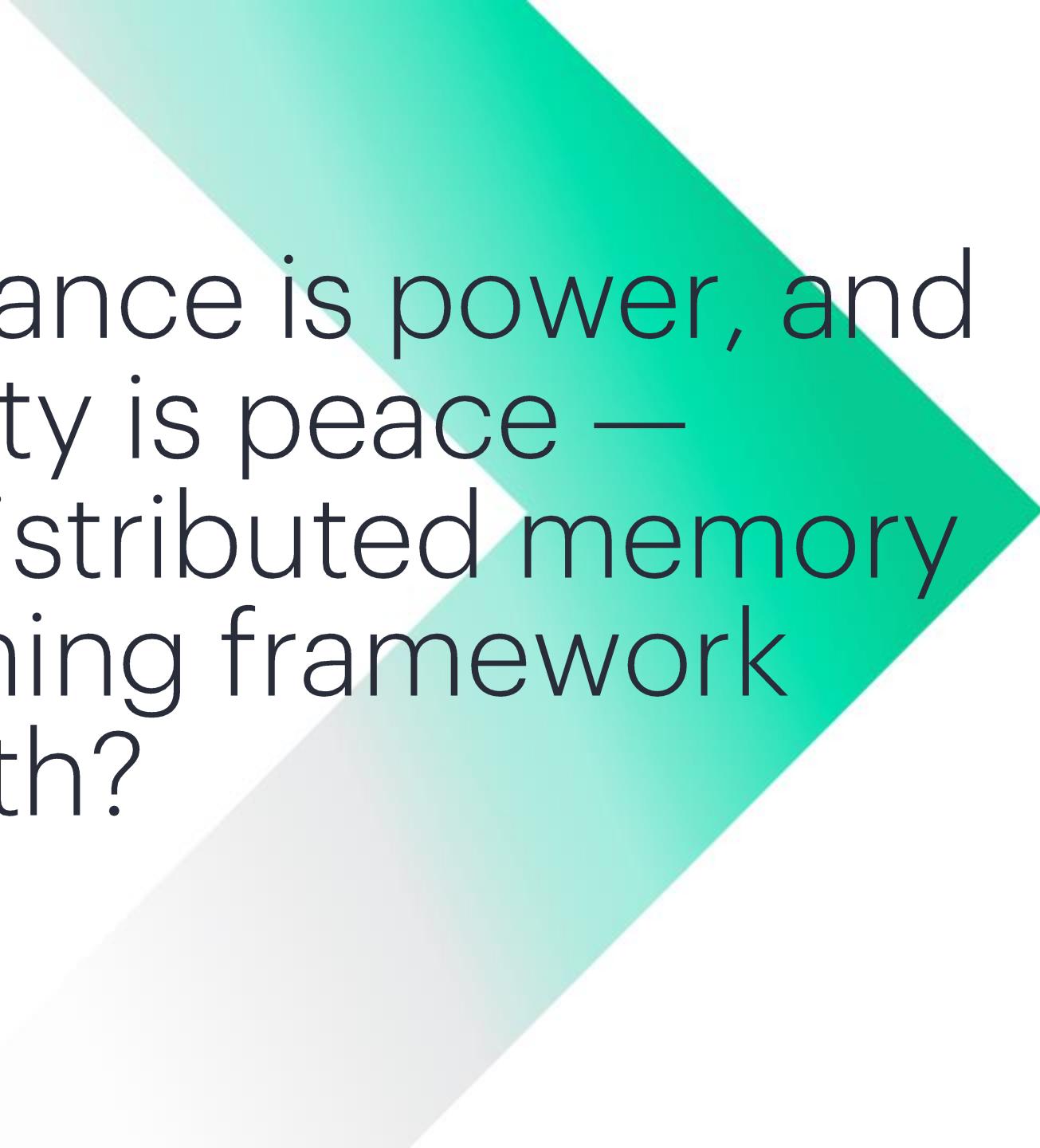


Comparing Distributed-Memory Frameworks: Performance and Productivity with Radix Sort

Michael Ferguson, Matt Drozt, Shreyas Khandekar, and Michelle Strout
Advanced Programming Team

Oct 10th, 2025



If performance is power, and
productivity is peace —
can one distributed memory
programming framework
deliver both?



