Comparing Distributed-Memory Programming Frameworks with Radix Sort

Michael Ferguson (HPE), Ryan Friese (PNNL), Shreyas Khandekar (HPE), Matt Drozt (HPE)

November 16th, 2025

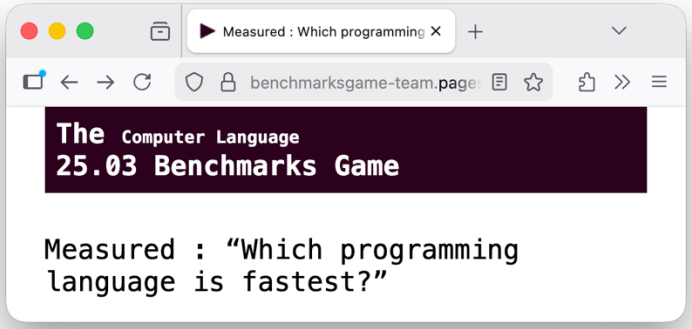
Comparing Distributed-Memory Programming Frameworks

We see a need to compare distributed-memory programming frameworks

- To help programmers choose appropriate tools for their tasks
- To show which problems each framework excels at
- To compare both productivity and performance

Inspiration: the Computer Programming Languages Benchmark Game

- Anybody can contribute improved programs in any of the languages
- Website includes regularly updated performance results



https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html

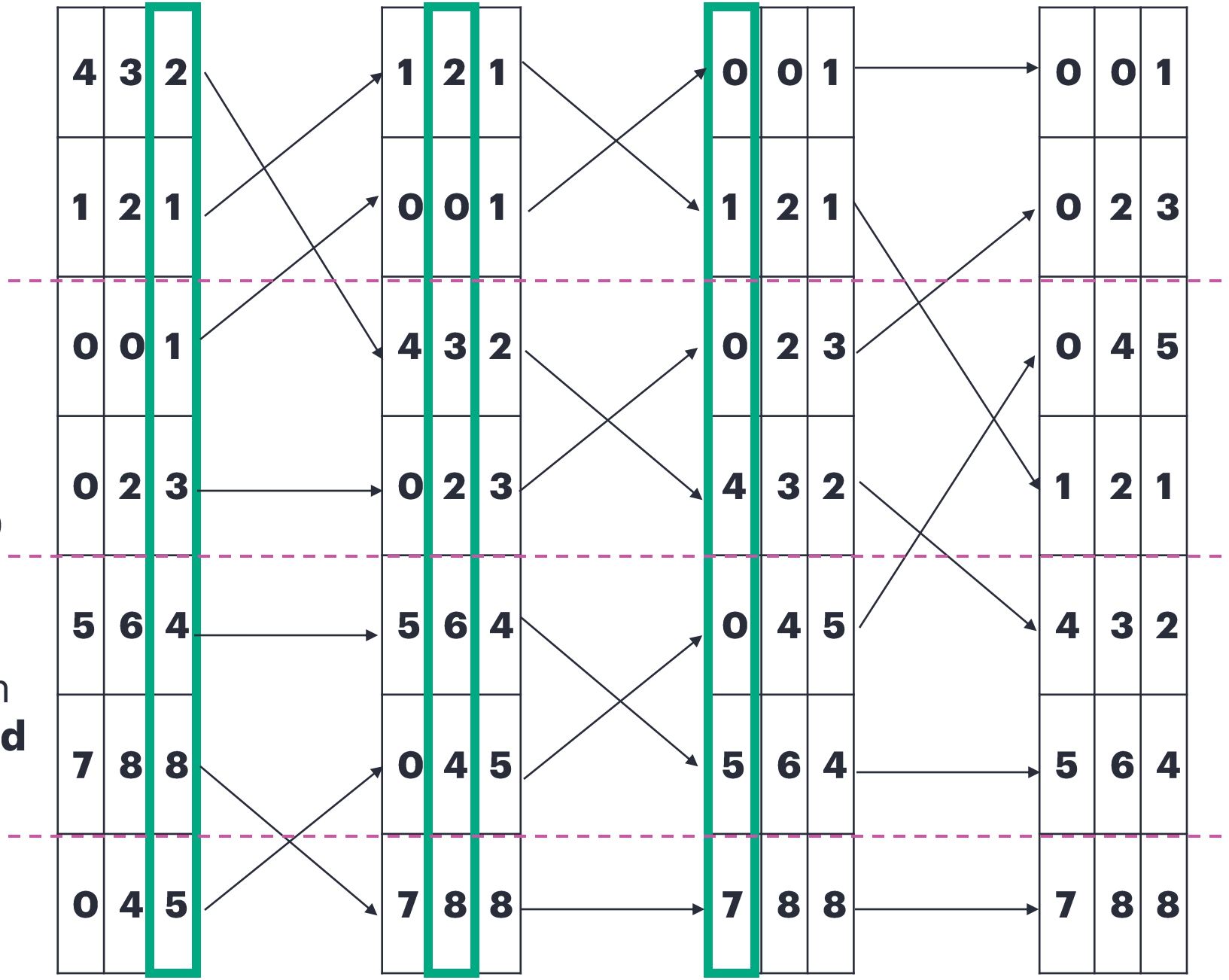
To keep the scope manageable, we focused on a single benchmark

