Parallel Block-Delayed Sequences

Sam Westrick
Carnegie Mellon University
Pittsburgh, PA, USA
swestric@cs.cmu.edu

Mike Rainey
Carnegie Mellon University
Pittsburgh, PA, USA
me@mike-rainey.site

Daniel Anderson
Carnegie Mellon University
Pittsburgh, PA, USA
dlanders@cs.cmu.edu

Guy E. Blelloch
Carnegie Mellon University
Pittsburgh, PA, USA
guyb@cs.cmu.edu

Abstract

Programming languages using functions on collections of values, such as map, reduce, scan and filter, have been used for over fifty years. Such collections have proven to be particularly useful in the context of parallelism because such functions are naturally parallel. However, if implemented naively they lead to the generation of temporary intermediate collections that can significantly increase memory usage and runtime. To avoid this pitfall, many approaches use "fusion" to combine operations and avoid temporary results. However, most of these approaches involve significant changes to a compiler and are limited to a small set of functions, such as maps and reduces.

In this paper we present a library-based approach that fuses widely used operations such as scans, filters, and flattens. In conjunction with existing techniques, this covers most of the common operations on collections. Our approach is based on a novel technique which parallelizes over blocks, with streams within each block. We demonstrate the approach by implementing libraries targeting multicore parallelism in two languages: Parallel ML and C++, which have very different semantics and compilers. To help users understand when to use the approach, we define a cost semantics that indicates when fusion occurs and how it reduces memory allocations. We present experimental results for a dozen benchmarks that demonstrate significant reductions in both time and space. In most cases the approach generates code that is near optimal for the machines it is running on.

CCS Concepts: • **Software and its engineering** → **Parallel programming languages**; *Functional languages*; • **Theory of computation** → *Parallel algorithms.*

This work is licensed under a Creative Commons Attribution International 4.0 License.
PPoPP '22, April 2–6, 2022, Seoul, Republic of Korea
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9204-4/22/04.
https://doi.org/10.1145/3503221.3508434

Keywords: parallel programming, fusion, collections, functional programming

1 Introduction

Collection-oriented programming is a style of programming in which programs use operations over collections of values, such as map, reduce, filter, and scan. Languages supporting this style date back to 1960s with APL (arrays) [18], SETL (sets and maps) [29], Codd's relational algebra (relations) [11] and FP (sequences) [3]. These languages allowed a particularly simple and elegant way to work with collections. With the advent of highly parallel machines in the mid 80s, there was a significant increase in interest in this style of programming. The observation is that by raising the level of abstraction, sequential loops go away, and code often becomes inherently parallel. Furthermore, working with collections promotes a functional style of programming, and hence mostly avoids mutation and, in the context of parallelism, the potential dangerous data races they cause. Early such *data parallel* languages include CM-Lisp [19], C* [28], and Nesl [4]. Later ones used in a distributed setting include map-reduce [14] and Spark [37].

It was quickly noted, however, that collections can incur large overheads due to the generation of intermediate results. For example, a map squaring every element of a vector, followed by a reduce summing the results would naïvely generate an intermediate vector of the products before summing them. A loop, on the other hand, would multiply and add as it went along. The generation of this intermediate vector wastes not only space but also time, due to additional reads and writes in situations where memory bandwidth is often the bottleneck. This problem of avoiding intermediate results has been studied extensively since the 1970s [1, 35] and is often referred to as "loop-fusion", just "fusion", or originally "jamming". Fusion has been applied to data-parallel languages since the start of the 90s with dozens of papers on the topic (e.g., [9, 12, 13, 17, 20–22, 24, 26, 31]). Most of these techniques rely on compiler transformations.

Interestingly, however, Keller et al. [20] were able to show that by taking advantage of standard compiler optimizations, fusion can be implemented efficiently for sequences as a

library interface (Repa). Their approach is based on the idea that operations on sequences, such as map, can be "delayed" by representing a sequence as a function from index to value. In the map-reduce example, the map can generate such a function at almost no cost, and the reduce can then call this function to generate values for all indices. Their observation is that a reasonable compiler can inline the function, avoiding the cost of a function call at each index, and generating near optimal code without a slew of special-purpose compiler optimizations. We refer to this as *index fusion.* Keller et al.'s approach, however, only supported a limited set of transformations, which (roughly speaking) consists of maps followed by maps, or maps followed by a reduce.

In this paper we present a technique that, like Repa, can be supported at the library level, but is significantly more general, allowing fusion in many more cases. In particular it allows fusion of scan operations with both maps before and after the scan, or even a scan followed by a scan, filter, or reduce. It furthermore allows fusion when a nested sequence is flattened before or after a map, scan, or filter. This covers the majority of common functions on collections.

Our approach uses two forms of delayed sequences. The first, called a ***random-access delayed (RAD)*** sequence, represents a sequence as a function from index to value, similar to Repa. The second, called a ***block-iterable delayed (BID)*** sequence, splits a sequence into many equal-sized blocks, where each block is a delayed stream of elements that can be retrieved sequentially. RADs allow random-access and are generated by operations such as map and tabulate. BIDs do not allow random-access, and are generated by scan, filter, or a flatten, as well as by a map or zip that is itself given a BID as input. The idea of a BID sequence is new, whether supported by a compiler or as a library as we do.

By achieving fusion optimizations entirely within a library, our approach requires no changes to the compiler, and is easy to implement in a broad set of languages. To demonstrate portability, we implement libraries for both C++ and Parallel ML, languages with very different semantics and compilers. To help users understand when there is a benefit from the approach, and in some cases potential loss, we define a cost semantics that captures the work, span (critical path length, a.k.a., depth), and temporary space generated by different computations. The space is a good prediction of performance due to memory bandwidth limitations.

To evaluate our delaying technique, we study the performance of a collection of benchmarks on a 72-core machine, with each benchmark implemented in both C++ and Parallel ML, using the MPL [2, 36] compiler for Parallel ML. We compare the delayed versions against programs that use a fast, non-delayed implementation of sequences. At scale, our delayed versions achieve speedups that are up to 5.3x faster in MPL, and 19x in C++. The speedup is in most cases is at least 2x. Moreover, we observe a 3x improvement in maximum space usage in most cases, and up to 250x in both

```
type α seq              // sequences with elements of type α
val empty: α seq                       // written ⟨⟩
val length: α seq → int                // written |a|
val sub: α seq → int → α               // written a[i]
val tabulate: int → (int → α) → α seq
val map: (α → β) → α seq → β seq
val zip: α seq × β seq → (α × β) seq
val reduce: (α × α → α) → α → α seq → α
val scan: (α × α → α) → α → α seq → α seq × α
val filter: (α → bool) → α seq → α seq
val filterOp†: (α → β option) → α seq → β seq
val flatten: α seq seq → α seq
```

†Also known as mapMaybe (Haskell) and mapPartial (SML).

Figure 1. Sequence interface: common operations

MPL and C++. Given our delayed sequence library, we could attain these speedups with only a modest effort for each benchmark, often with no additional implementation effort.

Our contributions include:

1. An approach based on blocking that allows agressive loop fusion for operations on sequences at the library level, covering several more cases than previously covered, and supporting nested parallelism.
2. Implementations in both C++ and Parallel ML.
3. A cost semantics that enables users to reason about the performance of fused operations in terms of work, span, and allocations (memory footprint).
4. An experimental evaluation on a collection of benchmarks demonstrating the effectiveness of our approach.

The approach described in this paper has already been integrated in the Parlay Library [5] and has been applied to improve the performance of several of the benchmarks in PBBS [30], including breadth-first search (BFS), inverted indices, and ray-triangle intersection.

2 Background

Our goal is to provide an interface like that shown in Figure 1. We present code in ML-like syntax, but we support the same functions in C++ via templates and overloading. The interface includes standard functions on sequences, similar to those found in a variety of collection-oriented languages (e.g. [3, 4, 8, 17, 18, 20, 37]).

Stream fusion. The idea of *stream fusion* is to create the elements of a list on demand starting at the front [12, 16, 24, 32, 34]. In a functional setting, a stream is typically implemented as a function S returning a pair (x, S'), where x is the first element and S' is a stream for the rest of the elements. When a function such as map f S is applied to the stream S, it can return a new stream such that each request for an element will first request an element from S and then apply f to it, returning the result and the rest of the stream. In an imperative setting, a stream can be implemented with

side effects as an iterator (where incrementing the iterator will evaluate the next element). Stream functions such as map or reduce can be composed. For example, a map followed by a reduce would never instantiate the intermediate list, only instantiating one element at a time, requiring $O(1)$ additional memory beyond the input to run.

Collection-oriented programming with stream fusion has gained recent popularity in the programming community, as demonstrated for example by the C++20 ranges library [27]. It promotes collection-oriented programming by supplying a variety of generic algorithms for ranges of elements in the form of so-called *view adapters*. What makes them interesting for us is that they are implemented using stream fusion.[1] The library allows operations such as maps, filters, and flattens to be composed and fused, and is entirely library-based, requiring no language extensions or specialized compiler support. However, it is designed for sequential computation, and does not support parallelism, except in easy cases. Similar tools exist in other languages, such as the `java.util.stream` library introduced in Java 8.

To make stream-fusion efficient requires the compiler to get rid of (possibly multiple) function calls on each iteration. Several special purpose compiler techniques have been suggested, and many modern compilers can do it without any special purpose techniques. Indeed we use stream fusion as a component and rely completely on off-the-shelf compilers for Parallel ML (MPL) and C++ (GCC).

