. In this section, we will show that iterators are language constructs that can increase the declarativity of Bloom languages, especially with respect to stream or unstructured data processing.

3.1 Reading Big Files

In early 2019, it was announced that a black hole had been photographed for the first time [1]. Soon after,

images of Dr. Katie Bouman, standing behind several stacks of hard drives, began circulating on Twitter. The data needed to map the black hole weighed a total of 4.5 petabytes. The question is, how can Bloom languages and the cloud in general even read such large files?

Database Management Systems can sort files of any size even with low memory specs. At the highest level, the key is to process the file incrementally, bringing it into memory one chunk at a time, and keeping track of the work that remains to be done. Most of this behavior is provided by the iterator abstraction. The caller invokes next, and the iterator returns a tuple or group of tuples (chunk), for processing. The iterator ensures that the next chunk of work to be processed by the caller.

One approach for managing chunk size is to use as many chunks as there are workers—to enable high-throughput reads without requiring coordination. Unfortunately, this approach is vulnerable to stragglers and cannot adapt on-the-fly to workers that enter or leave the workpool by default. A workaround is to use a strategy similar to that used in MapReduce to allow the number of chunks to be much larger than the number of workers [6]. Chunks are assigned to workers for delivery as they complete work. Faster workers do more work than slower workers. When new workers join, they can be assigned work at any time. If a worker fails or leaves the group, data they have completed is not reassigned, and data they have not completed is returned to the pool of pending work.

3.2 Exactly-Once Semantics

At least once delivery under set semantics is exactly-once semantics, because sets filter duplicates. But as we discussed in Section 2.4, we have reason to give the same entity more than one uuid, to enable re-insertion after deletion from a 2P-Set.

The solution then is to give each data unit (e.g. tuple) two identifiers: one uuid identifies the token (e.g. the byte-offset from the start of the file); the second uuid identifies the *use* of the token. Thus, an object that gets re-inserted to a 2P-Set would have the same token-uuid as the one that is in the tombstone, but it would have a different *use*-uuid. We believe it's possible to statically assign token-uuids to units of data without coordination, such as when we stamp each k-mer with its offset from the start of the file. Use-uuids are assigned at runtime and may use either uuids or timestamps, as discussed in Section 2.4,

With a token-uuid and use-uuid, it's possible to exploit at-least-once delivery on CRDT True-Sets to achieve exactly-once semantics. All this with the benefits of auto-scaling to workers entering or leaving the work group, or workers speeding up and slowing down.

4 Nil In The Cloud

In developing BuDDI and implementing our case study in both BuDDI and Bud in the following section, the question arises as to how the programmer would interpret the vairable Nil. In particular, the question is whether Nil tells us that the value does not exist for all queries and thus requires global coordination, or whether it tells us that the local worker is unable to find it. The latter behavior is more appropriate for monotonic programming; although it might introduce anomalies, it can maintain responsiveness during a network partition.

One could argue that the Nil concept is an inherited appendage from the monolithic computer era. Here we argue that for cloud programming we should rethink the Nil concept as having not one but two values: either DNE ("does not exist", which is a global assertion) or IDK ("I don't know", which is a local assertion). If the network is healthy and the data has been partitioned so that coordination is free or inexpensive (Section 2.2), the stronger property is considered and DNE is returned. Otherwise, IDK is returned. We believe that DNE and IDK are sufficiently well understood by programmers to be useful boolean primitives. For example, given a DNE, you could execute a block of code, whereas given a IDK, you would perform some other operation, such as retry, crash, sleep, or guess and apologize.

In Section 5 we will see how the Nil concept is not currently implemented in Bud and how its implementation would give the programmer more flexibility.

5 Case Study: K-mer Counter

A common computation in Computational Biology is to count the frequency of fixed length sequences known as k-mers. The k-mer histogram that we obtain as a result is valuable for understanding the distribution of biological subsequences and for profiling genomic and metagenomic data. For example, we may be interested in subsequences that occur within a certain interval or above a certain threshold.

The k-mer counting step often takes a large part of the total application runtime and it is a key computation within popular tools for taxonomic mapping [12], metagenome classification [4], genome assembly [10].

The k-mer count is arguably a simple calculation, but its efficient implementation is anything but simple. This problem has received much attention as an important target for shared memory parallelism. As data sets grow faster and faster, distributed memory parallelization is becoming more and more important. Nevertheless, the irregularity of the input data makes k-mer counting a difficult problem for distributed memory parallelization. In particular, the k-mer distribution over biological input

data is not fixed and can only be determined at runtime.