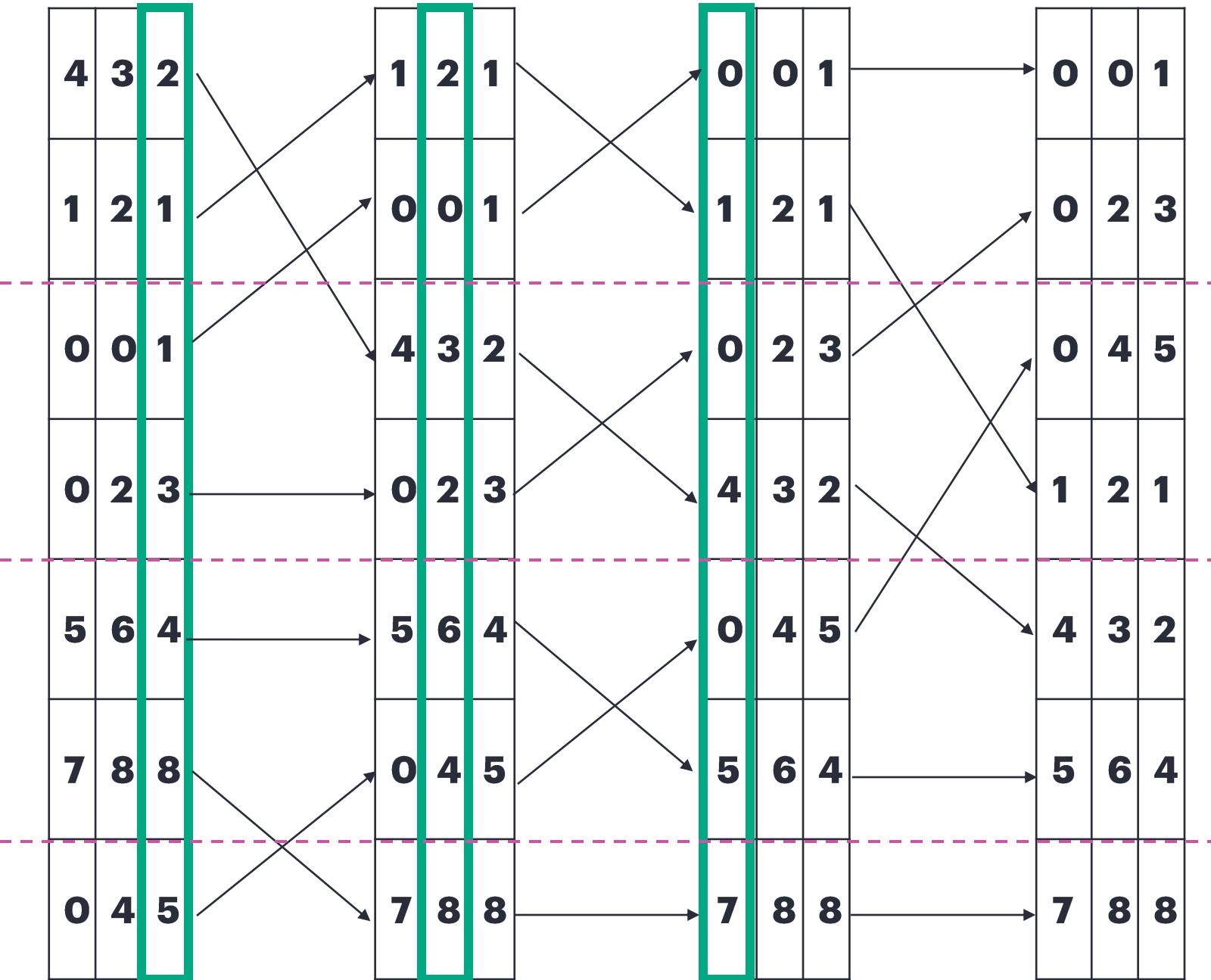
How This Study Came to Be

- We evaluated several **distributed-memory programming frameworks** using a **communication-intensive LSD radix sort** benchmark.
- The results and analysis are described in our paper:
“Comparing Distributed-Memory Programming Frameworks with Radix Sort”
submitted to the Parallel Applications Workshop, Alternatives to MPI+X (**PAW-ATM**), **SC’25**
- The following slides summarize the key findings from that work



