Parallel Block-Delayed Sequences




Sam Westrick
Mike Rainey
Daniel Anderson
Guy Blelloch

PPoPP 2022

Programming with Collections

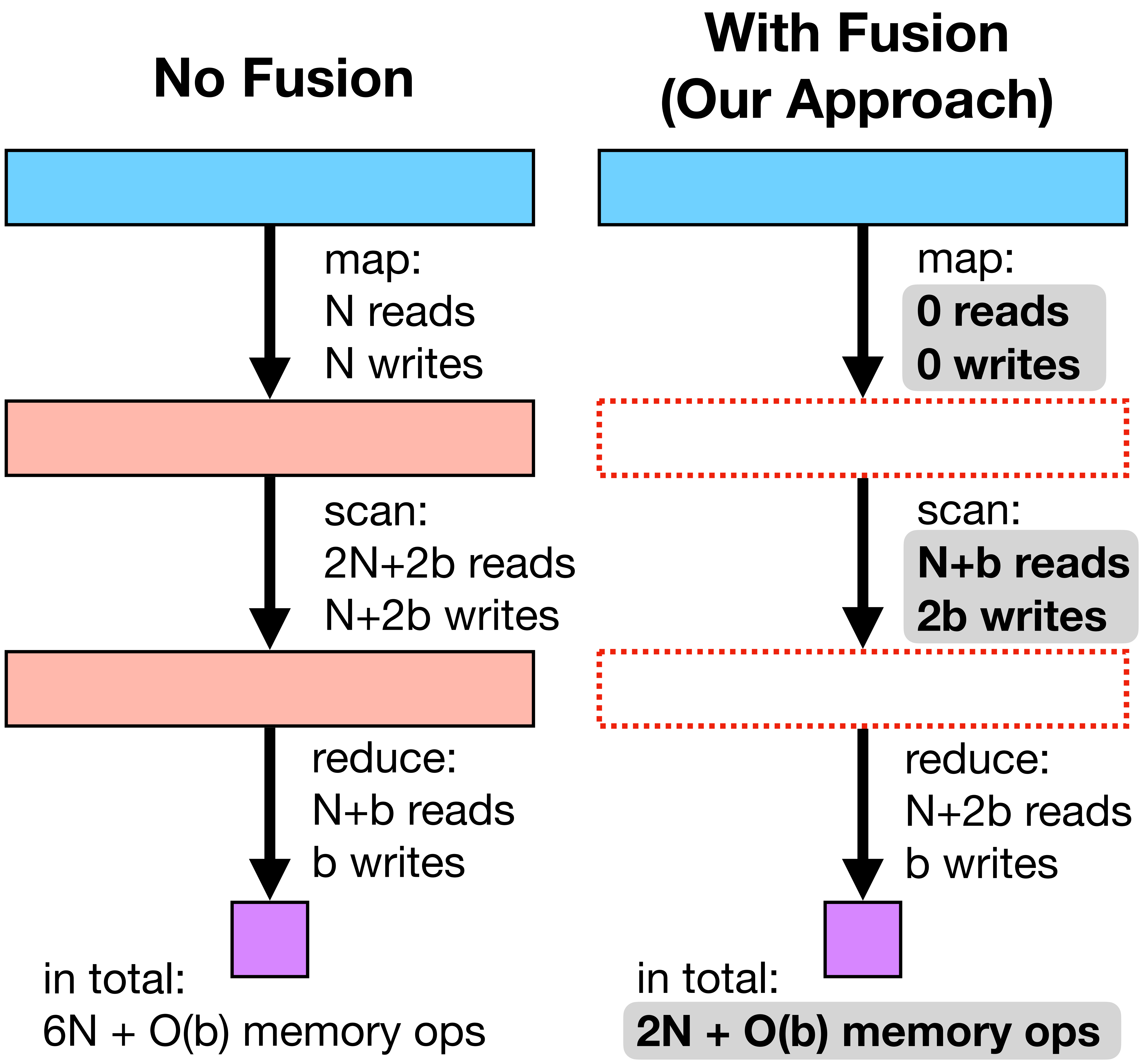
- sequences, sets, dictionaries, etc.
- map, reduce, filter, scan, etc.
- classic and popular
  - before I was born: APL, SETL, Backus's FP, CM-Lisp, C*, NESL, ...
  - nowadays, ubiquitous: MapReduce, Spark, Java Streams, Repa (Haskell), Futhark, NumPy (Python), MATLAB, Julia, LINQ (C#), ...
- **naturally parallel**
  - in terms of performance (bulk operations) and semantics (no concurrency by default)
  - functional style avoids race conditions
- succinct, easy-to-understand algorithms
  - abstract over algorithm design (e.g. divide-and-conquer ==> reduce)
  - higher-order functions

Efficiency?