5.1 Implementation

In this work, we implement a toy version of the k-mer counting kernel in Bud and compare it to a UPC++ implementation, which resembles a more standard way of implementing this computation. UPC++, similarly to Bud, makes use of asynchronous communication. From a high level perspective, the main difference between the two codes is that the implementation based on Bud must follow Bloom's rules of monotony and idempotency.

UPC++ UPC++ [3] is a C++ library supporting Partitioned Global Address Space (PGAS) programming. UPC++ is suited for implementing complex distributed data structures where communication is irregular or fine grained. The main abstractions in UPC++ are: (a) global pointer to improve locality, (b) asynchronous remote procedure call (RPC), and (c) futures.

Listing 1 illustrates the key implementation for the UPC++ version of k-mer counter. The k-mer counter materializes as a distributed hash table divided by keys, where k-mers are keys and their frequencies are values. The program first parses the input data in parallel, so that each processor has a part of the input sequences. Each processor parses its local sequences in k-mers and determines which k-mers remain local and which must be sent to another processor based on a hash function. When a processor receives incoming data from other processors, it updates its local partition of the k-mer hash table by incrementing the frequency corresponding to the received k-mers. A given k-mer is counted by one processor and only one processor.

Communication in UPC++ is asynchronous, and in our implementation we use a remote procedure call to update values in the hashmap that do not belong to the local processor. A remote procedure call causes a procedure to be executed in a different address space, encoded like a normal procedure call, without the programmer explicitly coding the details for the remote interaction.

Listing 1: UPC++ hashmap for k-mer counting.

```
class DistrMap
{
    /* <kmer, count> map */
    using dobj_map_t =
        dist_object<unordered_map<string,
        int>>;

    /* build empty map */
    dobj_map_t local_map{{}};

    /* compute owner for the given key */
    int get_target_rank(const string &key)
```

```
    {
        return hash<string>{}(key) % rank_n();
    };

    void local_update(unordered_map<string,
        int> &lmap, const string &key)
    {
        auto it = lmap.find(key);
        if(it != lmap.end()) it->second++;
    };

    future<> populate(const string &key)
    {
        /* send rpc to the owner rank */
        return rpc(get_target_rank(key),
        [](dobj_map_t &lmap, const string &key)
        {
            /* check if key in local map */
            if(lmap->count(key) == 0)
                (*lmap)[key] = 1;
            /* update local value */
            else local_update(*lmap, key);
        }, local_map, key);
    };
};
```

Bud Listings 2–4 show different implementations that we developed for our case study in Bud. In particular, the implementation in Listing 3 uses domain-specific knowledge to optimize the memory footprint.

Counting subsequences is monotone by nature, because once you have seen a k-mer instance, the k-mer count can only increase or remain unchanged. One might first think to use a lmap as a local partition, where k-mers are the keys and values are lmax lattices, since the k-mer count can only be incremented and lmax is defined as an integer that can only increment. However, the default merge function of lmax is in fact the maximum between two entries, not the sum as we would wish. It is not possible to override the merge function or create a custom lattice that uses sum as the merge function because sum is not idempotent.

The computation can be made idempotent by replacing lmax with lset in the local partition. In this implementation, each k-mer *instance* in the local partition is assigned a unique identifier. If ATAG occurs twice in the input data set, the corresponding lset in the local lmap stores two unique identifiers. Once the computation is complete, we perform a local count of lset and display the k-mer frequency for each k-mer in the input data set. In this work, we refer to this implementation as Implementation A (Listing 2).

Listing 2: Bud k-mer counting: Implementation A. This implementation is not memory efficient because it

stores as many uuid as k-mer instances.

```ruby
# parse file and store sequences in an array
class DNA
  def readseq(myinput)
    sequences = Array.new
    ParseFasta::SeqFile.open(myinput).each_record
        do |rec|
      sequences.push(rec.seq.upcase)
    end
  end

  def kmers(sequences, k)

    subsequences = Hash.new
    for item in sequences do
        i = 0
        while i < item.length-k+1 do
            subsequences[SecureRandom.uuid] =
                item[i..k+i-1]
            i += 1
        end
    end

    karray = Array.new
    karray = subsequences.to_a
  end
end

class CountKmer
    include Bud
    state do
      scratch :kmer,    [:uuid, :seq]
      scratch :receive, [:seq, :uuid]
      scratch :leave,   [:seq, :uuid, :owner]
      lmap  :local
      lmap  :incoming
      lmap  :counter
      table :result
      channel :msg, [:seq, :uuid, :@addr]
    end

    bloom :ownership do
      # ip port based on sequence
      owner  = hash(t.seq) % worldsize
      leave <= kmer {|t| [t.seq, t.uuid, owner]}

      msg <~ leave do |seq, uuid, owner|
        [seq, uuid, owner]
      end
    end

    # update local set with incoming data
    bloom :insert do
      receive <= msg do |seq, uuid, owner|
        if owner == ip_port
          [seq, uuid]
        end
      end
```

```ruby
      # merge incoming kmer into local map
      incoming <= receive {|t| {t.seq =>
          Bud::SetLattice.new([t.uuid])}}
      local    <= incoming
      counter  <= {ip_port => local}
    end

    bloom :count do
      result <= counter.to_collection do
          |owner, m|
          [owner, m.to_collection do |k, v|
            [k, v.size]
          end
          ]
      end
    end
end
```