Index fusion. The idea of *index fusion* is to create elements of a random-access array on demand by supplying the index [20]. Essentially, a delayed array is a function from index to value. When a function such as `map` f A is applied to a delayed array A it can return another delayed array that, given index i, evaluates and returns $f(A[i])$. The delayed map does almost no work—it just needs to create a single new function (typically removed by the compiler). As with stream fusion, functions such as map and reduce can be composed, and no intermediate results will be created. For example, in `reduce (+) (map` f A), the map would do almost no work, and the reduce would sum up the elements and evaluate f on each element when that index is requested. Also, as with stream fusion, for efficiency it is important to remove the function calls (e.g., inlining f and the addition functions into the compiled reduce). As noted by the Repa system [20], the Haskell (GHC) compiler can already do this in most cases. We have verified that the MPL and GCC compilers also can handle it.

Combining stream and index fusion. Stream and index fusion have their tradeoffs. Index fusion does not support functions where the result at an index depends on results at earlier indices. For example, in the scan operation, accessing the ith element requires evaluating all previous elements. Stream fusion can easily support scans since it has seen all previous elements and can keep a running sum. On the other hand, stream fusion does not support random access and is therefore not useful by itself for parallelism, while index fusion supports random access and works well in a parallel context. This suggests that a combination of the two could be powerful.

[1] Technically, they are required to be $O(1)$-movable, and $O(1)$ copyable if they are copyable, which effectively forces them to use stream fusion.

2.1 Stream-of-blocks

Previous work has shown how to use variants of stream fusion with parallelism [17, 24] using the *stream-of-blocks* approach. The idea is to break the stream into blocks of fixed length such that requesting the next "element" would instantiate a whole block and return it. Parallelism is then exploited within blocks: for example, for `map` f A, we could apply f to each element of a single block in parallel. This approach works well with fine-grained SIMD parallelism, such as with GPUs or vectorization (where the block-size could correspond to the size of a hardware vector). It is, however, not well suited for coarse-grained parallelism where blocks would have to be gigantic to overcome the cost of spawning tasks to run in parallel. In particular, our experiments (Section 6) show that it is hard to get any improvements with streams of blocks over non-fused operations on a multi-chip processor because of the high cost of synchronization. This work also does not support flatten.

2.2 Block-based implementations

Our approach relies on standard block-based implementations of reduce, scan, flatten, and filter. These are commonly used in practice (including in ParlayLib [5] and Parallel ML [2, 36]) since blocks are a natural mechanism for granularity control and load balancing. The idea is simply to break the input (and/or output) into equal sized blocks.[2] The number of blocks is often chosen to be proportional to the number of processors.

A reduce can be implemented efficiently in two phases. The first phase sequentially computes a partial sum for each block, in parallel across blocks. The second phase sums these partial sums, often sequentially because the number of blocks is small. For n elements across b blocks, the first phase does n reads and b writes, and the second b reads and one write.

Scan can be implemented similarly, with three phases [10] instead of two. The first phase again sums within blocks. The second phase then does a scan on these partial sums. The results of the scan are then used as offsets for the third phase, which rereads the input along with the offsets to do a scan within each block, each starting from an offset. See Figure 2. Here the first phase does n reads and b writes, the second b reads and writes, and the third $n + b$ reads, and n writes.

[2] Throughout we will ignore roundoff issues.

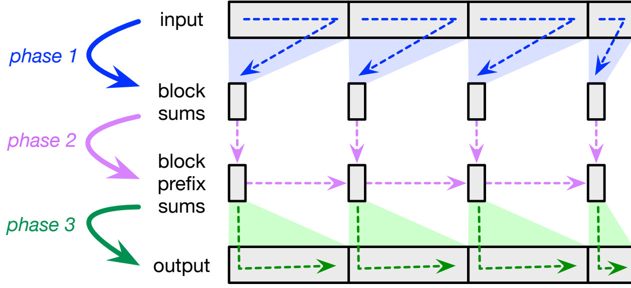

Figure 2. Three phases of scan

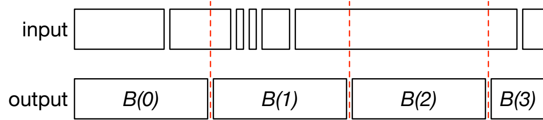

Figure 3. Flattening sequences into multiple uniform blocks.

The `flatten` operation—illustrated in Figure 3—blocks the output iteration space instead of the input. It first does a scan on the input lengths, which returns the offset for each subsequence in the output, as well as the overall size. The overall size is partitioned into equal sized blocks, and the start of each block is merged with the calculated offsets. This tells each block which subsequences or parts of them it is responsible for. Each block then copies its part to the output.

The block-based filter works in two phases. The first phase does a filter within each block, packing the kept elements within the block into a contiguous region. The second phase is simply a flatten on the packed blocks as described above.

Note that for these implementations, all inner loops within blocks are sequential. This is important for our technique.

3 Technical Overview and Examples

Our idea is to break a sequence into many equal sized blocks, where each block is a (delayed) stream. We call this representation a ***block-iterable delayed sequence (BID)***. The insight is that the BIDs work well with with the common block-based implementations of reduce, scan, filter, and flatten (as described in Section 2.2) because the inner loop on each block is sequential and therefore can be converted into a stream. If the block boundaries are chosen consistently across sequences, then the streams from one operation can be fused blockwise with streams of the previous or next operation. More implementation details are given in Section 4.

In comparison to the stream-of-blocks technique (Section 2.1), we essentially turn it "inside-out", using blocks of streams instead. This offers two important advantages. First, it is well-suited for coarse-grained parallelism which is required on multicores. Second, it efficiently supports a broader set of operations on sequences (such as `flatten`).

In addition to BIDs, we also support a form of random-access index fusion, which we refer to as ***random access delayed sequences (RADs)***. Any RAD can be converted to a BID in work and space proportional to the number of blocks, but to convert from BID to RAD requires first converting to a non-delayed sequence to allow for random access. We supply a mechanism to *force* a RAD or BID into a standard non-delayed sequence. This can be useful to avoid evaluating a delayed sequence more than once when passed to more than one operation. Our cost semantics (Section 5) helps the user determine the tradeoffs of when or when not to delay.

```
fun bestCut(A) =
  let val isEnd = map f A
      val (endCounts, _) = scan (op+) 0 isEnd
      val costs = map g endCounts
  in reduce h costs end
```

Figure 4. Simplified version of best-cut for ray tracing.

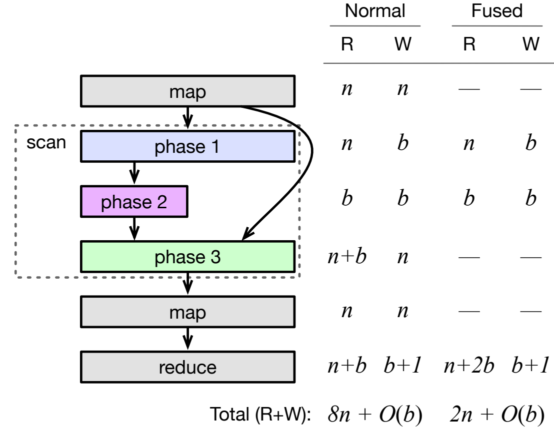

Figure 5. The sequence of operations for best-cut showing the three phases of scan for n elements and b blocks. On the right, the number of reads (R) and writes (W) for both normal and fused versions.

Example: best-cut for ray tracing. This example is motivated by a ray-tracing benchmark in the PBBS benchmark suite [30], which recursively builds a kd-tree by partitioning triangles based on the surface area heuristic [23]. To decide how to partition, the heuristic picks an axis-aligned cut of the volume so as to minimize a cost function involving the surface areas of the two resulting subvolumes, and the number of triangles that fall in each.

Figure 4 shows a simplified version of the best-cut code, ignoring some implementation details (e.g., the definitions of f, g, and h.) To compute the best cut, four operations are applied in sequence: map, scan, map, and reduce.

Figure 5 illustrates the sequence of operations with the scan broken into its three phases. To the right is the number of reads and writes required by each operation, assuming an input sequence of length n, and b blocks. Without fusion there are a total of $8n + O(b)$ reads and writes. In the fused version the initial map can be fused with the first phase of

```
type graph // graph interface
type vertex = int
val numVertices: graph → int
val outNeighbors: graph → vertex → vertex seq

fun BFS (G: graph) (s: vertex) = // search from "source" s
  let val n = numVertices G
      val P = arrayTabulate n (fn i ⇒ −1)     // parents
      fun outPairs (u: vertex) = // tag neighbors with self
        map (fn v ⇒ (u,v)) (outNeighbors G u)
      fun tryVisit(u,v) =     // try to visit v from parent u
        if compareAndSwap(P,v,−1,u) then SOME(v)
        else NONE
      fun search(F: vertex seq) =
        if |F| = 0 then () else
        let val E = flatten (map outPairs F)
            val F′ = filterOp tryVisit E   // next frontier
        in search F′ end
  in tryVisit (s,s);   // visit source vertex (use self as parent)
      search ⟨s⟩;       // do the BFS (first frontier: source only)
      P end             // after search is done, return parents
```

Figure 6. Forward BFS with sequences

the scan. In particular, the input is converted into b blocks of streams, each of which is fused with one of the blocks in the first phase of the scan. The parallelism is across blocks. The third phase of the scan can be fused with the map and then the reduce. Again this uses stream fusion within each block. The final fused loops only pass over the data twice requiring only a total of $2n + O(b)$ reads and $O(b)$ writes.

We note that the fusion comes at the cost of evaluating the elements of the initial map twice, once in each of the fused loops. An alternative is to force the result of the initial map so that it is only calculated once. This comes at the cost of increasing the number of reads and writes to $4n + O(b)$ (the force would require n reads and n writes). This difference is exposed by our cost semantics (Section 5) without needing to know the details of the implementation.