New benchmarks & implementations are welcome in our repository!

- https://github.com/hpc-ai-adv-dev/distributed-lsb/

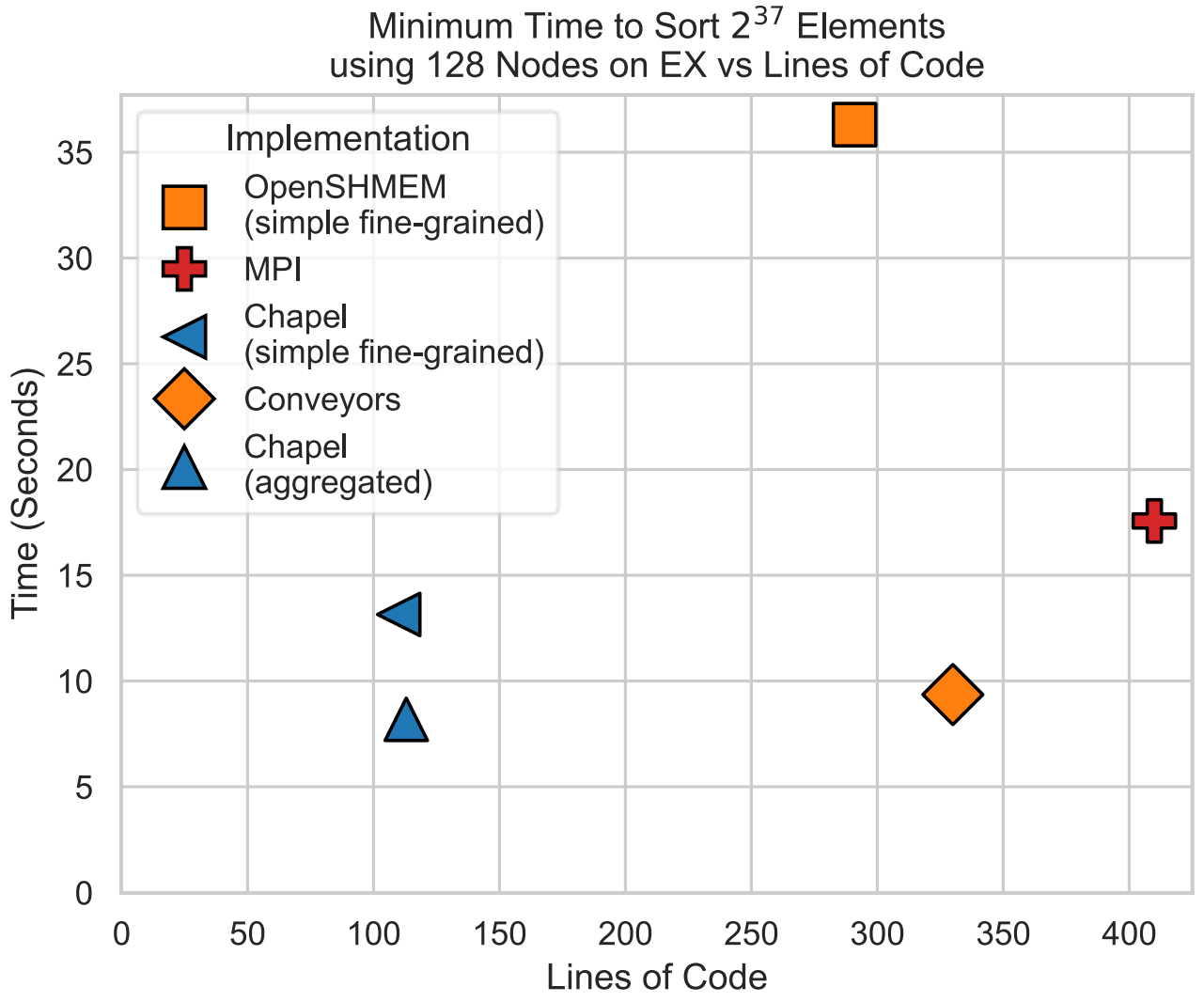
Experiment Overview

- 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)
- Implemented a communication intensive algorithm: **Distributed Least Significant Digit-First Radix Sort** (LSD Sort)