- **standard problem**:
  excess writes for temporary (intermediate) results

- **solution: fusion**
  - optimize across operations
  - delay computation until results are needed

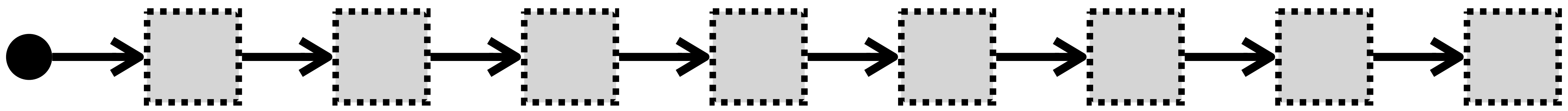
- for example:
  `map(f,map(g,S))`
  fuse
  `map(f∘g, S)`



Fusion Breakdown

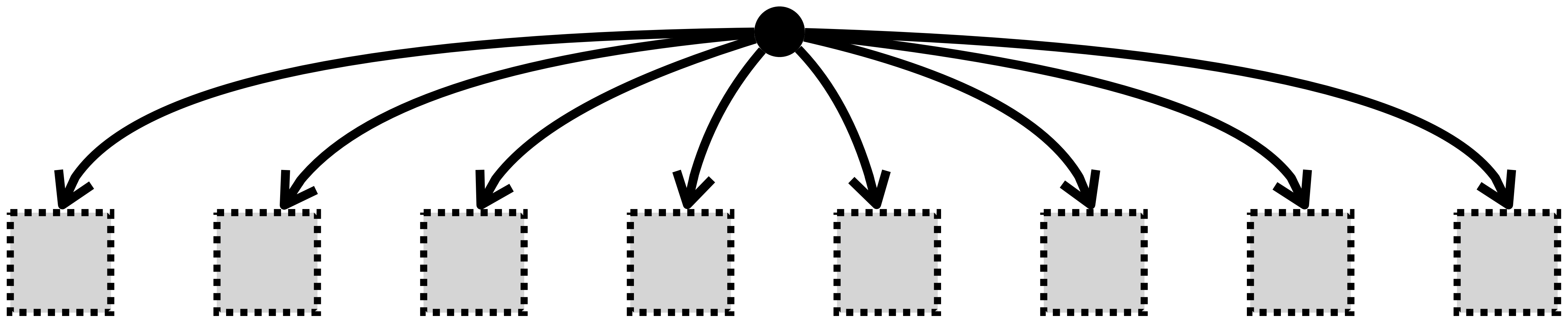
- **Stream fusion**
  - naturally sequential
  - e.g. lazy lists, Java streams, C++20 ranges/views, Rust iterators, ...

```
first: element
next: element → element
```



- **Index fusion**
  - naturally parallel
  - elements have to independent
  - good for map/zip/reduce fusion
  - e.g. Repa [1]

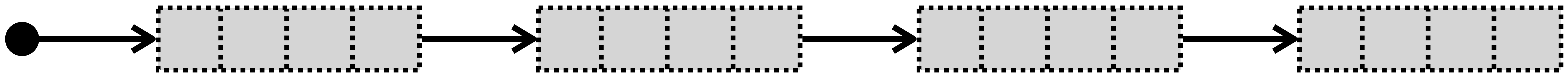
```
lookup: index → element
```



[1] **Regular, Shape-polymorphic, Parallel Arrays in Haskell.**
Gabriele Keller, Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Ben Lippmeier. ICFP 2010.

Fusion Breakdown (cont.)

- related work: **stream-of-blocks** [1,2]
  - parallelism within a block
  - stream fusion across blocks
  - well-suited for fine-grained SIMD
    - e.g. vectorized, GPU
      (can choose block size to match vectorization)
  - does not perform well on multicore
    - requires massive blocks to amortize synchronization



[1] **Futhark: Purely Functional GPU Programming with Nested Parallelism and In-Place Array Updates**.
Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. PLDI 2017.
[2] **Exploiting Vector Instructions with Generalized Stream Fusion**.
Geoffrey Mainland, Roman Leshchinskiy, and Simon Peyton Jones. CACM 2017.

Challenges

- **portability**
  - does it require integration with compiler?
  - does it rely on language-specific features?

- **fusion across wide set of parallel operations**
  - in addition to normal map/zip/reduce fusion, can it do:
    - scan (parallel prefix sums) ?
    - filter ?
    - flatten (`seq<seq<T>> → seq<T>`) ?

- **reasoning about performance**
  - where does fusion happen?
  - how many memory writes?

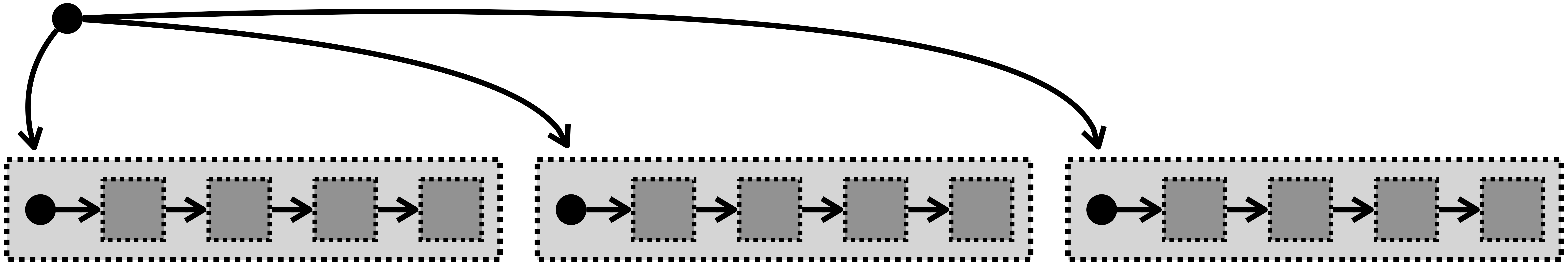
Block-Delayed Sequences

- **our approach: “blocks of streams”**
  - combine index- and stream-fusion
    - index fusion across blocks
    - stream fusion within blocks
  - well-suited for multicore hardware
  - fusion across scan, flatten, filter, etc.
  - simple cost model
    - work, span, memory writes (allocation)