Example: Breadth-first search (BFS). Our second example uses flatten and filter to generate a BFS tree. The pseudocode is shown in Figure 6. On each iteration the code maps over the frontier (i.e. all vertices at some distance i from the source), generates all their neighbors, and then keeps those that have not yet been visited. This first involves a `map` with a nested `map` inside (`outPairs`), followed by a `flatten` to generate a sequence of source-destination pairs corresponding to all pairs of a frontier vertex and an out neighbor. It then uses a `filterOp` to filter out any visited neighbors, and at the same time labeling each newly visited vertex with its parent in the BFS tree, using shared mutable state. There is a race (among those in the frontier to claim an unvisited neighbor and set its parent) which is resolved with a `CompareAndSwap`.

By using fusion in our approach, the flattened sequence of pairs is never instantiated. Furthermore, the filter only needs to pack down within blocks without having to return the result as a contiguous array. This significantly improves performance as show in Section 6, and improved the performance in the PBBS benchmarks [30]. A more in-depth cost analysis, using our cost semantics, is given in Section 5.1.

4 Block-Delayed Sequences

At a high level, our delayed sequence type is a tagged union of two different representations, called RAD and BID.

```
datatype α seq =   // the delayed sequence type
| RAD of int × int × (int → α)   // random–access delay
| BID of int × (int → α stream) // blocked–iterable delay
```

A ***RAD*** (random-access delayed sequence) uses a tuple (i, n, f) to represent the sequence $\langle f(i), \ldots, f(i+n-1)\rangle$. In this representation, any element can be retrieved independently by calling the index lookup function f at the appropriate index.

A ***BID*** (blocked-iterable delayed sequence) represents a sequence with the tuple (n, b) where n is the total number of elements and b is a function used to retrieve blocks. Specifically, $b(i)$ is the ith block, and there are $\lceil n/\mathcal{B}_n\rceil$ blocks for a sequence of size n, where $\mathcal{B}_n$ is the block size. Ignoring roundoff issues for the last block, each block contains the same number of elements.

There are a number of reasonable ways to choose the block size $\mathcal{B}_n$: it could be set as a constant at compile-time, or could be computed as n/P where P is the number of processors, etc. Our definitions work the same for any block-size.

Notation Convention. The code presented in this section uses three distinct types of sequence-like data: arrays, streams, and our delayed sequences. Some standard functions have multiple implementations, one for each type. To distinguish these, we write `a.XXX` to indicate the version of function `XXX` which operates on arrays, and similarly `s.XXX` for the stream version. No prefix is used for the version defined for delayed sequences.

4.1 Auxiliary Definitions

The implementation presented in this section relies on a few auxiliary functions, which are shown in Figure 7.

Parallel Apply. The `apply` function (Figure 7) is essentially a `tabulate` with no result. It takes a length n and function f, and executes $f(i)$ in parallel for each $0 \le i < n$. The `apply` function is the only parallel primitive we use, and can be implemented using parallel divide-and-conquer over the iteration space.

Arrays. Delayed sequences internally use arrays to hold intermediate results where necessary. We therefore use a

```
val apply: int → (int → unit) → unit

type α array   // arrays with elements of type α
val a.allocate: int → α array
val a.length: α array → int
val a.tabulate: int → (int → α) → α array
val a.map: (α → β) → α array → β array
val a.reduce: (α × α → α) → α → α array → α
val a.scan:
  (α × α → α) → α → α array → α array × α
```

Figure 7. Auxiliary code: parallel apply and arrays.

```
type α stream   // delayed streams of elements of type α
val s.length: α stream → int

// these operations require only O(1) work
val s.tabulate: int → (int → α) → α stream
val s.map: (α → β) → α stream → β stream
val s.zip: α stream × β stream → (α × β) stream
val s.scan:
  (α × α → α) → α → α stream → α stream

// these operations require (at least) linear work
val s.reduce: (α × α → α) → α → α stream → α
val s.applyStream: α stream → (α → unit) → unit
val s.packToArray:
  (α → bool) → α stream → α array
```

Figure 8. Streams

```
 1 fun BIDfromSeq (BID(n,b)) = BID(n,b)
 2 fun BIDfromSeq (RAD(i,n,f)) =
 3   BID(n, fn j ⇒
 4     s.tabulate 𝓑n (fn k ⇒ f(i + j · 𝓑n + k)))

 5 fun applySeq S f =
 6   let val BID(n,b) = BIDfromSeq S
 7       fun doBlock(i) = s.applyStream (b(i)) f
 8   in apply ⌈n/𝓑n⌉ doBlock end

 9 fun toArray S =
10   let val n = length S
11       val A = a.allocate n   // new array
12       val I = RAD(0,n, fn i ⇒ i)
13   in applySeq (zip (I,S)) (fn (i,x) ⇒ A[i] := x);
14       A end

15 fun RADfromArray A = RAD(0, |A|, fn i ⇒ A[i])

16 fun force S = RADfromArray (toArray S)

17 fun RADfromSeq (RAD(i,n,f)) = RAD(i,n,f)
18 fun RADfromSeq (BID(n,b)) = force (BID(n,b))
```

Figure 9. Conversions between arrays, RADs, and BIDs.

small parallel array library conforming to the interface shown in Figure 7.

4.2 Blocks as Streams

In the BID representation, each block is a delayed stream of values. Streams can be represented in a number of ways, and in fact our implementation of streams differs between Parallel ML and C++, which we discuss in more detail in Section 4.4. The only requirement here is that streams are delayed, and so can be constructed in $O(1)$.

Figure 8 shows the interface for a small set of functions on streams, implemented both in Parallel ML and C++ for our framework. Outside of standard sequence functions, there are two special functions we need: `s.applyStream` and `s.packToArray`. The `s.applyStream` function applies a function to each element produced by the stream. The `s.packToArray` function is essentially a filter, except it outputs an array; it is fully sequential and can be implemented with dynamic-resizing arrays to ensure that only as much memory is allocated as needed.

All stream functions require only $O(1)$ work except for three: `s.reduce`, `s.applyStream`, and `s.packToArray`. These three functions require $O(n)$ work and span for a stream of length n (assuming the function passed as argument is $O(1)$).

4.3 Implementing Block-Delayed Sequences

We present the implementation of our block-delayed sequences in Figures 9 and 10. Figure 9 defines a few functions which convert between different representations of sequences (arrays, RADs, and BIDs). These are then used in Figure 10, which implements standard functions such as `map`, `scan`, `filter`, etc.

Between RADs and BIDs. The function `BIDfromSeq` (Figure 9, lines 1-4) converts a sequence to the BID representation. Any BID given as input is left unchanged. RADs are blockified by reindexing: the jth block of the output starts at offset $j \cdot \mathcal{B}_n$.

Similarly, the function `RADfromSeq` (Figure 9, lines 17-18) ensures that a sequence is RAD. To convert a BID to a RAD, we "force" it (`toArray`, lines 9-14) by allocating an array of the appropriate size, and then traversing the BID to write each element into the array. Finally, we package up the array as a RAD. Traversing a BID is accomplished by the function `applySeq` (Figure 9, lines 5-8) which calls `s.applyStream` in parallel across the blocks.

Force. The `force` function (Figure 9, line 16), while not a standard sequence function, is useful for programming with delayed sequences, to ensure that all delayed work has been performed. It is essentially the same as `RADfromSeq`, except that a RAD given as input will be fully evaluated, too.

Tabulate and Map. The `tabulate` function (Figure 10, line 19) is fully delayed in the sense that it always returns a RAD, requiring only $O(1)$ work. Similarly, `map` (Figure 10,

```
19 fun tabulate n f = RAD(0,n,f)

20 fun map g (RAD(i,n,f)) = RAD(i,n,g ∘ f)
21 fun map g (BID(n,b)) = BID(n, (s.map g) ∘ b)

22 fun zip (RAD(i,n,f), RAD(j,n,g)) = // if both are RAD...
23   RAD(0,n, fn k ⇒ (f(i+k),g(j+k)))
24 fun zip (S1,S2) =   // if at least one is BID...
25   let val BID(n,b1) = BIDfromSeq S1
26       val BID(n,b2) = BIDfromSeq S2
27   in BID(n, fn i ⇒ s.zip(b1(i),b2(i))) end

28 fun reduce f z S =
29   let val BID(n,b) = BIDfromSeq S
30       val sums =   // phase 1
31         a.tabulate ⌈n/𝓑n⌉ ((s.reduce f z) ∘ b)
32   in a.reduce f z sums end // phase 2

33 fun scan f z S =
34   let val BID(n,b) = BIDfromSeq S
35       val sums =   // phase 1
36         a.tabulate ⌈n/𝓑n⌉ ((s.reduce f z) ∘ b)
37       val (P,t) = a.scan f z sums     // phase 2
38       val R =    // phase 3
39         BID(n, fn i ⇒ s.scan f (P[i]) (b(i)))
40   in (R,t) end

41 fun getRegion(S: α seq array, offsets, n, i) =
42   ... // 1. binary search in offsets to find start of i^th block
43       // 2. construct stream (length 𝓑n) to walk through S

44 fun flatten S =
45   let val S′ = a.map RADfromSeq (toArray S)
46       val (off,n) = a.scan (op+) 0 (a.map length S′)
47   in BID(n, fn i ⇒ getRegion(S′,off,n,i)) end

48 fun filter p S =
49   let val BID(n,b) = BIDfromSeq S
50       val S′ = a.tabulate ⌈n/𝓑n⌉
51         (RADfromArray ∘ (s.packToArray p) ∘ b)
52       val (off,m) = a.scan (op+) 0 (a.map length S′)
53   in BID(m, fn i ⇒ getRegion(S′,off,m,i)) end
```

Figure 10. Implementation of block-delayed sequences.

lines 20-21) is also fully delayed, and is efficiently implemented with function composition. There are two cases. For RAD input, the index function f is replaced by $g \circ f$. For BID input, the block-stream function b is replaced by the composition (s.map g)$\circ b$. That is, when a block of the output is needed later, the block will be computed by first calling b (which produces a stream) and then applying the stream-map, creating another stream of the resulting values. These function calls are delayed: each block of the output is encoded as a function that will only be called later when the elements of the block are needed.