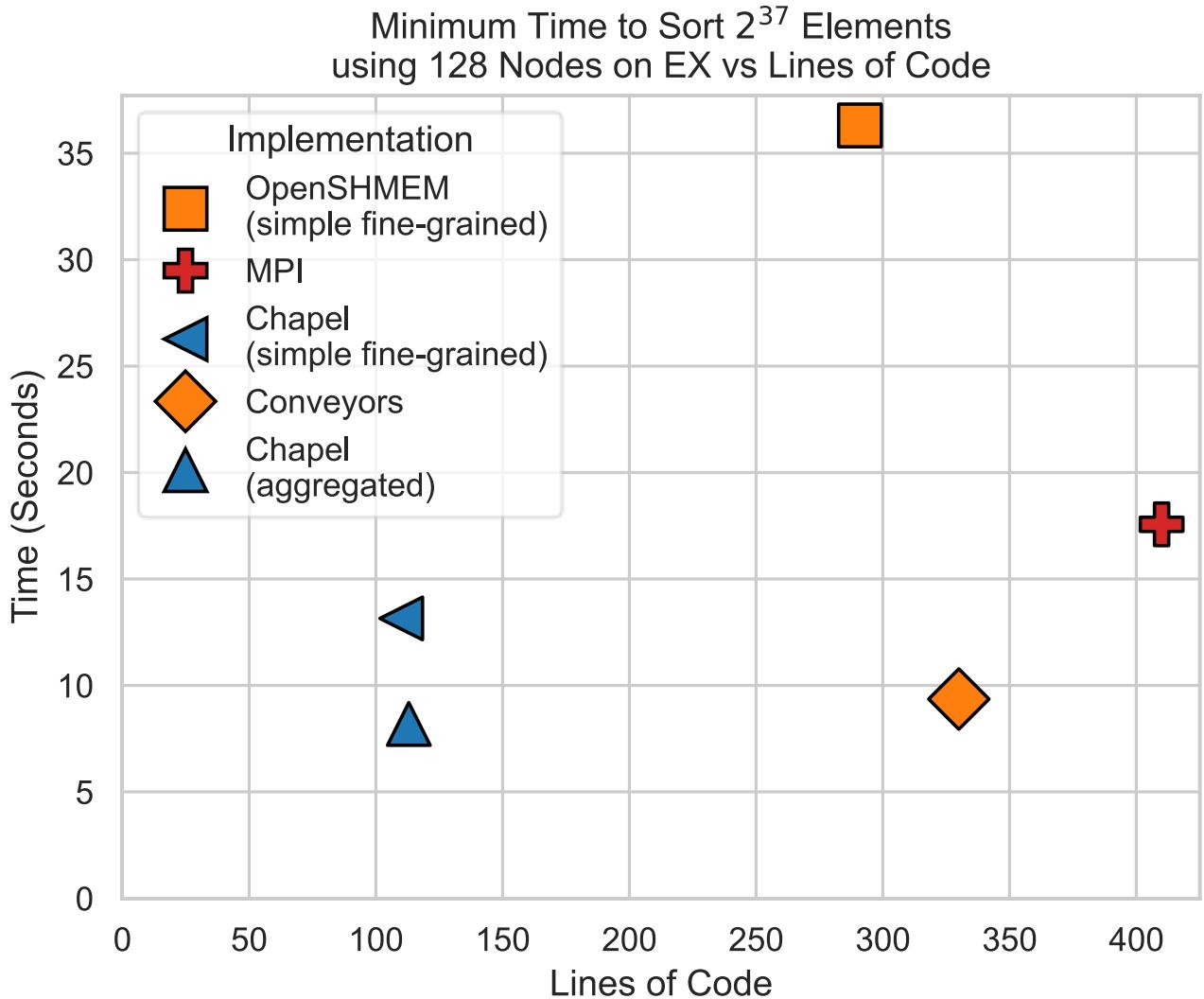
Experiment Overview

- Investigated five distributed programming frameworks on both HPE Cray EX (EX) and InfiniBand (IB) systems:
 - Chapel (with and without aggregation)
 - OpenSHMEM (with and without Conveyors for message aggregation)
 - MPI
 - Lamellar (currently only IB support)
- With each framework, we implemented a communication intensive algorithm: Distributed Least Significant Digit-First Radix Sort (LSD Sort)



Methodology

- **Development constraint:** ~1 day of implementation effort per framework by an experienced developer
- **Tuning:** Each LSD radix sort tested under multiple runtime configurations to identify optimal launch parameters
- **Benchmark:** Measured time to sort datasets of **128-bit key/value pairs** across varying data sizes and resource counts



Source Lines of Code and Performance Details

Framework	Version	Lines of Code	EX Sorting Performance ¹	IB Sorting Performance ²
--- Compared LSD Sorting Implementations ---				
Chapel	Simple, Fine-Grained	⭐ 110	10,455	182
Chapel	Aggregated	113	⭐ 16,782	⭐ 2,519
MPI	(AllToAll)	🔴 410	7,823	1,018
OpenSHMEM	Simple, Fine-Grained	291	🔴 3,786	🔴 48
Conveyors	Aggregated OpenSHMEM	330	14,687	1,323
Lamellar	Unsafe Array	185	--	475
Lamellar	Atomic Array	175	--	468

1. Sorting 2^{37} elements using 128 Nodes (M elements sorted / second)

2. Sorting 2^{34} elements using 32 Nodes (M elements sorted / second)



Chapel Builtins Promote Developer Productivity

We showed **multiple variants** with differing line counts and timings

Interestingly, Chapel saw a **1.6-13.8x** improvement with a small four-line change in the original source code

Other frameworks typically required larger code modifications or the use of additional layers to achieve comparable optimization