Implementation A is monotone and idempotent, however, it poses some concern about the memory consumption. In a medium-sized genome data set, we have billions of k-mers and some of these can occur hundreds of times in the input data set. Therefore, storing an identifier for each k-mer instance looks extremely inefficient from a memory standpoint. Fortunately, domain-specific knowledge helps us in the design of our Implementation B (Listing 3). In general a user is only interested in subsequences that occur within a certain interval or below/above a certain threshold. This information can be elegantly integrated into the Bud implementation using custom lattices.

Listing 3: Bud k-mer counting: Implementation B. This implementation uses domain-specific knowledge to save memory and stores uuid only up to a certain threshold. Implementation B uses a custom lattice Bud::BuddySetLattice which is a modified version of the standard Bud::SetLattice reported in Listing 4.

```ruby
class CountKmer
    include Bud
    [...]
    # update local set with incoming data
    bloom :insert do
      [..]
      # merge incoming kmer into local map
      incoming <= receive {|t| {t.seq =>
          Bud::BuddySetLattice.new([t.uuid])}}
      local    <+ incoming
      counter  <+ {ip_port => local}
    end
    [...]
end
```

For example, if we are only interested in subsequences that occur above a certain threshold, we can create a cus-

tom `lset` where our merge function computes the union of two sets only if the reference set has not yet reached the custom threshold (Listing 4). The threshold is commonly much smaller than the maximum frequency of high-frequency k-mers, making Implementation B much more memory efficient than Implementation A.

Listing 4: Bud::BuddySetLattice is identical to the standard `lset` except for its `merge` function that uses domain-specific knowledge to avoid wasting memory.

```
THRESHOLD = 10
class Bud::BuddySetLattice < Bud::Lattice
  wrapper_name :buddylset
    [...]
    def merge(i)
      if @v.size < THRESHOLD
          wrap_unsafe(@v | i.reveal)
      else
          wrap_unsafe(@v)
      end
    end
    [...]
end
```

Before we approached the solution with a custom lattice, we tried an implementation with native lattices. In Listing 5 we use a naive `lset` lattice as the value in the `lmap` instead of the custom `Buddylset` in Listing 4. In this case we want to add the received k-mer to the local `incoming` lmap only if we have not yet reached the threshold for this k-mer key. The first error we encountered in implementing this version was a *stratification* error, because when we merged `incoming` into `local` we used <= instead of <+. In this case, `incoming` merges with `local`, which in turn is used to calculate `incoming`. To fix this error, we had to defer the merging of `incoming` into `local` with <+ until the next time step. The stratification problem is solved at this point, but we have the problem that `local` may not contain the key we are looking up, and this caused a runtime error complaining that it cannot find the key. Here, we wonder if it would be possible to return a `Nil` value when a key is missing, rather than causing a runtime error. In Section 4, we briefly discussed how introducing two values (DNE and IDK) for the `Nil` concept would enable us to implement this implementation of k-mer counting without using a custom lattice. In particular, returning a local `IDK` would be sufficient in this case, since each k-mer (key) exists on one and only one process.

Listing 5: Bud k-mer counting: `Implementation B` using only native types and related error.

```
THRESHOLD = 10
class CountKmer
    include Bud
```

```
    [...]
    # update local set with incoming data
    bloom :insert do
      [..]
      # merge incoming kmer into local map
      incoming <= receive {|t| {t.seq =>
          Bud::SetLattice.new([t.uuid])} if
          (local.at(t.seq).size < THRESHOLD)}
      local     <+ incoming
      counter   <+ {ip_port => local}
    end
    [...]
end
```

5.2 BuDDI

In this section, we demonstrate a hypothetical implementation of the k-mer Counting algorithm in BuDDI. We use this example to demonstrate the use of G-Sets (a CRDT, or global table), and iterators (e.g. the call to `open` the file). The take-away from this subsection is that (i) BuDDI programs are simple for Bloom programmers to write and understand, (ii) that their semantics are clear and unambiguous, and (ii) that the language is sufficiently declarative to allow for acceptable performance with the aid of an intelligent compiler working in concert with a specialized runtime. We leave both as future work. For reference on achievable performance and semantics, the reader may want to refer back to Sections 2 and 3.