Zip. The `zip` function (Figure 10, lines 22-27) has two cases. When both sequences given as input are RAD, then the output is also RAD. But when at least one input sequence is BID, we fall back on converting both sequences to BID and then applying the stream-zip pairwise to block-streams. Note that we require that both sequences are the same length: this way, the blocks are guaranteed to align.

Reduce and Scan. The `reduce` function (Figure 10, lines 28-32) first converts the input to a BID, and then begins by eagerly computing the sum of each block by calling the stream-reduce for each block-stream (this is the first phase of the reduce, as described in Section 2). They are then summed up to produce the final result (second phase).

The `scan` function (Figure 10, lines 33-40) is more interesting since it supports functionality not supported by RADs. In particular, its output is a BID, which means that phase 3 of the algorithm can be delayed. Looking at the implementation, phase 1 can be fused with its input (either a BID or RAD) but will be fully evaluated to generate the `sums` sequence. In phase 2, the array-scan is computed eagerly, but only needs to operate over the number of blocks (rather than the full input size). Phase 3 then generates delayed streams for each block as a BID.

Flatten. The `flatten` function (Figure 10, lines 44-47), like `scan`, allows both its input and output to be fused with surrounding operations. For the output, the key idea is to partition the output index space into equal-sized blocks so that the output can be represented as a BID. This involves doing an array-scan on the sizes of each subsequence to produce offsets for the start of each subsequence in the output. Then each block can extract a region of the top-level sequence, as depicted previously in Figure 3.

In the code, the output blocks are defined by a function `getRegion` whose implemention is omitted for brevity. We implement this function by binary searching on the offsets to find which subsequence the block begins in, and then constructing a stream (of length $\mathcal{B}_n$) which walks left-to-right across adjacent subsequences. Because the beginning of a block may be in the middle of one of the subsequences, we force that all the subsequences are RAD (line 45).

Altogether, assuming all input sequences are RAD, this results in eager work proportional to the length of the outer sequence. The rest of the work is delayed (e.g. the binary searches to find the fronts of the blocks will not occur until the blocks are actually computed by a different function).

Filter. The `filter` function (Figure 10, lines 48-53) begins by packing the input down into many arrays, one for each block. Then (similar to `flatten`) the `getRegion` approach is again used to package up these arrays into a BID. As a result, the filter only allocates space for the elements which survive (as well as the space for the subsequence offsets). By using a BID for the output, we avoid needing to copy the surviving elements into a final output array.

4.4 Library Implementation

We implemented block-delayed sequences in both Parallel ML and C++ as libraries. Note that in Parallel ML, the library code is very similar to that shown in Figure 10.

In our C++ implementation, we use forward iterators to implement streams. The iterators maintain their current state, and each time they are incremented, the next position is evaluated, updating the state. Templates are used to make the type of the iterators specific to the underlying data, which could itself be a stream that was composed with a delayed operation, making it particularly easy for C++ compilers to inline the resulting fusion. Overloading is used to dispatch on the type of the sequence (BID, RAD or fully realized). Our implementation is built on top of the Parlay Library [5].

In ML, we used an approach similar to forward iterators, in the form of functions of the type `unit` → `unit` → α. These functions have a particular mode of use, where applying the first unit produces a stateful "trickle" function that can be called repeatedly to produce elements of the stream. Internally, the trickle function maintains the current state of the stream. All of these details are internal to the library implementation.

5 Cost Semantics

We define costs in terms of work, span, and allocations. The ***work*** $\mathcal{W}(e)$ is the total number of steps to evaluate an expression e. The ***span*** $\mathcal{S}(e)$ counts the number of steps on the critical path. The ***allocation-count*** $\mathcal{A}(e)$ is the total size of all intermediate arrays. These costs are defined in Figure 11 for selected functions from the sequence library. Costs for omitted functions can be similarly defined.

In Figure 11, the eager costs ($\mathcal{W}$, $\mathcal{S}$, $\mathcal{A}$) are those that are incurred now, and the delayed costs ($\mathcal{W}^*$, $\mathcal{S}^*$, $\mathcal{A}^*$) are those that are possibly incurred later, when the output sequence is passed as input to another operation. Delayed costs appear in the cost semantics as properties of individual sequences. That is, just as a sequence X has a length $|X|$ and elements X_i, we also equip it with delayed costs at each index.

1. $\mathcal{W}^*_X(i) \in \mathbb{N}$ is the delayed work at index i of X.
2. $\mathcal{S}^*_X(i) \in \mathbb{N}$ is the delayed span at index i of X.
3. $\mathcal{A}^*_X(i) \in \mathbb{N}$ is the delayed allocation at index i of X.

For example, the expression **tabulate** $n\ f$ always has constant eager costs, i.e., $\mathcal{W}(\textbf{tabulate}\ n\ f) \in O(1)$. This is because all computation is delayed in the output Y. Later, these delayed costs may contribute to an eager cost, such as in **force** which causes all delayed computation to occur eagerly. In Figure 11 this is expressed as $\mathcal{W}(\textbf{force}\ X) = \sum_i \mathcal{W}^*_X(i)$, i.e., the work of **force** is the sum of delayed input work (and similarly for span and allocations).

We also keep track of the representation $\mathcal{R}(X) \in \{\textbf{RAD}, \textbf{BID}\}$ of each sequence X, which can affect the cost of operations. Specifically, in some cases in Figure 11, we assume the input sequence is **RAD**. This is purely for brevity and is not a restriction on when the costs are defined. Our library conforms to these cost specifications by implicitly **force**'ing sequences where necessary.

Block maxes. When specifying span costs, we write bmax^n_i to indicate the max of sums of blocks in a sequence of length n. Specifically, this is defined as follows.

$$\text{bmax}^n_i \ldots = \max_{j=0}^{\lceil n/\mathcal{B}_n \rceil - 1} \sum_{i=j\mathcal{B}_n}^{(j+1)\mathcal{B}_n - 1} \ldots$$

For **BID**s, bmax computes spans appropriately, as each block is internally sequential (requiring a sum) but all blocks are processed in parallel (requiring a max). Note also that many operations on **RAD**s internally convert to a **BID** (see Figure 10), and therefore are still subject to bmax'ed spans.

Simple function arguments. There is a standard problem with higher-order functions, where to accurately specify costs, the implementation of the library itself must be exposed. Therefore, for simplicity here, we assume that the functions f passed as argument to both **reduce** and **scan** are "simple" in the sense that they are constant-time functions for all inputs and do not perform any intermediate allocations. This covers typical uses such as scan-plus, reduce-min, etc.

5.1 Cost analysis example: Forward BFS

We now use our cost semantics to analyze the BFS from Figure 6. For an input graph with N vertices, M edges, and a diameter D, the total (eager) costs of the BFS under our semantics are $O(N + M)$ work, $O(D(\log N + \mathcal{B}))$ span, and $O(N + M/\mathcal{B})$ allocation. (For this analysis we assume the block-size is a fixed value $\mathcal{B}$.) The work bound shows that BFS is work-efficient, and the span bound shows that it is highly parallel. The amount of allocation is asymptotically better than the $O(N + M)$ allocation that would be incurred by a naïve array-based implementation.

These bounds are derivable directly from our cost semantics. For example, consider analyzing the allocations. A single round of BFS has input F, output F', and consists of a map, a flatten, and a filterOp (which has the same cost as filter). The map incurs no eager allocation, and delays no allocations in its output. The flatten incurs $|F|$ eager allocation (as well as any delayed allocation $\mathcal{A}^*_F(i)$ for each element of F, but these costs are 0). The filter then allocates $|F'| + |E|/\mathcal{B}$ (as well as any allocations due to applying the `tryVisit` function or delayed allocation on each element, but again these are 0). Altogether, that adds up $|F| + |F'| + |E|/\mathcal{B}$ allocation for a single round. Summing over all rounds then yields a total allocation of $O(N + M/\mathcal{B})$.