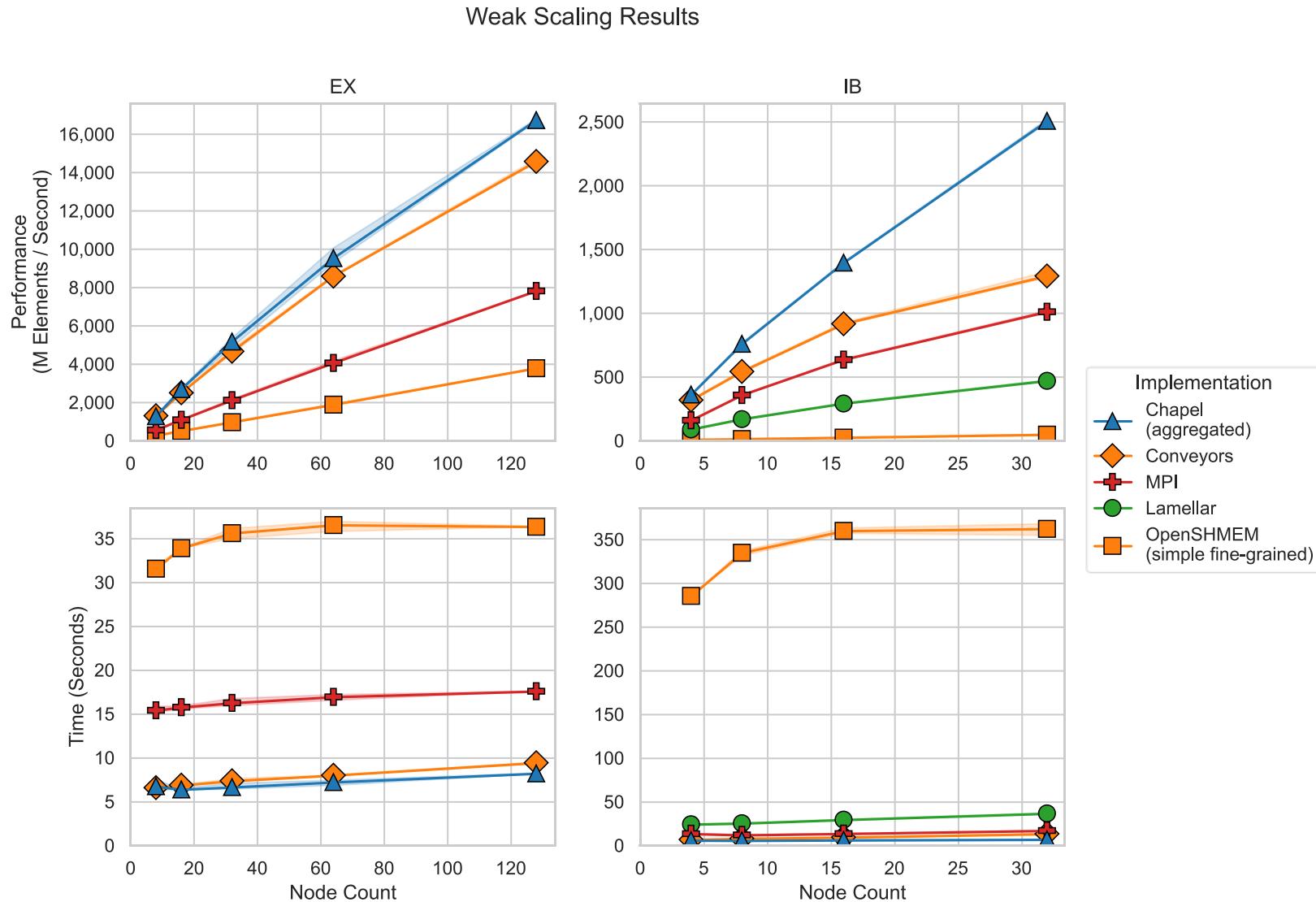
By contrast, Chapel's standard library offers several purpose built to allow for incremental refactoring for performance

The **aggregated Chapel implementation** is used as the representative version in this LSD radix sort comparison

```
+ use CopyAggregation;  
... // coforall task begin { ...  
var tD = calcBlock(task, 1D.low, 1D.high);  
// calc new position and put data there in temp  
{  
    var aggregator = new DstAggregator(t);  
    for i in tD {  
        const ref tempi = temp.localAccess[i];  
        const key = comparator.key(tempi);  
        var bucket = getDigit(key, rshift, last, negs);  
        var pos = taskBucketPos[bucket];  
        taskBucketPos[bucket] += 1;  
        aggregator.copy(a[pos], tempi); // a[pos] = tempi;  
    }  
    aggregator.flush();  
}  
} //coforall task
```



Weak-Scaling Results Across Frameworks



- Figure shows **weak-scaling performance** of the most performant LSD radix sort implementation per framework
 - Sorted 2^{30} elements per node on EX
 - Sorted 2^{29} elements per node on IB
- Under ideal weak scaling, doubling nodes and data size should **increase total throughput** while maintaining constant runtime
- All the LSD sort implementations showed good weak scaling
- The aggregated Chapel implementation achieved the **highest performance** on both systems