requires no special compiler support or language features

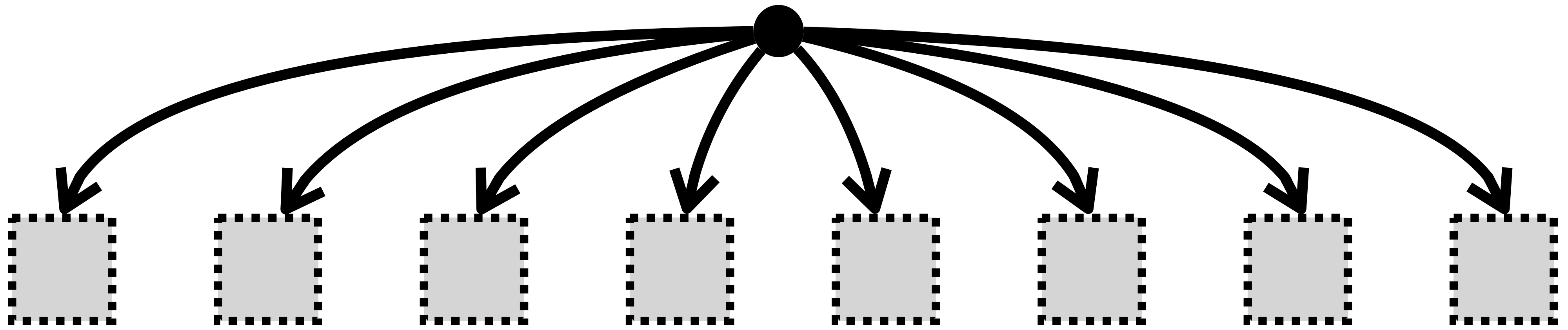
libraries implemented in two very different languages:

- C++
- Parallel ML (functional programming)



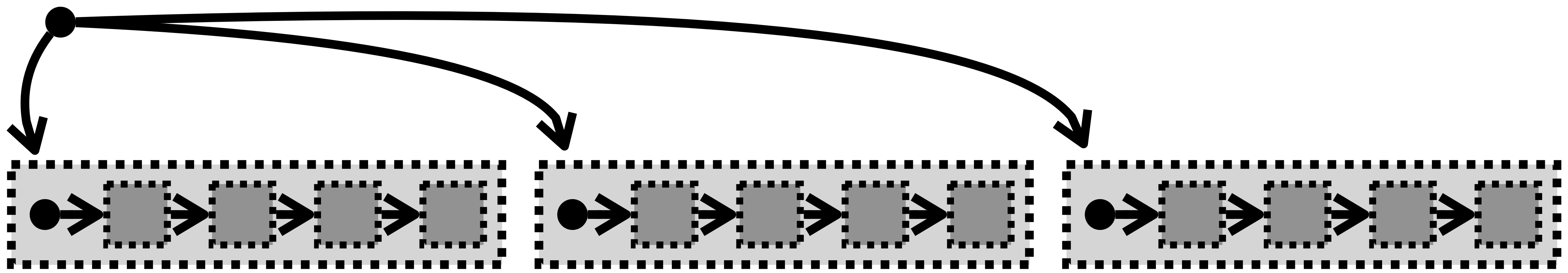
Block-Delayed Sequences

Random **A**ccess **D**elay
`RAD(i,n,f)`
`i: start index`
`n: length`
`f: index → element`



`[f(i), f(i+1), ..., f(i+n-1)]`

Blocked-**I**terable **D**elay
`BID(n,b)`
`n: length`
`b: block index → stream<element>`



n elements split into n/B blocks (block size B)
`b(i)`: stream of elements for i[th] block