6 Experimental evaluation

We present an empirical evaluation of our delayed sequences, as implemented in both C++ and Parallel ML, against two baseline implementations: a highly optimized parallel array library with no fusion, and an extension of the array-only library which enables RAD fusion, but not BID fusion. These

Operation	Output Representation and Delayed Costs				Eager Work	Eager Span	Eager Alloc
	$\mathcal{R}(Y)$	$\mathcal{W}_Y^*(i)$	$\mathcal{S}_Y^*(i)$	$\mathcal{A}_Y^*(i)$	$\mathcal{W}$	$\mathcal{S}$	$\mathcal{A}$
$Y =$ **force** X	**RAD**	1	1	0	$\sum_i \mathcal{W}_X^*(i)$	$\text{bmax}_i^{\lvert X\rvert}\ \mathcal{S}_X^*(i)$	$\lvert X\rvert + \sum_i \mathcal{A}_X^*(i)$
$Y =$ **tabulate** $n\ f$	**RAD**	$\mathcal{W}(f(i))$	$\mathcal{S}(f(i))$	$\mathcal{A}(f(i))$	1	1	0
$Y =$ **map** $f\ X$	$\mathcal{R}(X)$	$\mathcal{W}_X^*(i)+$ $\mathcal{W}(f(X_i))$	$\mathcal{S}_X^*(i)+$ $\mathcal{S}(f(X_i))$	$\mathcal{A}_X^*(i)+$ $\mathcal{A}(f(X_i))$	1	1	0
$Y =$ **filter** $p\ X$	**BID**	1	1	0	$\sum_i \left[\mathcal{W}_X^*(i) + \mathcal{W}(p(X_i))\right]$	$\text{bmax}_i^{\lvert X\rvert} \left[\mathcal{S}_X^*(i) + \mathcal{S}(p(X_i))\right] + \log\lvert X\rvert$	$\lvert Y\rvert + \lvert X\rvert/\mathcal{B}_{\lvert X\rvert} + \sum_i \left[\mathcal{A}(p(X_i)) + \mathcal{A}_X^*(i)\right]$
$Y =$ **flatten** X where $\mathcal{R}(X_i) =$ **RAD**	**BID**	See note†	See note†	See note†	$\sum_i \mathcal{W}_X^*(i)$	$\log\lvert X\rvert + \text{bmax}_i^{\lvert X\rvert}\ \mathcal{S}_X^*(i)$	$\lvert X\rvert + \sum_i \mathcal{A}_X^*(i)$
$(Y, _) =$ **scan** $f\ b\ X$ where f is simple	**BID**	$1 + \mathcal{W}_X^*(i)$	$1 + \mathcal{S}_X^*(i)$	$1 + \mathcal{A}_X^*(i)$	$\sum_i \mathcal{W}_X^*(i)$	$\log\lvert X\rvert + \text{bmax}_i^{\lvert X\rvert}\ \mathcal{S}_X^*(i)$	$\lvert X\rvert/\mathcal{B}_{\lvert X\rvert} + \sum_i \mathcal{A}_X^*(i)$
$_ =$ **reduce** $f\ b\ X$ where f is simple	–	–	–	–	$\sum_i \mathcal{W}_X^*(i)$	$\log\lvert X\rvert + \text{bmax}_i^{\lvert X\rvert}\ \mathcal{S}_X^*(i)$	$\lvert X\rvert/\mathcal{B}_{\lvert X\rvert} + \sum_i \mathcal{A}_X^*(i)$

†For $Y =$ **flatten** X, delayed costs for each index are carried through to the output. Specifically, we have $\mathcal{W}_Y^*(i) = \mathcal{W}_{X_j}^*(i - O(j))$ where X_j is the inner sequence which contains ith overall element and $O(j)$ is the offset of X_j. Delayed span and allocations are computed in the same way.

Figure 11. Costs of sequence operations. Each operation takes an input sequence X, eagerly performs work $\mathcal{W}$, span $\mathcal{S}$, and allocations $\mathcal{A}$, and (possibly) produces output sequence Y.

Name	Abbr	Fusion	Description
array	A	none	highly optimized parallel arrays
rad	R	RAD only	extends A with RAD fusion (for tabulate, map, reduce, etc.)
delay	**Ours**	RAD+BID	our block-delayed sequences

Figure 12. Library implementations used in evaluation

comparisons allow us to separately determine the impact of fusion due to our two (RAD and BID) representations.

The three libraries used in this section, as summarized in Figure 12, are respectively referred to as *array* (A), *rad* (R), and ***delay* (Ours)**. Each library is implemented in both C++ and Parallel ML, resulting in six implementations total. The two baseline libraries (array and rad) are based on ParlayLib [5] for C++, and all Parallel ML code (libraries and benchmarks) is ported from C++. All code used in this evaluation is publicly available.[3]

Our experiments collectively show that:

- For benchmarks which utilize BID fusion, in almost all cases, we see a significant improvement in both time and space due to the BID representation.
- At full scale (72 processors), BIDs are consistently faster and use less space than RADs alone, with up to 2.7x and 10x improvements in time and space, respectively.
- RADs alone offer significant benefits in comparison to arrays without fusion, providing up to 19x improvement in time and 92x improvement in space on 72 processors.
- Our block-delayed sequences are portable across different programming languages and paradigms, providing similar performance benefits in both C++ and Parallel ML.

Benchmarks. Our benchmarks are split into two categories: those that utilize BID fusion (via operations such as `scan`, `flatten`, and `filter`), and those that only utilize RAD fusion. All benchmark codes are well optimized; in particular, we verified that with our block-delayed sequences library, the C++ benchmarks perform similarly to hand-optimized codes from the state-of-the-art PBBS [30] benchmark suite. Many of the benchmarks utilize nested parallelism, which our libraries support seamlessly.

BID Benchmarks. The following benchmarks utilize both BID and RAD fusion. The ***bestcut*** benchmark performs a kd-tree best cut (as described in Section 3) on 200M bounding boxes of triangles. ***bfs*** computes a forward BFS as shown in Figure 6; the input is random power-law graph [7] with approximately 16.7M vertices and 199M edges. ***bignum-add*** performs addition on two bignums of 500M bytes each. ***primes*** calculates all primes less than 100M. ***tokens*** splits 500M characters into words (average word length 7).

RAD Benchmarks. These benchmarks do not use BIDs but make extensive use of RADs. ***grep*** finds all lines of a file that contain a pattern, similar to Unix `grep`; the input is 843M characters across 28M lines where approximately 850K lines match the pattern. ***integrate*** calculates the integral of $\sqrt{1/x}$ for $x \in [1, 1000]$ using $n = 500\text{M}$ points. ***linearrec*** solves a linear recurrence of the form $R_i = x_i R_{i-1} + y_i$; the input is 500M pairs of doubles (x_i, y_i). ***linefit*** finds a line of best fit for 500M 2D points (pairs of doubles). ***mcss*** computes the maximum contiguous subsequence sum of an array of 500M 64-bit integers. ***quickhull*** computes the convex hull of 20M points (pairs of doubles) in a circle from a uniform distribution. ***sparse-mxv*** multiplies a sparse matrix (2M rows, 200M nonzero entries) with a vector of length 2M. ***wc*** counts the number of lines, words, and bytes (same as Unix `wc`) in a file of 500M chars.

Experimental setup. We run all experiments on a 72-core Dell PowerEdge R930 with 4 × 2.4GHz Intel 18-core E7-8867 v4 Xeon chips, 1TB of main memory, 45MB of L3 cache per chip, and a memory bus clocked at 4800MHz. The C++ programs are compiled with `g++` version 7.4.0 with `-O3`,

[3] https://github.com/mpllang/delayed-seq

	Time								Space							
	$P = 1$				$P = 72$				$P = 1$				$P = 72$			
	A	R	Ours	$\frac{R}{Ours}$	A	R	Ours	$\frac{R}{Ours}$	A	R	Ours	$\frac{R}{Ours}$	A	R	Ours	$\frac{R}{Ours}$
bestcut	43.9	13.9	9.52	**1.5**	.912	.258	.158	**1.6**	34	9.0	.82	**11**	33	8.9	.87	**10**
bfs	18.8	14.1	15.2	**0.93**	.509	.470	.425	**1.1**	9.6	7.7	4.3	**1.8**	13	8.7	5.7	**1.5**
bignum-add	7.87	7.38	4.56	**1.6**	.151	.132	.080	**1.7**	2.6	2.1	1.5	**1.4**	2.7	2.2	1.6	**1.4**
primes	13.7	9.05	7.99	**1.1**	.647	.181	.136	**1.3**	8.1	2.2	.16	**14**	7.5	2.2	.22	**10**
tokens	22.7	12.6	6.85	**1.8**	.746	.322	.125	**2.6**	21	9.6	1.5	**6.4**	18	9.7	1.6	**6.1**
bestcut	3.40	1.90	2.90	**0.66**	.196	.063	.053	**1.2**	8.8	2.4	.87	**2.8**	10	3.6	2.0	**1.8**
bfs	17.1	8.41	6.72	**1.3**	.406	.211	.168	**1.3**	7.1	5.7	5.3	**1.1**	8.1	6.9	6.4	**1.1**
bignum-add	2.03	2.15	2.07	**1.0**	.150	.093	.048	**1.9**	7.0	5.0	3.1	**1.6**	8.2	6.2	4.3	**1.4**
primes	2.54	2.42	1.56	**1.6**	.159	.118	.071	**1.7**	5.0	3.0	.98	**3.1**	6.3	4.1	2.1	**2.0**
tokens	3.62	2.86	1.59	**1.8**	.176	.077	.029	**2.7**	7.0	3.0	1.6	**1.9**	8.2	4.2	2.7	**1.6**

Figure 13. Benchmarks with BID improvement. Top half: results for MPL (Parallel ML). Bottom half: C++. Times are in seconds, and space numbers are in GB.