The BuDDI programmer writes the k-mer counting program as a SQL query with aggregation over a global table. In this case, the global table is called `kmers`, and it is implemented by a G-Set (or grow-only set). Aggregation, when exact, is a non-monotonic operation, but the runtime may begin processing the query on-the-fly, and either report intermediate results or withhold them until the end as preferred by the user.

BuDDI executions allow for workers to join or leave the worker pool on-the-fly. New workers are added with the `register_worker` method which monotonically inserts the worker's metadata into a channel. BuDDI also uses an iterator (e.g. `open` in `read_dna_file`) to read the potentially massive file in parallel. As we discussed in Section 3. The iterator is able to estimate network conditions and worker performance by the rate at which workers request data from the iterator. Thus, with the iterator, BuDDI is able to detect failures (a worker stops requesting data) and adapt on-the-fly to slowdowns.

The BuDDI compiler statically recognizes that this workload is hash-partitionable on the `seq` column of the `kmers` table, and may rendezvouz tuples with matching token- uuids (no use-uuid required, since we are inserting into a G-Set), into the same worker, so no communication is required to compute an aggregate. If the

Listing 6: BuDDI k-mer counting.

```python
from buddy import Channel, GSet, View
from buddy import uuid, init

worker = Channel(key='@addr', value='file_uri')

kmers = GSet(key='uuid', value='seq')
result = View(query="""
                SELECT seq, COUNT(*)
                FROM kmers
                GROUP BY seq
                """)

def read_dna_file(kmer_stream =
    open(worker.file_uri, req_id=uuid(),
    addr=worker.addr, mode='char[4]')):
    global kmers
    kmers += {(uuid(), kmer) for kmer in
        kmer_stream}

@init
def register_worker(address: 'str',
    dna_file_uri: str):
    global worker
    worker += {(address, dna_file_uri)}
```

runtime does not replicate the data stored in the `kmers` CRDTs, then this execution and schedule achieves comparable performance to the hand-tuned Bud implementation, with all the additional benefits of declarativity.

5.3 Count Min Sketch Design

Count min sketch [5] is a probabilistic data structure that serves as a frequency table of events in a stream of data. It is usually implemented as a data structure like a matrix, where h rows represent your hash functions and m columns represent the ranges, where m is smaller than the number of k-mers because it is a sublinear data structure. When inserting a k-mer x, this is hashed with the h different hash functions and the counter in the corresponding column of the matrix calculated as hashed value module m is incremented. And then we take the minimum count over the h cells in the matrix where this k-mer was hashed. In practice, m is related to the actual number of k-mers, and usually people use the HyperLogLog algorithm [7] to estimate the cardinality of the k-mer and chose m accordingly.

The implementation of a count-min sketch is more complicated than that of a regular k-mer counting algorithm, since the data distribution is not straightforward and the data access pattern is not contiguous. In this paper, we present two possible high-level designs and describe their current shortcomings. The first design con-

sists of the composition of standard Bloom lattices while the second design is based on a custom lattice that is more similar to the original data structure.

Using standard lattices, we can use a `lset` of size m, where each entry is a `lmap` of size h, where the key is the hash function id and the value is a `lset` of unique identifiers of k-mers; when merged, the `lset` increases its size by adding the corresponding identifier (uuid) to the entry $[m]\rightarrow[h]\rightarrow[\text{uuid}]$. In this case, the data distribution could be based on the m ranges, where for P processes each process has m/P entries to take care of. This distribution is relatively easy to implement, but the irregular data access pattern makes it more complicated to implement the `min` operator, because the h entries on which we want to perform the operation could potentially belong to h different processes. This data structure and distribution requires cross-process coordination, and the communication pattern resembles an MPI `Reduce` or `Allreduce` collective communication. The implementation of collective communication in BUD will remain as future work. This first design trades coordination for memory usage, since each process has only one partition of the entire data structure, but may require coordination of all processes to reveal the final result.

A second design, based on a custom lattice, results in the opposite compromise: memory usage to reduce coordination. The custom lattice is very similar to the original data structure, i.e. a $h \times m$ matrix $\mathbf{A}$ (e.g. two Ruby Array structures), where each $\mathbf{A}(i,j)$ entry is a k-mer unique identifier `lset`. In this case, the distribution may be the same as we saw in the regular k-mer counting algorithm, i.e. each k-mer is hashed to a single process, and this process calculates the h hash function on this k-mer and updates the corresponding matrix entries. Here each process has a local copy of the entire matrix, and the merge operation is the `lset` merge operation applied to each of the matrix entries. This means that consistency is ensured by an analysis at application level, which resolves write conflicts with a shared state. This design allows easier distribution and less coordination, but the memory consumption is higher because each process has a local copy of the entire data structure.