RAD to BID: free

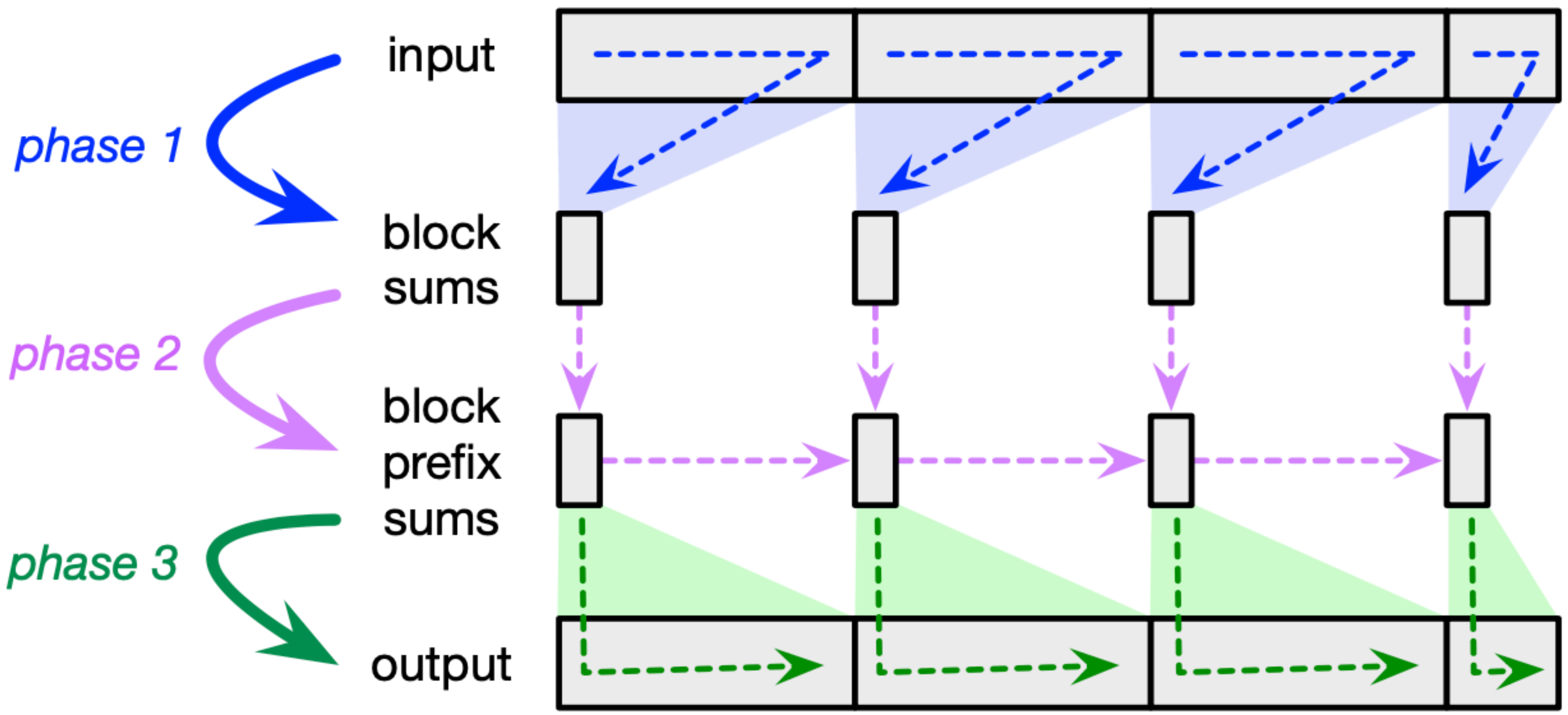
BID to RAD: O(N) writes

Block-Delayed Sequences: Scan

eager
convert input to BID
fuses with prior operations

eager

delayed
represent output as BID
fuses with later operations

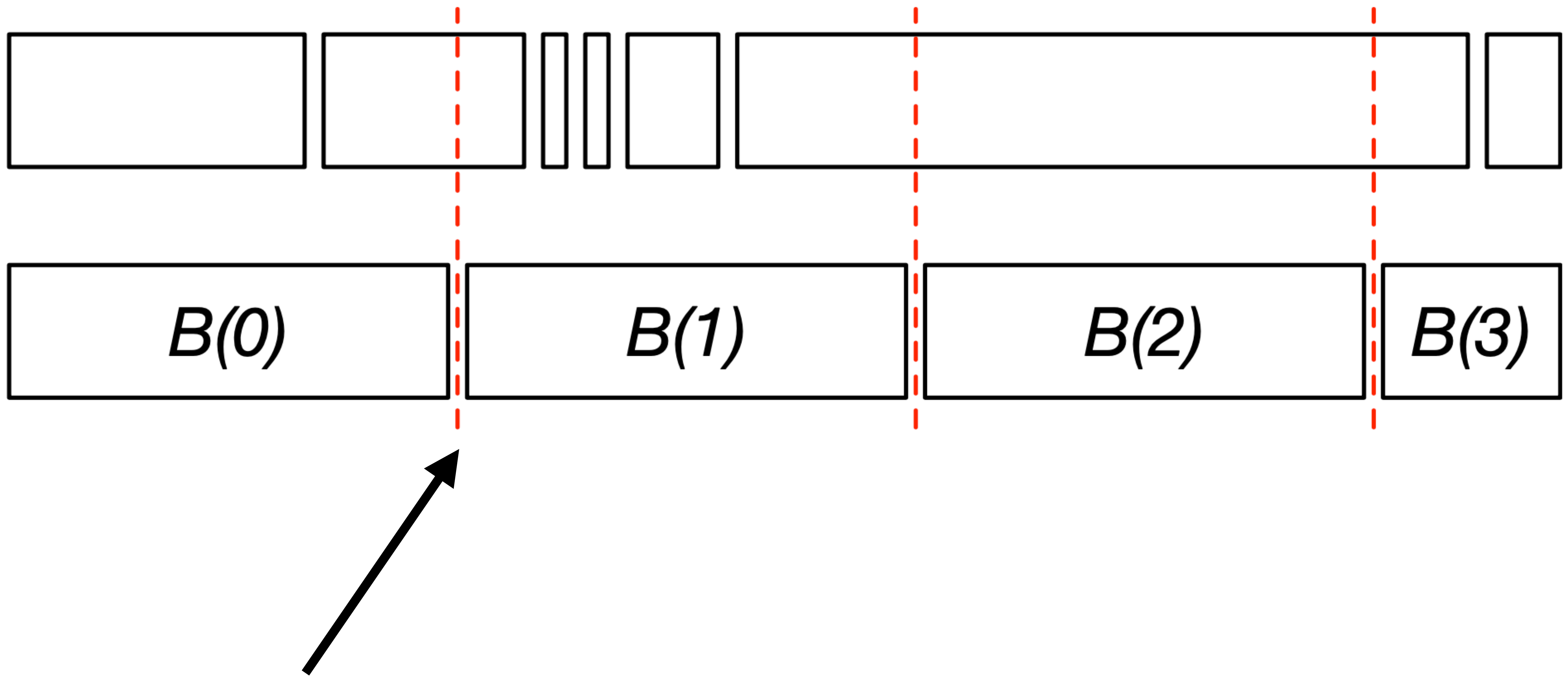


Block-Delayed Sequences: Flatten

(Filter is similar)

input
fuses with previous operations
(force outer elements to compute offsets; inner remain delayed)

output
represented as BID
fuses with later operations



block stream:

1. binary search (on length offsets) to find start
2. to compute next, advance pointer in subsequence or move to next subsequence

Example (BFS) and Cost Analysis

```
function nextFrontier(F):
  E = flatten(map(outEdges, F))
  F' = filter(tryVisit, E)
  return F'
```

Cost analysis (single round of BFS):
linear work
polylog span
only O(|F| + |F'| + |E|/B) memory writes

```
function outEdges(u):
  return map(fn(v) => (u,v), neighbors(u))

// visit v from edge (u,v); return v if success
function tryVisit(u,v):
  ...
```

Implementations

C++

- streams as stateful iterators
  - templated to specialize for a particular type
- overloading used to dispatch on sequence representation (BID vs RAD)
- updated PBBS benchmarks

Parallel ML (MPL)

- streams as stateful functions of type `unit → unit → 'a`. For example:

```
S = makeStream()
x0 = S(); x1 = S(); ...
```

- algebraic datatype for sequences, one variant per representation
  - standard compiler optimizations inline and specialize

ParlayLib+PBBS (C++)

github.com/cmuparlay/parlaylib
github.com/cmuparlay/pbbsbench

MaPLe Compiler (Parallel ML)

github.com/mpllang/mpl
github.com/mpllang/delayed-seq

Experimental Evaluation

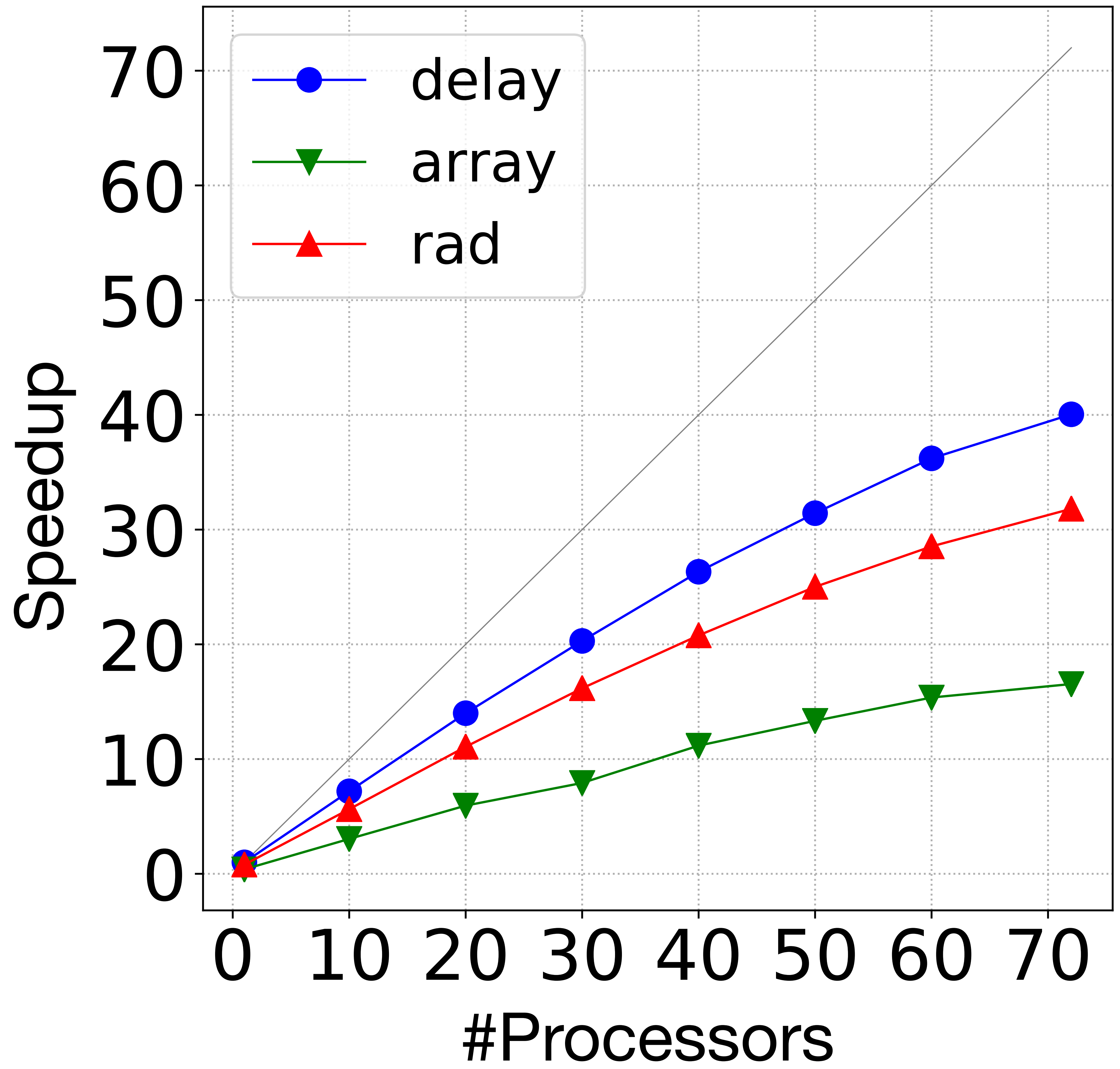
Three libraries compared:

- **array**: no fusion, arrays only
- **rad**: extends **array** with RAD fusion
- **delay** (full library): extends **array** with RAD+BID fusion

Six libraries in total
(Everything implemented in both C++ and Parallel ML)