	Time						Space					
	$P = 1$			$P = 72$			$P = 1$			$P = 72$		
	A	Ours	$\frac{A}{Ours}$	A	Ours	$\frac{A}{Ours}$	A	Ours	$\frac{A}{Ours}$	A	Ours	$\frac{A}{Ours}$
grep	14.7	8.61	**1.7**	.359	.153	**2.3**	8.9	1.7	**5.2**	9.2	2.0	**4.6**
integrate	4.99	3.24	**1.5**	.130	.053	**2.5**	4.0	.015	**267**	4.1	.044	**93**
linearrec	41.9	10.7	**3.9**	1.04	.454	**2.3**	55	33	**1.7**	64	33	**1.9**
linefit	10.5	2.39	**4.4**	.497	.143	**3.5**	24	8.0	**3.0**	24	8.1	**3.0**
mcss	11.5	4.83	**2.4**	.397	.081	**4.9**	20	4.0	**5.0**	20	4.1	**4.9**
quickhull	11.7	3.60	**3.2**	.479	.121	**4.0**	6.7	2.7	**2.5**	11	3.1	**3.5**
sparse-mxv	4.12	2.10	**2.0**	.090	.052	**1.7**	6.3	4.2	**1.5**	4.8	4.9	**0.98**
wc	5.89	2.25	**2.6**	.195	.037	**5.3**	8.5	.54	**16**	8.6	.58	**15**
grep	4.33	3.29	**1.3**	.085	.064	**1.3**	2.4	1.5	**1.6**	3.6	2.6	**1.4**
integrate	2.41	1.85	**1.3**	.086	.032	**2.7**	4.0	.016	**250**	5.2	1.2	**4.3**
linearrec	4.76	2.44	**2.0**	.366	.198	**1.8**	20	12	**1.7**	21	13	**1.6**
linefit	4.36	2.14	**2.0**	.308	.146	**2.1**	16	8.0	**2.0**	17	9.2	**1.8**
mcss	4.83	.987	**4.9**	.376	.038	**9.9**	20	4.0	**5.0**	21	5.2	**4.0**
quickhull	1.33	1.14	**1.2**	.055	.033	**1.7**	2.8	2.6	**1.1**	4.2	3.8	**1.1**
sparse-mxv	3.07	1.78	**1.7**	.062	.042	**1.5**	4.2	4.2	**1.0**	5.3	5.3	**1.0**
wc	2.96	.510	**5.8**	.171	.009	**19**	8.5	.52	**16**	9.7	1.7	**5.7**

Figure 14. Benchmarks with RAD-only improvements. Top half: MPL (Parallel ML). Bottom half: C++. Times are in seconds, and space numbers are in GB.

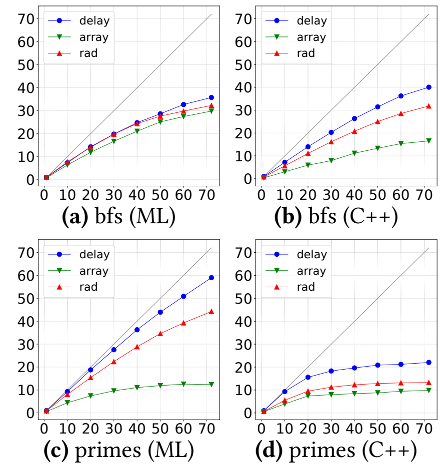

Figure 15. Speedups w.r.t 1-processor delay. (Processors on x-axis, speedups on y-axis.)

using ParlayLib [5] and the `jemalloc` library. For C++, we use `numactl -i all` to allocate pages on NUMA nodes in a round-robin fashion. The Parallel ML programs are compiled with MPL [2, 36], version `v0.2`. For each benchmark result, we report the average over 10 runs. We compute space with maximum residency as reported by Linux for a single run of the benchmark (and then average across 10 repetitions).

Disentangled Codes. All of our Parallel ML codes are *disentangled* [36], a property which enables provably efficient parallel memory management [2], and is a requirement of using the MPL compiler.

6.1 Impact of BID fusion

Although it offers more opportunities for fusion, our BID representation also introduces some potential for noticeable runtime overheads. Such overheads might be triggered for example by increased code size, additional conditionals (e.g. to check the representation of a sequence) hampered compiler optimizations, etc. To determine the impact of BIDs in comparison to the much simpler RAD-only fusion, we consider the five benchmarks that utilize at least one instance of BID fusion.

In Figure 13, we present results for these benchmarks in terms of execution time (columns 1-8) and space usage (columns 9-16) on both $P = 1$ and $P = 72$ processors. Each benchmark has three versions for the three libraries we consider, as described in Figure 12: array-based (A), RAD-based (R), and the full library (Ours). The improvement of the full library version due to BID fusion (in comparison to the RAD-only library) is shown via the ratio in column "R/Ours". We also include the array-based numbers here just for completeness; note that the RAD results are consistently faster and

use less space than the array-based implementations, often by a significant margin.

Comparing our full library to the RAD-only library, we first observe that across the board, our full library consistently uses less space, with improvements ranging from 1.1x to 14x. On 72 processors, the full library is always faster, with between 1.1x and 2.7x improvement. On 1 processor, there are two instances in which our full library is slower: ***bfs*** in Parallel ML and ***bestcut*** in C++. This is due in part to the BID representation being optimized for parallel execution. BIDs require additional work to manage multiple representations, for example in `flatten` where the library checks that all inner sequences are RAD (Figure 10, line 45). There may also be additional performance impacts due to other factors such as insufficient code inlining or increased code size. At single-core scale, where execution time is more sensitive to compute-based overheads, this additional work (although small) appears to be occasionally non-negligible. At full scale however, where memory bandwidth becomes the bottleneck, these overheads are easily outweighed by reduced memory pressure. As such, we believe that our BIDs fall on the right side of the time/space tradeoff: BIDs reduce memory pressure at the expense of some modest compute-based overheads, which favors performance at full scale.

6.2 Impact of RAD-only fusion

In Figure 14, we compare the performance of our delayed-sequence library against the baseline array library. The benchmarks in this section do not use BIDs, but make extensive use of RAD fusion. For each benchmark, the column labeled "Ours" uses our library, whereas the "A" column uses array-based sequences.

Overall, in both MPL and C++, our delayed versions are as fast or faster in all configurations. Speedups range from 1x to 19x, but most are close to 3x. The impact of delaying is especially pronounced in cases where its use changes the memory access pattern of the workload. In some cases, for instance, delaying can improve the benchmark from performing $O(n)$ reads and writes to $O(n)$ reads and $O(1)$ writes, which can be significant owing to the relatively large cost of writes on our test machine. The benchmarks that are affected in this manner are ***mcss***, ***linefit***, ***wc***, and ***integrate***.

In the majority of cases, the speedup from delaying is larger when at scale, sometimes by a significant amount. Avoiding temporary arrays allows us to achieve performance close to the peak bandwidth of the machine in some cases. For example, the ***linefit*** algorithm has to go over the input data twice. Given that each element is 16 bytes and the input is 500M elements, the total number of bytes that need to be read is 16 GB. Our machine has a peak read bandwidth of 140Gbytes/sec, which implies a peak performance of 16/140 = .114 seconds. We achieve about .145 seconds for both C++ and SML, which is reasonably close to the peak bandwidth. The time for the non-delayed version is more than twice slower, which corresponds to the extra read and extra write for each element it requires (write bandwidth is slower).

The space usage of the delayed version is often significantly less than the non-delayed version. The largest we measure is approximately 250x space reduction for ***integrate*** in both Parallel ML and C++ on 1 processor. This massive space reduction owes to the delayed version avoiding entirely the need to allocate one large intermediate array. On 72 processors, the space reduction is less, due primarily to additional allocations in the runtime system (in both Parallel ML and C++) which account for the additional parallelism.

There is one benchmark for which our delaying has little effect on space usage: ***sparse-mxv***. This is because the arrays being eliminated by delaying are tiny, around 100 items big, and therefore do not significantly increase space usage. The delayed version nevertheless achieves speedup thanks to reducing the number of writes and allocation of the memory for an inner map function.

6.3 Parallel ML vs. C++

Although not the focus of our comparison, it is worth noting that on 72 cores, the Parallel ML codes using our library (with both RAD and BID fusion) are often no more than 3x slower than their C++ counterparts. The only exceptions are ***tokens*** (4.3x), ***quickhull*** (3.7x), and ***wc*** (4.1x). Multiple benchmarks even perform within a factor of 2 (***bignum-add***, ***primes***, ***integrate***, ***linefit***, ***sparse-mxv***).

The performance gap on 1 core is more significant; we believe this gap is due to additional work for runtime checks (such as array bounds checks) and automatic memory management. As the number of cores increases, this additional work parallelizes well, causing the gap to decrease. In particular, the MPL compiler has a memory manager which benefits significantly from parallelism [2, 36].

6.4 Scalability

To study scaling trends, we present a few speedups plots in Figure 15. In these plots, the line marked "delay" is our full library (with both BID and RAD fusion), "rad" is the library with RAD-only fusion, and finally "array" is the array-based library with no fusion. The speedups (y-axis) are given with respect to the 1-processor time using the full library ("delay"). The number of processors is on the x-axis.

Overall, the delayed versions of the benchmarks scale noticeably better than their counterparts with lesser fusion. The primary reason for the better scaling in delayed versions is the reduced memory pressure afforded by delaying. Scaling is more pronounced in the MPL versions because the sequential MPL baseline is burdened with sources of overhead that are absent in C++, and some of these overheads parallelize very well, as discussed in Section 6.3. The relative steepness of the MPL curves shows the system effectively parallelizing away the costs associated with a high-level, mostly-functional programming language.

Block size	T	$\frac{T}{A}$	$\frac{T}{\text{Ours}}$
10^5	0.550	2.8	10
10^6	0.271	1.4	5.1
10^7	0.211	1.1	4.0
10^8	0.198	1.0	3.7

Figure 16. Times T (in seconds) of stream-of-blocks bestcut on 72 processors across different block sizes, and comparisons with array-based (A) and block-delayed (Ours).

6.5 Stream-of-blocks vs. blocks-of-streams

As described in Section 2.1, one prior block-based fusion technique is to represent a sequence as a stream of blocks. Our approach contrasts with stream-of-blocks by turning it "inside out", i.e., by instead using blocks of streams.

Here we present a small comparison of the two approaches by considering the ***bestcut*** benchmark. For this comparison, we implemented a stream-of-blocks version of the benchmark. As described in Section 3, this benchmark roughly has the structure of a map, followed by a scan, followed by another map, and finally a reduce. The stream-of-blocks version therefore maintains a small array (of size B, the block-size) which undergoes these operations, in that order, before then moving on to the next block. This continues iteratively until all blocks have been processed. All parallelism occurs within blocks, rather than across blocks.