Methodology and Results

- **Development constraint:** ~1 day of implementation effort per framework by an experienced developer
- **Tuning:** Tested several configurations to identify optimal launch parameters
- **Benchmark:** Measured time to sort datasets of random **128-bit key/value pairs**



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	(AlltToAll)	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)

The 4-Line Change in the Chapel Version

Chapel saw a **1.6-13.8x** improvement ...
with a small four-line change

```
// coforall task begin { ...
    var tD = calcBlock(task, lD.low, lD.high);
    // calc new position and put data there in temp
    {


        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;
            a[pos] = tempi;
        }


     }
} //coforall task
```

The 4-Line Change in the Chapel Version

Chapel saw a **1.6-13.8x** improvement with a small four-line change

```
+ use CopyAggregation;
...     // coforall task begin { ...
            var tD = calcBlock(task, lD.low, lD.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
```

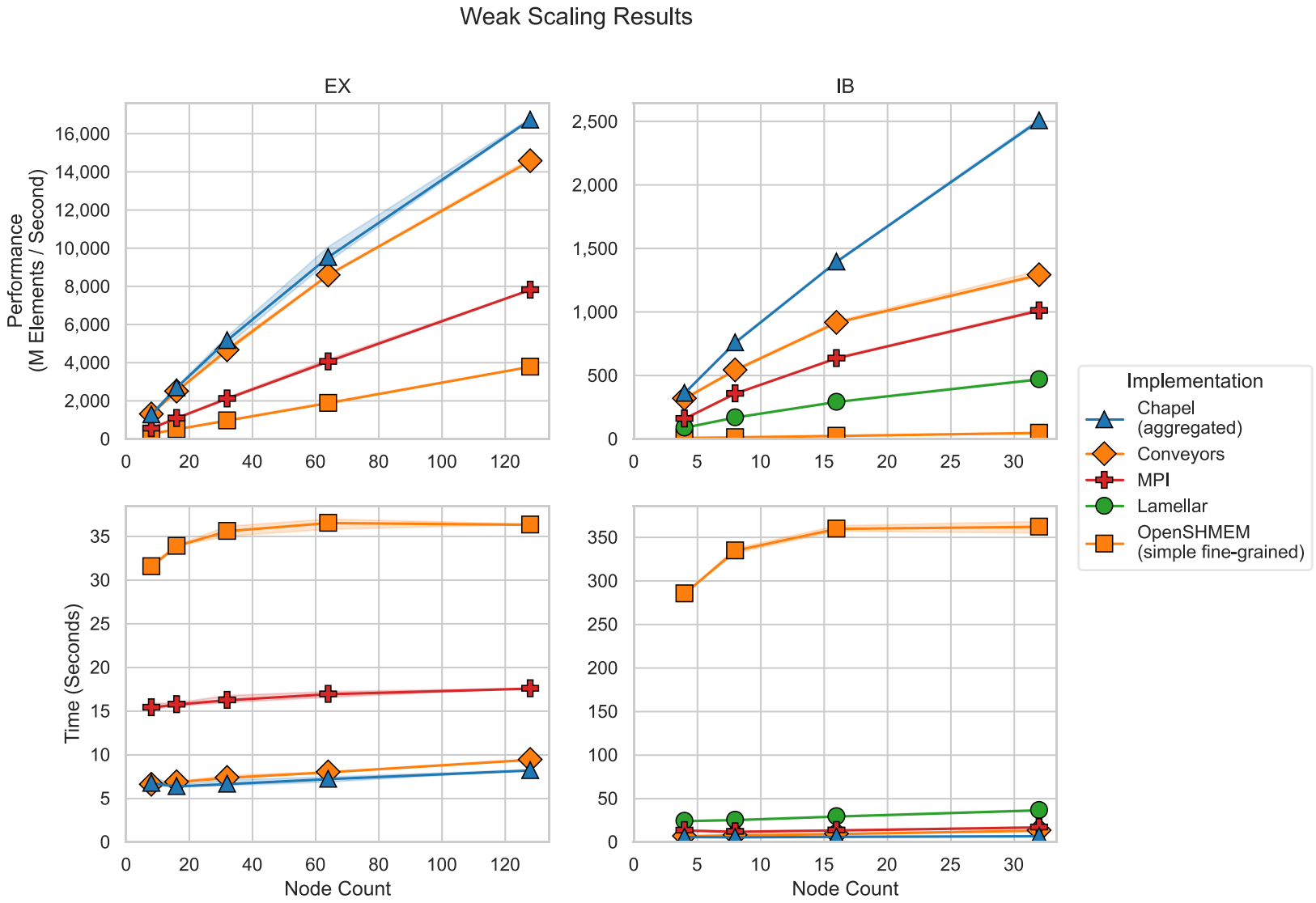
What Changed From OpenSHMEM to Conveyors? One part:

```
for (int64_t i = 0; i < locN; i++) {




  SortElement elt = localPart[i];
  int bucket = getBucket(elt, digit);
  int64_t &next = (*starts)[bucket];
  int64_t dstGlobalIdx = next;
  // store 'elt' into 'dstGlobalIdx'
  auto dst = B.globalIdxToLocalIdx(dstGlobalIdx);
  shmem_putmem(GB + dst.locIdx, &elt, sizeof(SortElement), dst.rank);
  next += 1;
}
```