The use of identical copies of the matrix on each process justifies the design and use of BuDDI and its logically global tables. In BuDDI, the runtime may do replication or partitioning of data structures, but this is abstracted for the programmer. In addition, BuDDI gives us access to the application-side code that we can use to learn the semantics of the application layer. The application semantics may give BuDDI a way to resolve an apparent conflict, and can therefore allow BuDDI to reach good performance using weak consistency.

6 Conclusion and Future Work

In this paper, we performed a case study of a representative HPC workload to evaluate the fitness of CALM programming for computational science. In particular, we compared a Bud implementation with a UPC++ implementation and found that both are equally expressive. Although a detailed performance comparison remains a future work, Bud provides enough low-level control to assume similar scaling behavior as UPC++.

However, Bud's low-level characteristics complicate distributed programming by offloading performance and correctness considerations to the user. Our case study motivated the design of BuDDI, a *more* declarative Bloom language that provides Distributed Data Independence. Our declarative abstractions rely on CRDTs and Iterators to maintain acceptable consistency and performance with the many benefits of declarative logic programming. Our hope is that the design of BuDDI will motivate our readers to work with us to implement the compiler and runtime.

Contributions

RG worked primarily on the design of BuDDI, and discussion of Global Tables and Iterators. GG worked primarily on the case study, implementation, and related discussion of k-mer counting in Bud and UPC++. RG and GG contributed equally to the writing of the report.

This paper is submitted in fulfillment of the requirements of UC Berkeley's CS294 graduate seminar: "Programming the Cloud." Taught by Professor Hellerstein and Dr. Milano.

References

[1] AKIYAMA, K., ALBERDI, A., ALEF, W., ASADA, K., AZULAY, R., BACZKO, A.-K., BALL, D., BALOKOVIĆ, M., BARRETT, J., BINTLEY, D., ET AL. First m87 event horizon telescope results. iv. imaging the central supermassive black hole. *The Astrophysical Journal Letters 875*, 1 (2019), L4.

[2] ALVARO, P., CONWAY, N., HELLERSTEIN, J. M., AND MARCZAK, W. R. Consistency analysis in bloom: a calm and collected approach. In *CIDR* (2011), Citeseer, pp. 249–260.

[3] BACHAN, J., BONACHEA, D., HARGROVE, P. H., HOFMEYR, S., JACQUELIN, M., KAMIL, A., VAN STRAALEN, B., AND BADEN, S. B. The upc++ pgas library for exascale computing. In *Proceedings of the Second Annual PGAS Applications Workshop* (2017), pp. 1–4.

[4] BENOIT, G., PETERLONGO, P., MARIADASSOU, M., DREZEN, E., SCHBATH, S., LAVENIER, D., AND LEMAITRE, C. Multiple comparative metagenomics using multiset k-mer counting. *PeerJ Computer Science 2* (2016), e94.

[5] CORMODE, G., AND MUTHUKRISHNAN, S. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms 55*, 1 (2005), 58–75.

[6] DEAN, J., AND GHEMAWAT, S. Mapreduce: a flexible data processing tool. *Communications of the ACM 53*, 1 (2010), 72–77.

[7] FLAJOLET, P., FUSY, É., GANDOUET, O., AND MEUNIER, F. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm.

[8] GREEN, T. J., HUANG, S. S., LOO, B. T., ZHOU, W., ET AL. *Datalog and recursive query processing*. Now Publishers, 2013.

[9] HELLERSTEIN, J. M., AND ALVARO, P. Keeping calm: when distributed consistency is easy. *Communications of the ACM 63*, 9 (2020), 72–81.

[10] LI, D., LIU, C.-M., LUO, R., SADAKANE, K., AND LAM, T.-W. Megahit: an ultra-fast single-node solution for large and complex metagenomics assembly via succinct de bruijn graph. *Bioinformatics 31*, 10 (2015), 1674–1676.

[11] SHAPIRO, M., PREGUIÇA, N., BAQUERO, C., AND ZAWIRSKI, M. Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems* (2011), Springer, pp. 386–400.

[12] WOOD, D. E., AND SALZBERG, S. L. Kraken: ultrafast metagenomic sequence classification using exact alignments. *Genome biology 15*, 3 (2014), 1–12.