In Figure 16, we present runtimes of the stream-of-blocks bestcut on 72 processors across a range of block sizes, and compare these times against both array-based sequences and our block-delayed sequences. First, we observe that the performance of the stream-of-blocks version is never better than the array-based version, and is at least 3.7x slower than with our block-delayed sequences. In fact, as the block size increases, the performance of the stream-of-blocks version improves, until it eventually matches the performance of the array-based version. This demonstrates that, for multicore (coarse) parallelism, the overhead of synchronization makes it difficult to exploit the parallelism available within the stream-of-blocks approach, especially for small block sizes.

7 Related Work

Fusion dates back to the 70s [1] and Allen and Cocke's seminal paper on compiler optimizations. This early work was interested in fusing actual loops, and not focused on collection-oriented languages. As such its emphasis was more on issues such as strides, offsets, and loops along different dimensions of an array [13, 21, 35]. Some of the work did allow the fusion of a "map" (loop with no dependencies), with a "reduce" (a loop which calculated a sum using an associative function), but not other functions on collections.

The first work we know of that focused on data-parallel languages was the work of Chatterjee et al. [9]. It is also the only work of we know that fuses both the first and third phase of the three-phase scan implementation The approach was applied to VCODE [6] which was the intermediate language for Nesl [4]. The compiler approach is quite sophisticated, requiring size-inference as a subcomponent. This approach was not applied to the `filter` or `flatten` functions.

Prior methods based on streams [16, 32, 34], indexing [20], and streams of blocks [12, 17, 24] were described in Section 2. In addition to the original work on delayed sequences [20], there has been follow-up work on addressing the challenge of clients reasoning about cost (e.g., determining whether or not a computation is delayed). To address this issue, Lippmeier et al. [22] propose making certain representation details explicit at the level of types. In our approach, such details are made explicit via a cost semantics (Section 5), enabling clients to perform a cost analysis. Other work extended the original delaying approach so that a map can be fused with the first phase of a scan [26], and so that some operations can fuse more efficiently by avoiding conditional branches in inner loops [33], which can be important for vectorization and GPU programming.

In the distributed setting, there has been work on fusion for operations such as map, reduce, scan, and filter [15, 25]. Our work differs from this prior work in multiple ways. First, data-parallel operations in our proposal may be nested, and our RAD representation allows for random access. Second, our approach utilizes stream fusion within blocks and therefore is able to significantly decrease the number of memory writes. For instance, in a reduce after a scan, our approach performs only $O(b)$ writes, where b is the number of blocks. Without stream fusion, $O(n)$ writes to memory (where n is the number of elements) would be needed.

8 Conclusion

We present a fusion technique for collection-oriented programming based on uniform blocks of (delayed) streams. This approach efficiently supports a wide range of common operations (including `scan`, `flatten`, and `filter`), and is well-suited for coarse granularities required on multicores. Our library implementations and experiments in both C++ and Parallel ML demonstrate that the technique is portable across multiple languages, offering significant performance improvements without needing any changes to the compiler. In future work, it would be interesting to adapt the technique for many-core (e.g. GPU) and distributed settings, where the cost of synchronization differs significantly.

Acknowledgements

We thank the anonymous reviewers for their comments and suggestions, and Umut Acar for many helpful discussions along the way. This work was supported by the National Science Foundation under grants CCF-1901381, CCF-1910030, CCF-1919223, CCF-2028921, CCF-2107241, CCF-2119352, and CCF-2115104.

References

[1] Frances E. Allen and John Cocke. 1971. *A Catalogue of Optimizing Transformations.* IBM Thomas J. Watson Research Center.

[2] Jatin Arora, Sam Westrick, and Umut A. Acar. 2021. Provably Space Efficient Parallel Functional Programming. In *Proceedings of the 48th Annual ACM Symposium on Principles of Programming Languages (POPL)"*.

[3] John W. Backus. 1978. Can Programming Be Liberated From the von Neumann Style? A Functional Style and its Algebra of Programs. *Commun. ACM* 21, 8 (1978), 613–641. https://doi.org/10.1145/359576.359579

[4] Guy E. Blelloch. 1992. *NESL: A Nested Data-Parallel Language.* Technical Report CMU-CS-92-103. School of Computer Science, Carnegie Mellon University.

[5] Guy E. Blelloch, Daniel Anderson, and Laxman Dhulipala. 2020. ParlayLib - A Toolkit for Parallel Algorithms on Shared-Memory Multicore Machines. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA).* https://doi.org/10.1145/3350755.3400254

[6] Guy. E. Blelloch and Siddhartha Chatterjee. 1990. Vcode: a data-parallel intermediate language. In *IEEE Frontiers of Massively Parallel Computation.* 471–480.

[7] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A recursive model for graph mining. In *SIAM SDM.*

[8] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. 2007. Data Parallel Haskell: A Status Report. In *Workshop on Declarative Aspects of Multicore Programming (DAMP).* 10–18.

[9] Siddhartha Chatterjee, Guy E. Blelloch, and Allan L. Fisher. 1991. Size and Access Inference for Data-Parallel Programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation PLDI).* 130–144.

[10] Siddhartha Chatterjee, Guy E. Blelloch, and Marco Zagha. 1990. Scan Primitives for Vector Computers. In *1990 ACM/IEEE Conference on Supercomputing (SC).* 666–675.

[11] E. F. Codd. 1970. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM* 13, 6 (June 1970), 377–387.

[12] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. 2007. Stream Fusion: From Lists to Streams to Nothing at All. In *ACM SIGPLAN International Conference on Functional Programming (ICFP).* 315–326.

[13] Alain Darte. 1999. On the complexity of loop fusion. In *IEEE Int. Conference on Parallel Architectures and Compilation Techniques (PACT).*

[14] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[15] Kento Emoto and Kiminori Matsuzaki. 2014. An automatic fusion mechanism for variable-length list skeletons in SkeTo. *International Journal of Parallel Programming* 42, 4 (2014), 546–563.

[16] Andrew Gill, John Launchbury, and Simon L. Peyton Jones. 1993. A Short Cut to Deforestation. In *Proc. Conference on Functional Programming Languages and Computer Architecture (FPCA).* https://doi.org/10.1145/165180.165214

[17] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. 2017. Futhark: Purely Functional GPU-Programming with Nested Parallelism and in-Place Array Updates. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).* 556–571.

[18] Kenneth E. Iverson. 1962. *A Programming Language.* Wiley, New York.

[19] Guy L. Steele Jr. and W. Daniel Hillis. 1986. Connection Machine LISP: Fine-Grained Parallel Symbolic Processing. In *ACM Conference on LISP and Functional Programming (LFP).* 279–297.

[20] Gabriele Keller, Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon L. Peyton Jones, and Ben Lippmeier. 2010. Regular, shape-polymorphic, parallel arrays in Haskell. In *ACM SIGPLAN international conference on Functional programming (ICFP).* ACM, 261–272.

[21] Ken Kennedy and Kathryn S. McKinley. 1993. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Int. Workshop on Languages and Compilers for Parallel Computing.*

[22] Ben Lippmeier, Manuel M. T. Chakravarty, Gabriele Keller, and Simon L. Peyton Jones. 2012. Guiding parallel array fusion with indexed types. In *ACM SIGPLAN Symposium on Haskell.* 25–36.

[23] J. David MacDonald and Kellogg S. Booth. 1990. Heuristics for ray tracing using space subdivision. *Vis. Comput.* 6, 3 (1990), 153–166. https://doi.org/10.1007/BF01911006

[24] Geoffrey Mainland, Roman Leshchinskiy, and Simon Peyton Jones. 2017. Exploiting vector instructions with generalized stream fusion. *Commun. ACM* 60, 5 (2017), 83–91.

[25] Kiminori Matsuzaki and Kento Emoto. 2009. Implementing fusion-equipped parallel skeletons by expression templates. In *International Symposium on Implementation and Application of Functional Languages.* Springer, 72–89.

[26] Trevor L. McDonell, Manuel M.T. Chakravarty, Gabriele Keller, and Ben Lippmeier. 2013. Optimising Purely Functional GPU Programs. In *ACM SIGPLAN International Conference on Functional Programming (ICFP).* 49–60.

[27] Eric Niebler, Casey Carter, and Christopher Di Bella. 2018. The One Ranges Proposal. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0896r4.pdf.

[28] John R. Rose and Guy L. Steele Jr. 1987. C*: An Extended C Language. In *Proceedings of the C++ Workshop. Santa Fe, NM, USA, November 1987.* USENIX Association, 361–398.

[29] J. T. Schwartz, R.B.K Dewar, E. Dubinsky, and E. Schonberg. 1986. *Programming with Sets: An Introduction to SETL.* Springer-Verlag, New York.

[30] Julian Shun, Guy E. Blelloch, Jeremy T Fineman, Phillip B Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. 2012. Brief announcement: the Problem-Based Benchmark Suite. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA).*

[31] Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. 2015. Generating Performance Portable Code Using Rewrite Rules: From High-Level Functional Expressions to High-Performance OpenCL Code. In *ACM SIGPLAN International Conference on Functional Programming (ICFP).* 205–217.

[32] Josef Svenningsson. 2002. Shortcut Fusion for Accumulating Parameters & Zip-like Functions. In *Proc ACM SIGPLAN International Conference on Functional Programming (ICFP).* https://doi.org/10.1145/581478.581491

[33] Bo Joel Svensson and Josef Svenningsson. 2014. Defunctionalizing Push Arrays. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Functional High-Performance Computing* (Gothenburg, Sweden) *(FHPC '14).* Association for Computing Machinery, New York, NY, USA, 43–52. https://doi.org/10.1145/2636228.2636231

[34] Philip Wadler. 1990. Deforestation: Transforming Programs to Eliminate Trees. *Theor. Comput. Sci.* 73, 2 (1990), 231–248.