Adding Conveyors involves control flow adjustment

```
convey_begin(request, sizeof(IdxSortElement), alignof(IdxSortElement));
int64_t i = 0;
while (convey_advance(request, i == locN)) {
  for (; i < locN; i++) {
    SortElement elt = localPart[i];
    int bucket = getBucket(elt, digit);
    int64_t &next = (*starts)[bucket];
    int64_t dstGlobalIdx = next;
    // store 'elt' into 'dstGlobalIdx'
    auto dst = B.globalIdxToLocalIdx(dstGlobalIdx);
    IdxSortElement payload = { .locIdx = dst.locIdx, .value = elt };
    if (! convey_push(request, &payload, dst.rank)) break;
    next += 1;
  }
  IdxSortElement* local;
  while((local = (IdxSortElement*)convey_apull(request, NULL)) != NULL) {
    GB[local->locIdx] = local->value;
  }
}
convey_reset(request);
```

Weak-Scaling Results Across Frameworks



- Problem Sizes:
  - EX: 2^{30} elements per node
  - IB: 2^{29} elements per node
- All the LSD sort implementations showed good weak scaling
- The aggregated Chapel implementation achieved the **highest performance** on both systems

How Do Other Distributed Sort Implementations Compare?

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	(AlltToAll)	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

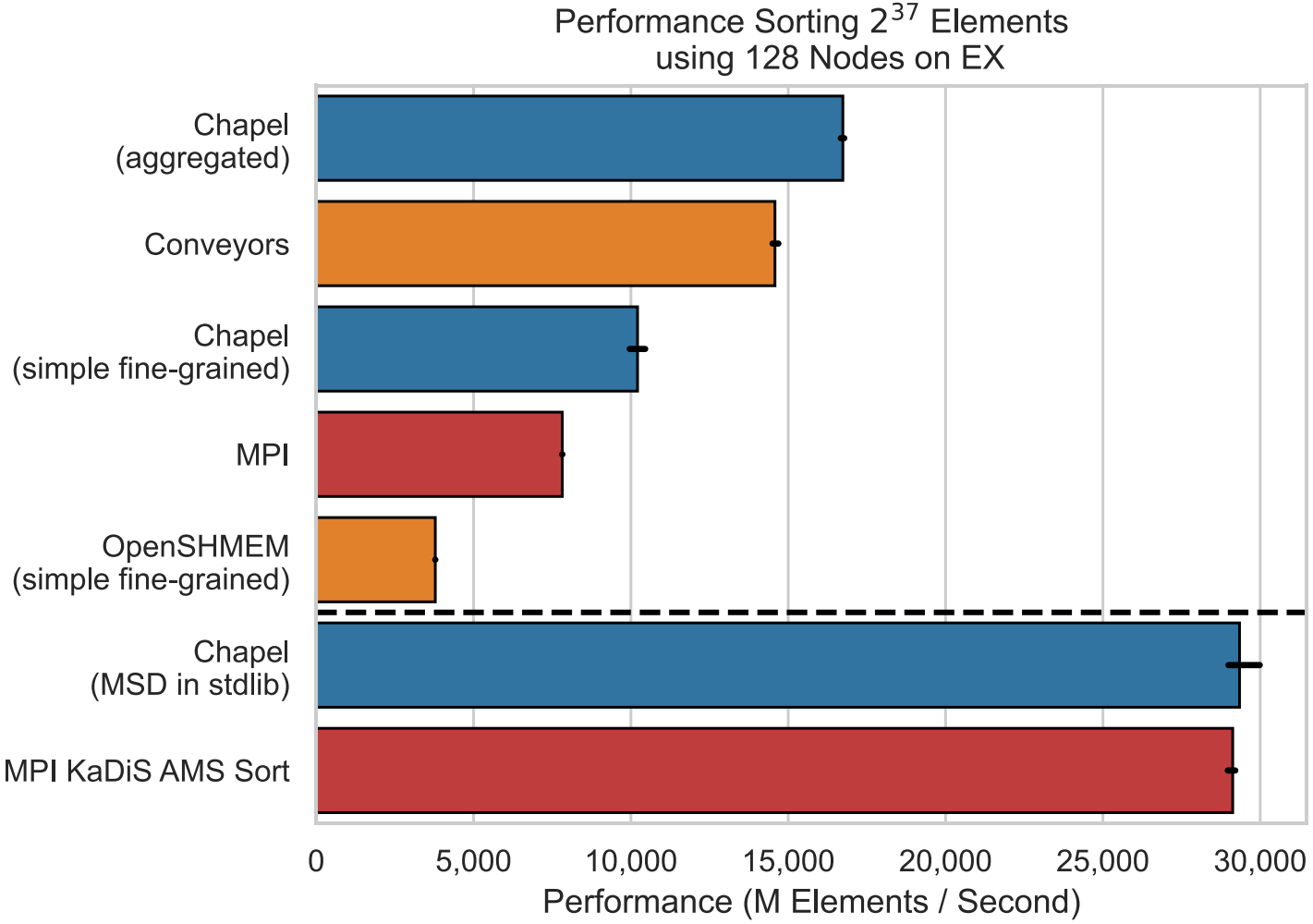
1. Sorting 2^{37} elements using 128 Nodes (M elements sorted / second)
2. Sorting 2^{34} elements using 32 Nodes (M elements sorted / second)

How Do Other Distributed Sort Implementations Compare?

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	(AlltToAll)	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 Libary)	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)