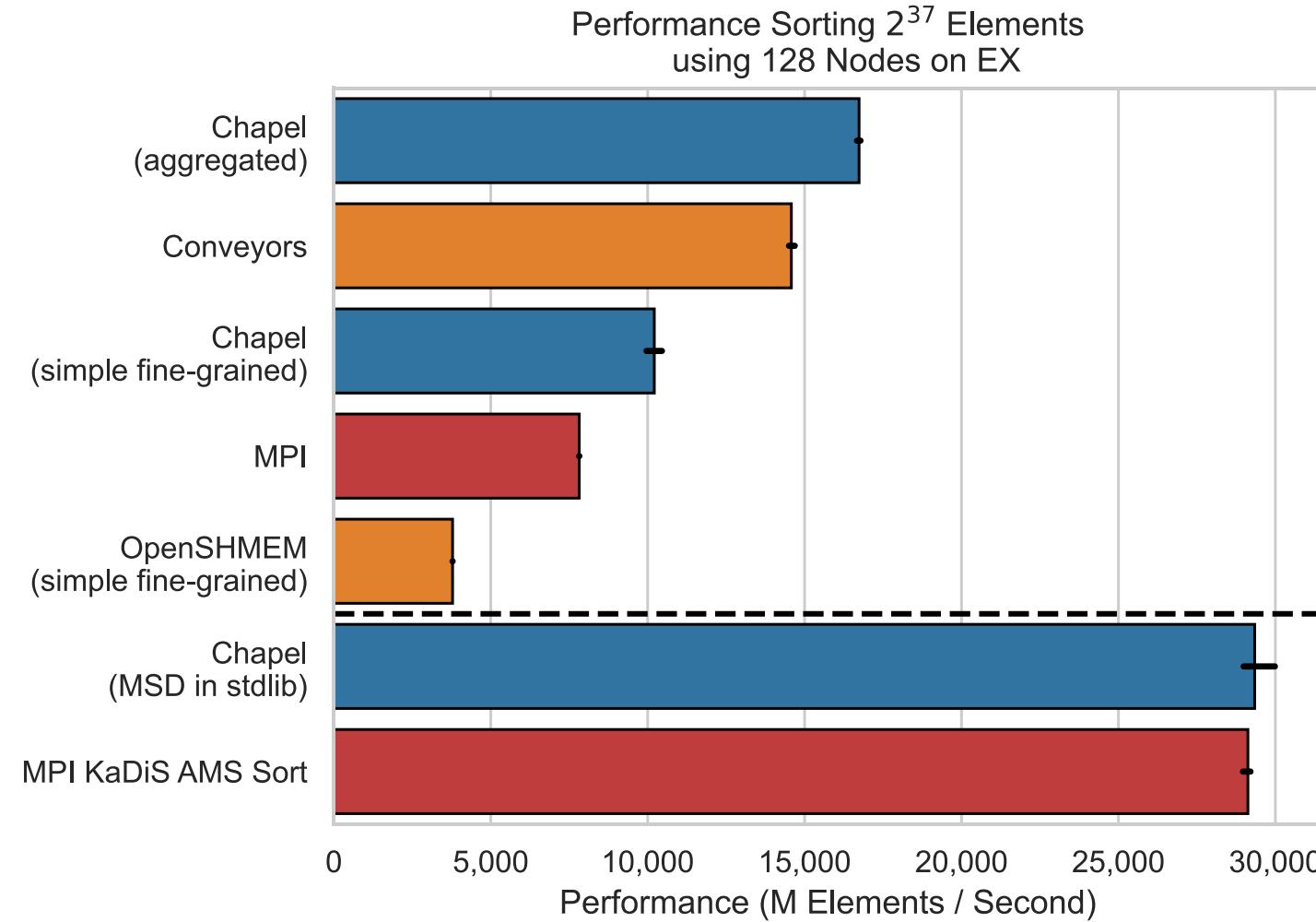
Source Lines of Code and Performance Details