[35] Joe Warren. 1984. A Hierarchical Basis for Reordering Transformations. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL).*

[36] Sam Westrick, Rohan Yadav, Matthew Fluet, and Umut A. Acar. 2020. Disentanglement in Nested-Parallel Programs. In *Proceedings of the 47th Annual ACM Symposium on Principles of Programming Languages (POPL)".*

[37] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65.

A Artifact Description

A.1 Overview

The artifact is a self-contained Docker[4] image containing all code and scripts necessary for reproducing our results. In particular, this includes source code for our library implementations of block-delayed sequences (in both C++ and Parallel ML) as well as all benchmarks used in evaluation, and also all experiment scripts. These are described in detail in Section A.7, below.

For evaluating the artifact, we provide two sets of instructions, one for a "small" evaluation, and the other for a "full" evaluation. The small evaluation (Section A.5) considers just a few benchmarks with reduced problem sizes, and takes about 5 minutes to run. The full evaluation (Section A.6) is intended for fully reproducing our results in the paper, and takes between 4.5 and 10 hours to run (depending on how much is reproduced).

A.2 Requirements

Running experiments requires the following.

- **Software**: Docker
- **Operating System**: Linux, macOS, or Windows. The provided setup instructions (Section A.4) should work on either Linux or macOS. All other instructions are run inside the container, and therefore should work regardless of underlying OS.
- **Hardware (small evaluation)**: Multicore server with at least 8 cores and 6GB of memory.
- **Hardware (full evaluation)**: Multicore server with a large number of cores (ideally ≥32) and 100GB of memory.

Results are written to the terminal and in PDF files (for speedup plots).

A.3 Getting the artifact

The artifact is available on Zenodo:

https://zenodo.org/record/5733288

All source code is also released on GitHub:

https://github.com/mpllang/delayed-seq

A.4 Setup

Step 1: Load the Docker image. If downloaded via Zenodo, the image can be loaded directly:

```
$ sudo docker image load
  -i shwestrick-ppopp22-artifact-image.tar.gz
```

If desired, the image is also available from Docker Hub:

```
$ sudo docker pull shwestrick/ppopp22-artifact
```

For the rest of the instructions, we assume the arifact is locally tagged `shwestrick/ppopp22-artifact`. This should be the default behavior from both of the above commands.

[4] www.docker.com

Step 2: Start the container. First, make a local directory `ARTIFACT-RESULTS` which will be mounted in the docker container (this lets us copy files out of the container). Then start the container as shown below. This opens a bash shell inside the container, which has the prompt #.

```
$ mkdir ARTIFACT-RESULTS
$ sudo docker run --rm --privileged
  -v $(pwd -P)/ARTIFACT-RESULTS:/ARTIFACT-RESULTS
  -it shwestrick/ppopp22-artifact
  /bin/bash
```

Note: the `--privileged` flag is necessary for NUMA control in the experiments.

A.5 Small Evaluation

Step 1: Run benchmarks. Run the following commands inside the container (the prompt inside the container is #).

```
# ./run-small
# ./report-small
  | tee /ARTIFACT-RESULTS/small-output
# cp -r small/figures
  /ARTIFACT-RESULTS/small-figures
```

Step 2: Check results. The output of the previous step consists of tables (printed to stdout, and copied to `ARTIFACT-RESULTS/small-output`) and a few speedup plots (copied to `ARTIFACT-RESULTS/small-figures`).

- The tables (`ARTIFACT-RESULTS/small-output`) are comparable to Figures 13 and 14 in the main paper, except with two important differences: the problem sizes are reduced by a factor 10, and only 8 cores are used (as opposed to 72 in the paper). The reported improvement ratios (R/Ours and A/Ours) therefore will not be exactly the same as in the paper, but should still generally be larger than 1, indicating improvement (speedup or reduced space usage).
- The speedup plots (`ARTIFACT-RESULTS/small-figures/*`) are named respectively:
  - `mpl-cc-XXX-speedups.pdf`: MPL (Parallel ML) results on benchmark `XXX`
  - `cpp-XXX-speedups.pdf`: C++ results on benchmark `XXX`

  The speedup plots should show that the `delay` version (our full library) scales consistently better than both the `array` (no fusion) and `rad` (RAD-only) versions. These are similar to Figure 15 in the paper, but on only a small number of cores. The speedups will not be as high due to the reduced problem size and smaller number of cores used.

A.6 Full Evaluation

Step 1: Generate inputs. Generating inputs should take approximately 2 minutes.

```
# ./generate-inputs
```

Step 2: Full experiments. Run the following commands inside the container.

```
# ./run --procs <PROCLIST>
# ./report | tee /ARTIFACT-RESULTS/full-output
# cp -r figures /ARTIFACT-RESULTS/full-figures
```

The `run` script takes an argument `--procs <PROCLIST>` which is a comma-separated (no spaces) list of processor counts to consider. We recommend choosing a maximum number of processors corresponding to physical cores, to avoid complications with hyperthreading. We also recommend choosing a range of intermediate processor counts, to see informative speedup curves.

For example, in our experiments we used a 72-core machine and the command `--procs 1,10,20,30,40,50,60,72`. With 32 cores, we recommend `--procs 1,10,20,32`. With 64 cores, we recommend `--procs 1,10,20,30,40,50,64`.

For reference, on our machine, it takes 4.5 hours to run the command `./run --procs 1,72`. This is the minimum required for reproducing Figures 13 and 14.

Step 3: Check results. Similar to the small evaluation, the tables produced (`ARTIFACT-RESULTS/full-output`) are comparable to Figures 13 and 14, and the speedup plots (`ARTIFACT-RESULTS/full-figures`) are comparable to Figure 15. If a large number of processors (≥64) were used, the reported improvement ratios (R/Ours and A/Ours) should be similar to those reported in Figures 13 and 14, modulo hardware differences and containerization overheads.

A.7 Reuse and Repurposing

The source code (library, benchmarks, and experiment scripts) used in the artifact can easily be adapted for other uses. All code is additionally available on GitHub (https://github.com/MPLLang/delayed-seq).

At a high level, there are two sets of source codes:

- `cpp-new/` contains all C++ code, and
- `ml/` contains all Parallel ML code.

In `cpp-new/`, each benchmark source is in a file named `BENCHMARK.VERSION.cpp` where `VERSION` is either `array` (no fusion), `rad` (RAD-only fusion), or `delay` (full fusion, i.e. both RAD and BID). The definition of the library codes are in `cpp-new/pbbsbench/parlaylib`, which is a checkout of the ParlayLib (https://github.com/cmuparlay/parlaylib) framework. The delayed sequences as described in the paper have been incorporated into this framework.

In `ml/`, the subdirectory `bench/` contains one folder for each version (array, rad, and delay) and these folders respectively each have a file `sources.mlb` for compilation. The definition of the library codes are in `ml/lib/`.

There are two primary makefiles: `cpp-new/Makefile` and `ml/Makefile`. The former has targets of the form `BENCHMARK.VERSION.cpp.bin` and the latter has targets of the form `BENCHMARK.VERSION.mpl-v02.bin`. When making a benchmark, the resulting binary is placed in a `bin/` subdirectory (`cpp-new/bin` and `ml/bin`).

Running C++ benchmarks. The cpp binaries all can be run with the following syntax, where `<N>` is the number of threads to use, `<ARGS>` are benchmark-specific arguments, `<R>` is the number of repetitions, and `<W>` is the length (in seconds) of the warmup period. The warmup is performed by running the benchmark back-to-back until the warmup period has expired. The benchmark is then run back-to-back for the number of repetitions specified.

```
[cpp-new/]$
  PARLAY_NUM_THREADS=<N>
  bin/<BENCHMARK>.<VERSION>.cpp.bin
  <ARGS> -repeat <R> -warmup <W>
```

Running Parallel ML benchmarks. The ML binaries are similar to the above, but with slightly different syntax:

```
[delayed-seq/ml]$
  bin/<BENCHMARK>.<VERSION>.mpl-v02.bin
  @mpl procs <N> --
  <ARGS> -repeat <R> -warmup <W>
```

Benchmark arguments. Many of the benchmarks (all except `quickhull`, `bfs`, and `grep`) take only a single size argument, `-n <SIZE>`. This makes it easy to test performance across a range of problem sizes. For example, here are the commands for running the C++ linefit benchmark (with `delay` version of the library, i.e. full fusion) on 16 cores across a range of problem sizes, with 10 repetitions and 3 seconds of warmup.

```
$ cd cpp-new
$ make linefit.delay.cpp.bin
$ PARLAY_NUM_THREADS=16 bin/linefit.delay.cpp.bin
  -n 1000000 -repeat 10 -warmup 3
$ PARLAY_NUM_THREADS=16 bin/linefit.delay.cpp.bin
  -n 500000000 -repeat 10 -warmup 3
```

And similarly for ML:

```
$ cd ml
$ make linefit.delay.mpl-v02.bin
$ bin/linefit.delay.mpl-v02.bin @mpl procs 16 --
  -n 1000000 -repeat 10 -warmup 3
$ bin/linefit.delay.mpl-v02.bin @mpl procs 16 --
  -n 500000000 -repeat 10 -warmup 3
```

Scripts. In the top-level folder, there are JSON files to specify the parameters used in our experiments. These specifications are passed to `scripts/gencmds` which produces "rows" of key-value pairs, where each row describes one experiment. Examples of keys include `config`, `tag`, `impl`, etc. The config is either `cpp` or `mpl-v02`, the tag is the benchmark name, the impl is the version of the library used, etc.

The output of `scripts/gencmds` is then piped into `scripts/runcmds` to produce results. See the `run` script for more detail.

Finally, the script `report` parses the results and produces tables and figures.