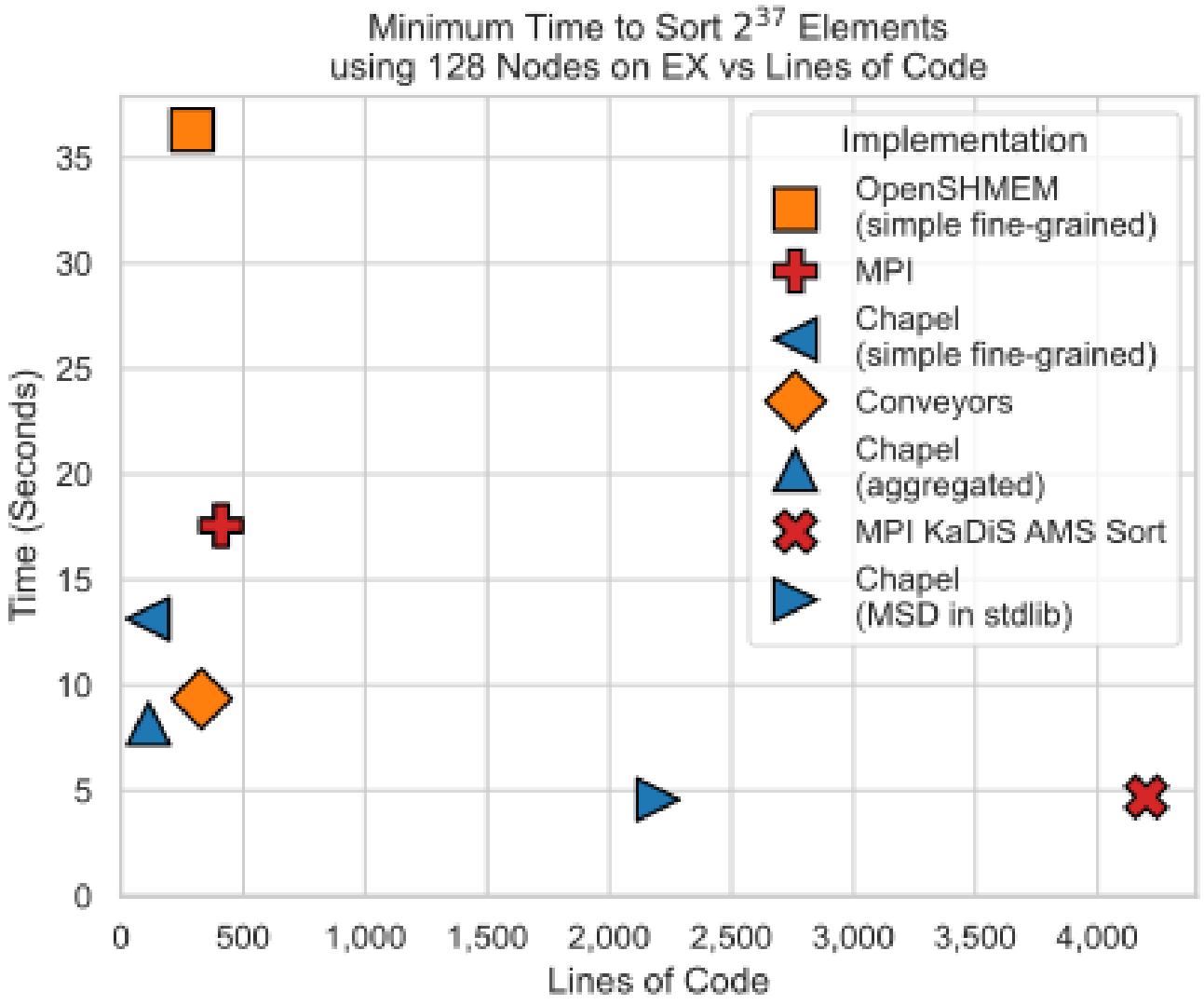
Comparing Performance on 128 HPE Cray EX Nodes



- Bars shown above the dashed line are comparable implementations of LSD sort
  - Chapel using aggregation:
    - ~**4x** faster than OpenSHMEM
    - **2x** faster than MPI
  - Chapel & Conveyors implementations benefit from:
    - message aggregation
    - one-sided communication
    - communication overlap

- Bars shown below the dashed line are other, more complex, distributed sorting algorithms for comparison

Comparing Performance and Productivity



- This scatterplot compares
  - time to sort 2^{37} elements using 128 nodes on EX
  - the number of source lines of code as counted by `cloc`
- Ideal: near origin in the lower left
- 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
- What made some implementations shorter?
  - native support for distributed arrays
  - built-in parallel prefix sum (aka scan)

A Qualitative Comparison of Examined Frameworks

MPI

- **Pro:** Industry standard, large developer community
- **Con:** Missing distributed arrays, automatic aggregation
- **Con:** Often requires additional parallel strategies (OpenMP, CUDA, etc); we observed best performance with fewer ranks than cores

Chapel