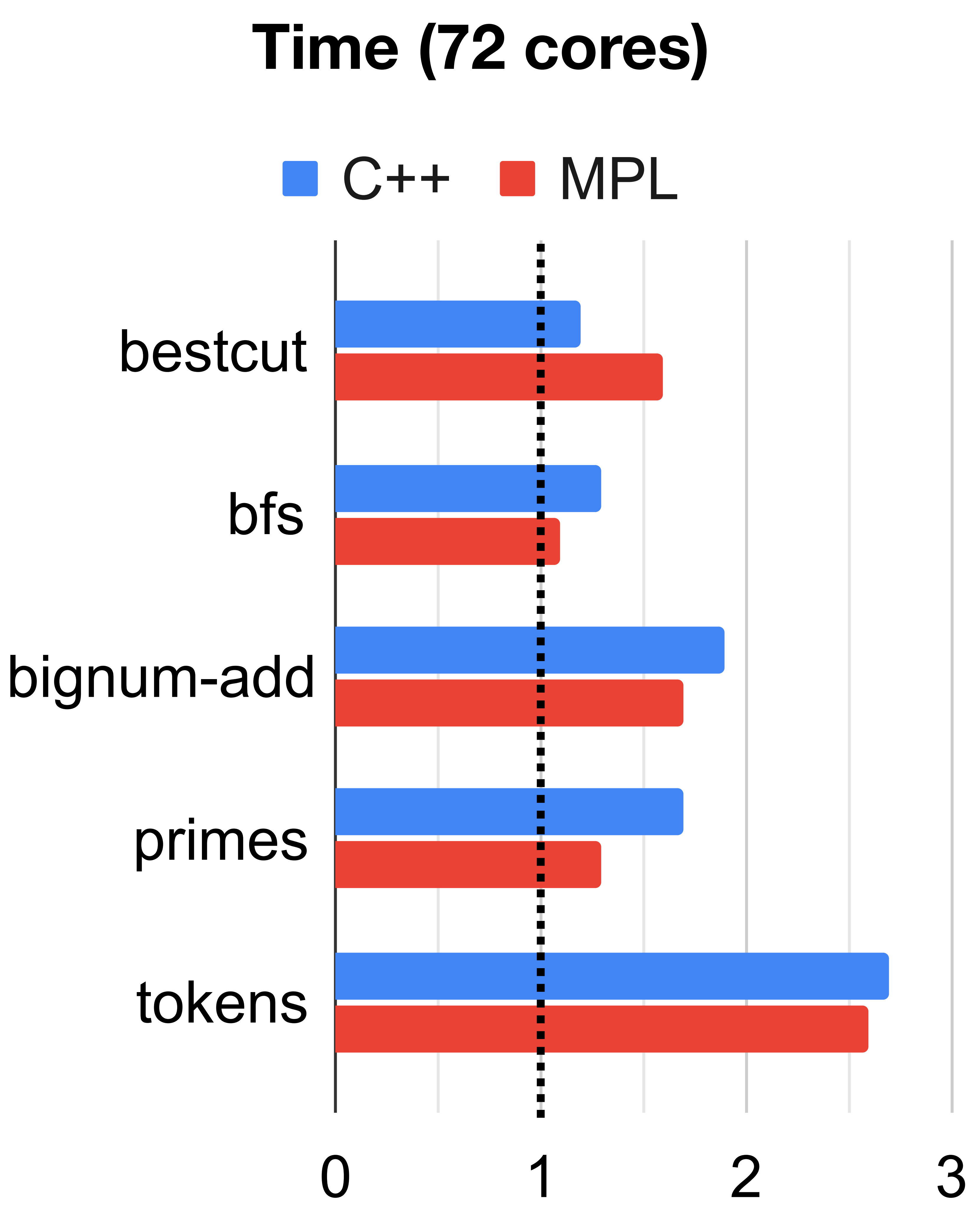
13 PBBS benchmarks

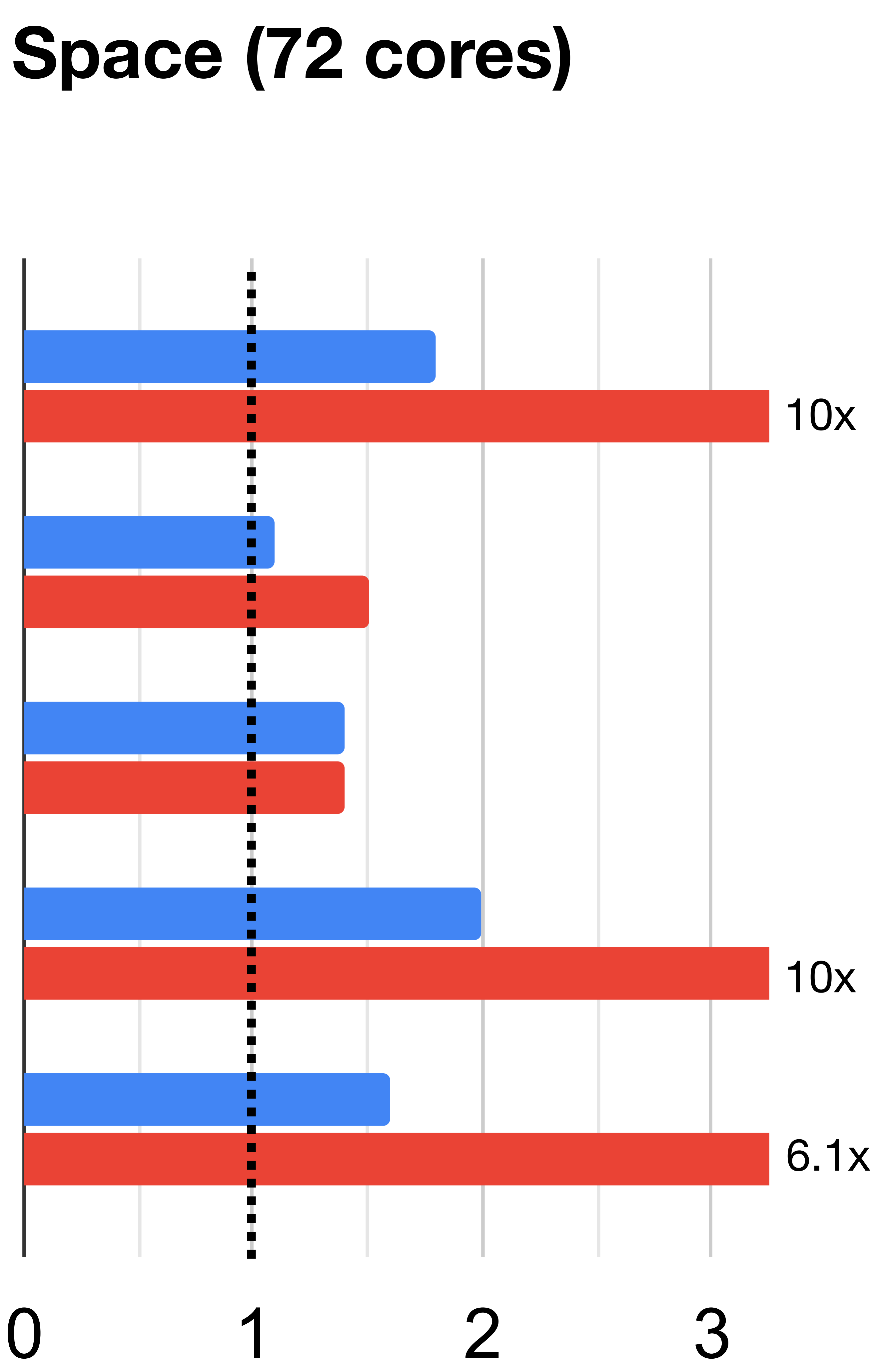
- 5 benefit from BID+RAD fusion
- 8 benefit from only RAD fusion



e.g.: C++ BFS Speedup

BID Fusion Improvement





delay (full RAD+BID fusion)