Framework	Version	Lines of Code	EX Sorting Performance ¹	IB Sorting Performance ²
--- Compared LSD Sorting Implementations ---				
Chapel	Simple, Fine-Grained	⭐ 110	10,455	182
Chapel	Aggregated	113	⭐ 16,782	⭐ 2,519
MPI	(AllToAll)	🔴 410	7,823	1,018
OpenSHMEM	Simple, Fine-Grained	291	🔴 3,786	🔴 48
Conveyors	Aggregated OpenSHMEM	330	14,687	1,323
Lamellar	Unsafe Array	185	--	475
Lamellar	Atomic Array	175	--	468
--- Additional Sorting Implementations for Comparison ---				
Chapel	MSD Sort (Standard Library)	2,200	27,555	2,432
MPI	KaDiS AMS Sort	4,200	29,209	--

1. Sorting 2^{37} elements using 128 Nodes (M elements sorted / second)

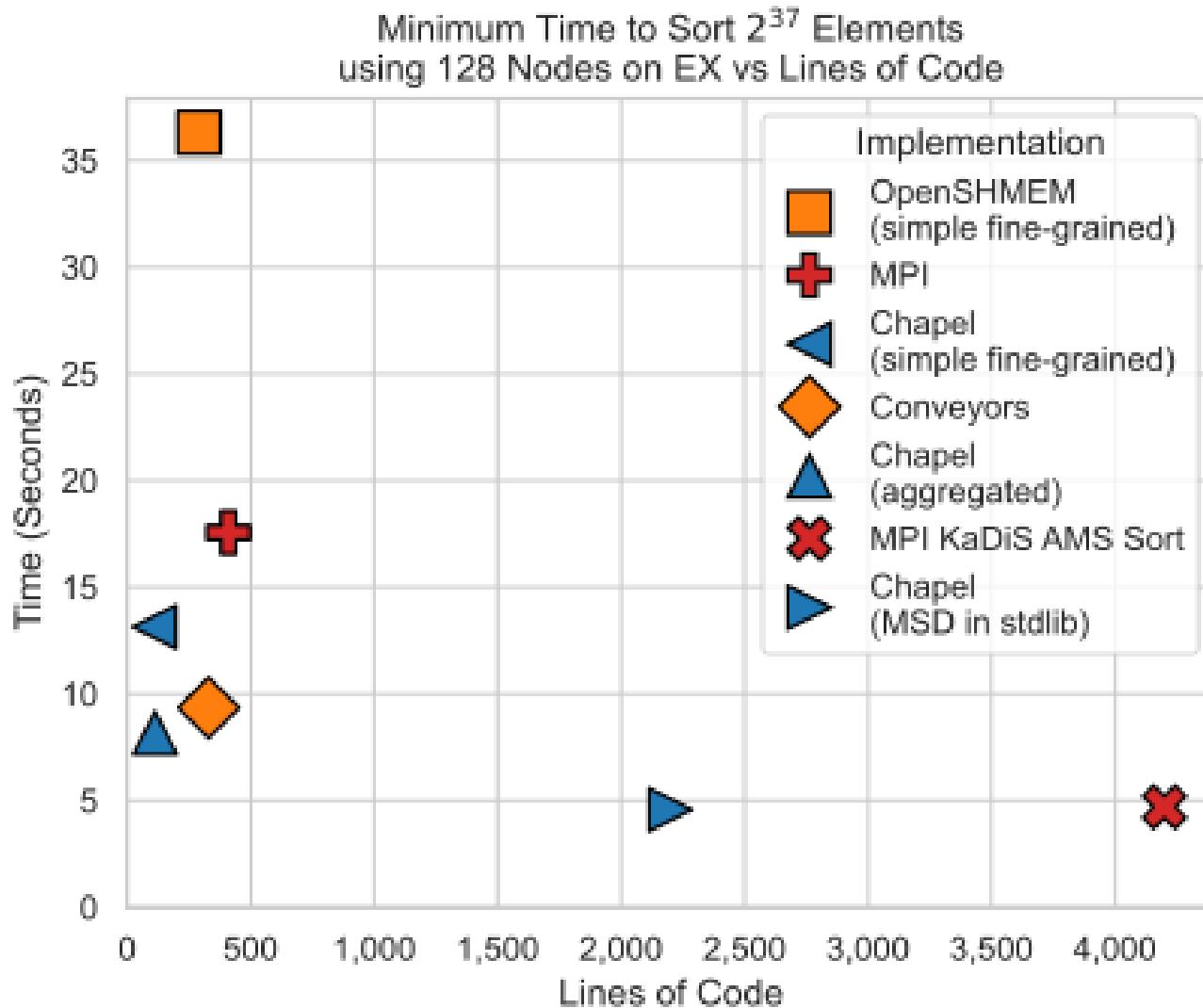
2. Sorting 2^{34} elements using 32 Nodes (M elements sorted / second)