- **Pro:** Concisely create parallel tasks across cores and nodes
- **Pro:** Aggregation and sorting in the standard library help productivity in distributed environments
- **Con:** Long compile times

OpenSHMEM

- **Pro:** Very easy to install
- **Pro:** Fast compile times
- **Con:** Uses a custom *collective* allocator, limiting compatibility with C++ data types (e.g. std::vector)
- **Con:** Requires the use of other libraries (Conveyors) to provide aggregation with significant code changes

Lamellar

- **Pro:** Very easy to install as a Rust crate
- **Pro:** Emphasis on memory safety
- **Pro:** Descriptive error messages help to catch bugs at compile time
- **Con:** Currently only supports InfiniBand systems
- **Con:** Long compile times (typical of Rust projects)
- **Con:** Not as mature

Contribute to this Study

- Repository can be found at: https://github.com/hpc-ai-adv-dev/distributed-lsb/
- Contributions welcome and appreciated!
- We plan to make this a *living study repository* and update the performance results when new or improved LSD sort implementations are added

Thank You – Happy Sorting

Michael Ferguson – michael.ferguson@hpe.com
Ryan Friese – ryan.friese@pnnl.com
Shreyas Khandekar – shreyas.khandekar@hpe.com
Matt Drozt – drozt@hpe.com