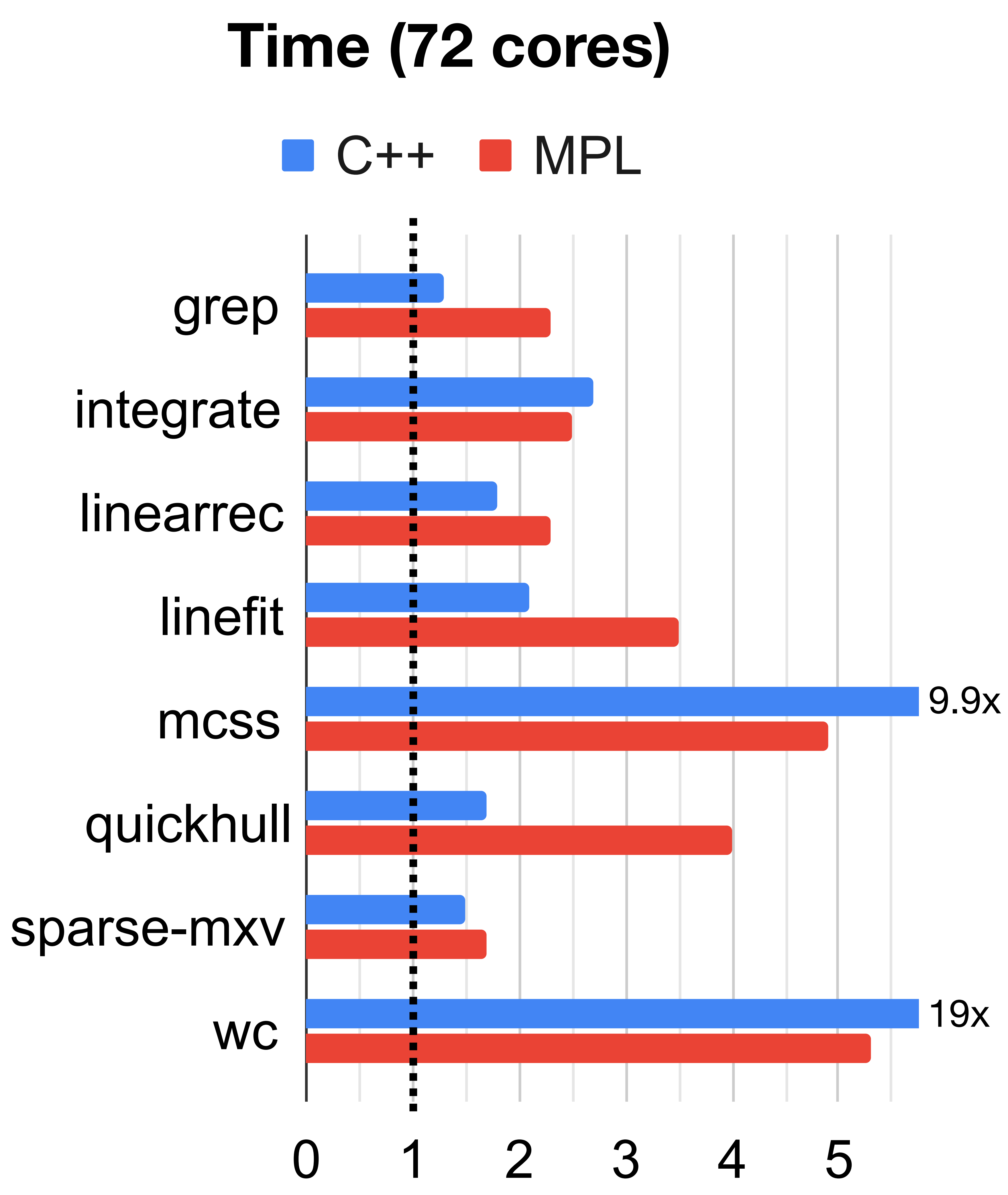
vs

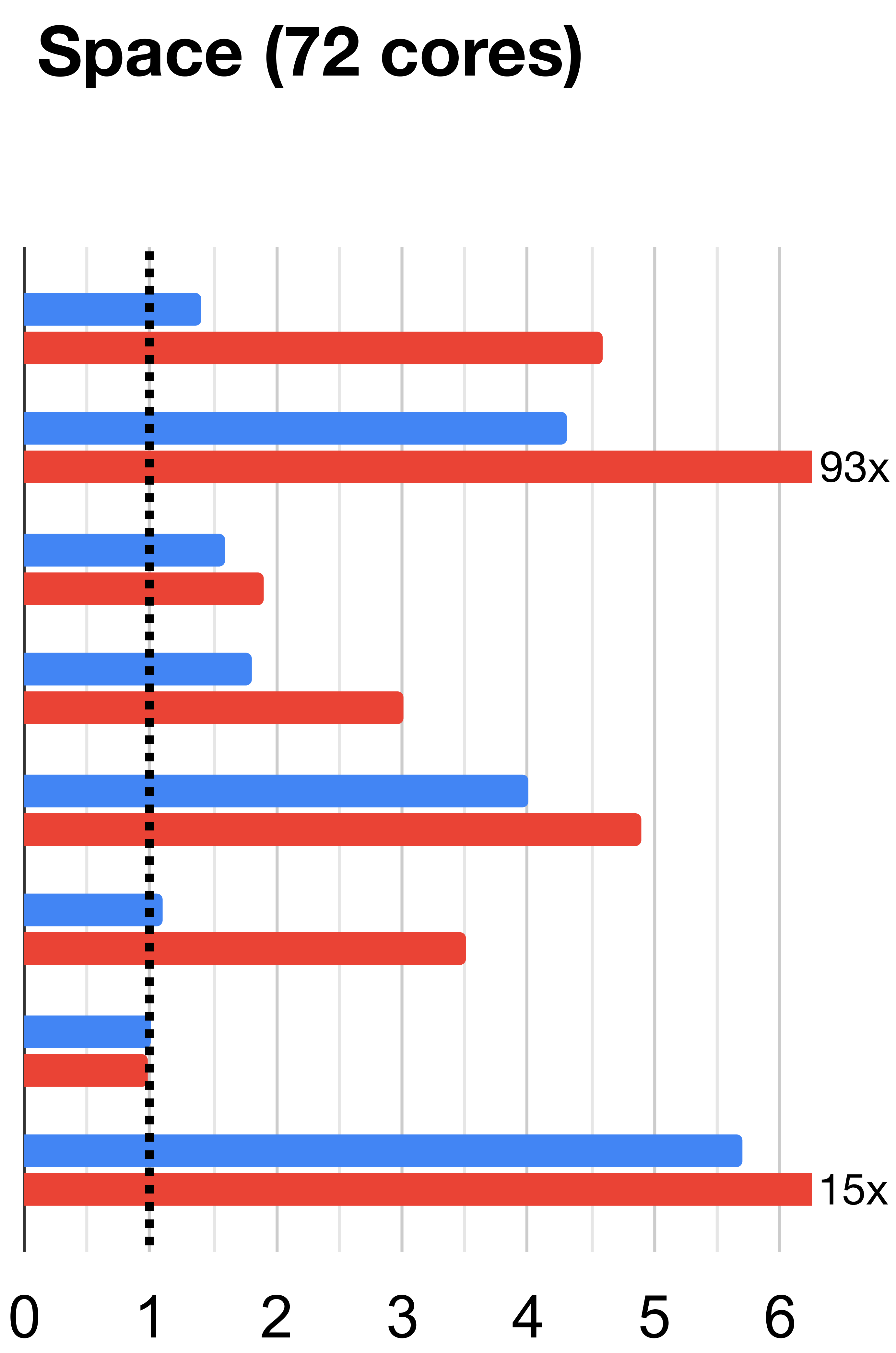
rad (RAD-only fusion)

up to 2.7x time improvement

up to 10x space improvement

RAD Fusion Improvement





delay (full RAD+BID fusion)

vs

array (no fusion)

up to 19x time improvement

up to 93x space improvement

Summary

parallel block-delayed sequences

- new “**blocks of streams”** representation (BID)
- supports fusion across all common operations (including scan, flatten, filter, etc.)
- simple to implement
- portable across multiple languages
  - implementations in both C++ and Parallel ML
- significant improvements in both space and time

future work

- extend to many-core and distributed computing

ParlayLib+PBBS (C++)

`github.com/cmuparlay/parlaylib`
`github.com/cmuparlay/pbbsbench`

MaPLe Compiler (Parallel ML)

`github.com/mpllang/mpl`
`github.com/mpllang/delayed-seq`