Chapel Shows Superior Performance to Contemporaries at High Node Counts



- The bar plot shows the sort performance when sorting 2^{37} 128-byte elements when using 128 nodes on an EX system
- Bars shown above the dashed line are comparable implementations of an LSD Radix sort
 - Of these, the Chapel version utilizing aggregation was performed the best, approximately **4x** faster than OpenSHMEM and **2x** faster than MPI
 - The Chapel and Conveyors implementations benefit from message aggregation, one-sided communication, and communication overlap
- Bars shown below the dashed line are other, more complex, distributed sorting algorithms for comparison
 - Chapel is capable of being as performant as other more sophisticated distributed sorting algorithms

Chapel Offers Combined Performance and Productivity



- This scatterplot compares
 - time it takes each sorting implementation to sort 2^{37} elements using 128 nodes on an EX system
 - and the number of source lines of code as counted by `cloc`
- Optimal frameworks will approach the origin in the lower left: fewer lines of code indicate increased developer productivity and shorter run times indicating high performance
- Of the LSD sort implementations, the aggregated Chapel version is the most performant while containing **~4x fewer lines of source code** as compared to MPI
- Shorter codes benefited from native support for distributed arrays and parallel aggregation, aiding developer productivity

A Qualitative Comparison of Examined Frameworks

MPI

- **Pro:** The industry standard, very large developer community
- **Con:** Missing primitives that developers may expect in other frameworks

Chapel

- **Pro:** Concise ability to create parallel tasks across cores and nodes across a supercomputer
- **Pro:** Documentation and extensive standard library aid in parallel programming

OpenSHMEM

- **Pro:** Very easy to install
- **Pro:** Fast compile times
- **Con:** Uses a custom collective allocator, limiting scalability

Lamellar

- **Pro:** Very easy to install, follows the standard installation path of most Rust crates
- **Pro:** Familiar API to seasoned Rust developers
- **Pro:** Descriptive (albeit potentially cryptic to newcomers) error messages aid development by catching bugs at compile time
- **Con:** Currently only supports InfiniBand systems
- **Con:** Long compile times (typical of Rust projects)

Contribute to this Study

- The source code for each of the LSB-Radix sort implementations is open source and available on GitHub
- Repository can be found at:
<https://github.com/hpc-ai-adv-dev/distributed-lsb/>
- Contributions are both welcome and extremely appreciated
- We plan to make this a “living study repository” and hope that as others find or submit optimized versions of LSB-Radix sort we can keep this repository up to date with later findings
- Tell us what we did wrong!! 😊



Thank You – Happy Sorting

Shreyas Khandekar – shreyas.khandekar@hpe.com

Matt Drozt – drozt@hpe.com

Michael Ferguson – michael.ferguson@hpe.com

Ryan Friese – ryan.friese@pnnl.com