Distributed-Memory Sorting in the Chapel Standard Library

Michael Ferguson, HPE Labs

2025.10.10

Overview

Chapel's standard library 'sort' can now sort a Block-distributed array

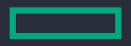
- since Chapel 2.5

This standard distributed sort works with all supported comparators

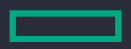
The core implementation is an MSD radix sort

It uses a new method of tracking subproblems that is efficient in distributed memory

It's fast, too!



Background: Radix Sort



An Interface for Radix Sort

Chapel's standard library sort allows flexible specification of how to order elements:

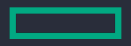
- Provide a '.key' method to sort with a known type (e.g., sort a tuple of integers by element 2)
- Provide a '.compare' method if you only know how to compare elements
- Provide a '.keyPart' method provide part of a key to sort by (e.g., the i^{th} letter in a string)

Standard library sorts in many languages provide something akin to '.key' and '.compare'

- Typically, using `.key` that returns an integer will result in a radix sort
- Other patterns use comparison sort

Chapel is unique in providing a '.keyPart' method which provides a generalized interface for radix sorting!

Terminology: I will use the term key to mean what we are sorting by



What is Radix Sorting?

Sorting is often described as having complexity $\Omega(n \log n)$

- That is, it can't be done in a way that scales better, aside from constant factors

That complexity bound only applies to comparison sort!

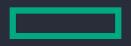
Radix sorting is an $O(n)$ sorting method

- More specifically, for a key length k, it is $O(k\ n)$

Radix sorting is a natural algorithm when the number of possible keys is small

For example, to sort a deck of cards by suit (♤, ♡, ♢, ♧):

1. Get a bucket for each suit (4 buckets for the 4 possible keys)
2. Go through the deck once, putting each card into the bucket according to its suit
3. Combine the cards from the buckets in your preferred suit order



A Building Block: Counting Sort

Counting Sort is a strategy to bring that buckets-and-cards algorithm to arrays

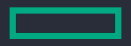
In simplest form, this algorithm uses 3 arrays:

A: Input Array (*n* elements)
B: Output Array (*n* elements)
C: Counts Array (1 count per possible key; e.g., an integer for each of ♤, ♡, ♢, ♧)

The algorithm proceeds like this:

1. For each element in the input array, increment the counts array accordingly: C[A[i]]++
2. Compute the starting indices in the output array B from the counts (a *prefix sum*) & store these in C
3. For each element in the input array, store and increment the count: k=A[i]; B[C[k]] = A[i]; C[k]++

This algorithm can be done in parallel and is stable (preserves order of equal elements)
Terminology: I'll call this algorithm ***partition*** *and the regions it produces* ***buckets***



Partition / Counting Sort Example

Index	Key	Value
0	♤	A
1	♧	B
2	♡	C
3	♡	D
4	♧	E
5	♡	F
6	♧	G
7	♤	H
8	♢	I
9	♡	J

Input

Count

Key	Count
♤	2
♡	4
♢	1
♧	2

Prefix Sum / exclusive scan

Key	Start
♤	0
♡	2
♢	6
♧	7

Output

Index	Key	Value
0	♤	A
1	♤	H
2	♡	C
3	♡	D
4	♡	F
5	♡	J
6	♢	I
7	♧	B
8	♧	E
9	♧	G

Parallel version similar, computes a start offset for each key for each task (see [1])

Simple Least-Significant-Digit-First Radix Sort

Assuming we are working with k-digit numbers, we can sort integers using this recipe:

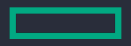
Arrays *A*, *B*, *C* as before (*A* and *B* store *n* elements, *C* stores an integer for each possible digit)

For each digit position *i*, starting from the least significant:

1. Partition from *A* to *B* according to the digits at position *i*
2. Swap *A* and *B*

Observations:

- The partition operation has to be stable (preserve the order of elements with the same digit value)
- LSD sort is parallel if the partition operation is parallel
- LSD sort assumes that the array elements all have the same number of digits
  - This makes it unsuitable for sorting strings of arbitrary length
  - We can imagine padding out strings to match the longest length, but that would make it much less efficient



Least-Significant-Digit-First Radix Sort: Example

Let’s sort 0123 0323 2130 0001 2120 3120 0002 3323 2323 0000

Partition by digit 3 (the least-significant digit):

2130 2120 3120 0000 | 0001 | 0002 | 0123 0323 3323 2323

Partition by digit 2:

0000 0001 0002 | | 2120 3120 0123 0323 3323 2323

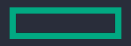
Partition by digit 1:

0000 0001 0002 | 2120 3120 0123 | | 0323 3323 2323

Partition by digit 0:

0000 0001 0002 0123 0323 | | 2120 2323 | 3120 3323

Sorted!



Most-Significant-Digit-First Radix Sort

Arrays *A*, *B*, *C* as before (*A* and *B* store *n* elements, *C* stores an integer for each possible digit)

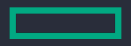
A queue *Q* containing elements like {start: int, end: int, digit: int}

Add {start=0, end=n-1, pos=0} to *Q*

While the queue is not empty:

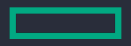
1. w = pop(Q)
2. Partition from A[w.start .. w.end] to B[w.start .. w.end] according to the digit position w.pos
3. For each bucket b that the partition created:
   - If it has no more digits, continue
   - If it has a small number of elements, sort it with a comparison sort
   - Otherwise, push {b.start, b.end, w.pos+1} to Q
4. B[w.start .. w.end] = A[w.start .. w.end]

Note: This slide uses non-optimal data movement between arrays



Most-Significant-Digit-First Radix Sort: Observations

- MSD Radix sort can directly handle keys with varying length
  - Partition can make a region dedicated to elements that have no more digits
  - With '.keyPart', users can specify how to retrieve the i^{th} digit & possibly indicate no more digits
- The number of work items in the queue grows while their size shrinks
  - As a result, a parallel partition helps primarily with the initial partition
- It's difficult to work with a queue like this for a parallel and distributed version



Most-Significant-Digit-First Radix Sort: Example

Let's sort 0123 033 2130 00 221 3120 02 3323 3 0000

When we partition, elements with no more digits go into a first region indicated by [ ]

Partition by digit 0 (the most significant digit):

0123 033 00 02 0000 | | 2130 221 | 3120 3323 3

Partition within each multi-element region by digit 1:

00 0000 | 0123 | 02 | 033 | 2130 221 | [3] | 3120 | 3323

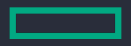
Partition within each multi-element region by digit 2:

[00] | 0000 | 0123 | 02 | 033 | 221 2130 | [3] | 3120 | 3323

Partition within each multi-element region by digit 3:

[00] | 0000 | 0123 | 02 | 033 | [221] | 2130 | [3] | 3120 | 3323

Sorted!



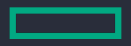
What About Comparison Sorting?

We can create a sample sort based on an MSD Radix Sort

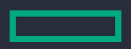
Instead of partitioning a bucket based on a digit position, partition based on splitters

- randomly sample elements in the bucket and sort them and keep every k^{th} one to create splitters
- compute the bucket for each element with binary search on the splitters

With this strategy, we also have a distributed sample sort in the Chapel standard library as of 2.5!



The New Radix Sort



Distributed MSD Radix Sort: The Journey

Previously, we had a good distributed LSD Radix sort implemented in Chapel (Arkouda's sort)

- Problem: LSD sort doesn't support '.keyPart', so Arkouda's LSD sort is not ideal for the standard library

Can we make a good distributed MSD Radix Sort?

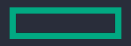
- I attempted this in 2019 (see my CHIUW 2019 talk) but it wasn't scaling as well as Arkouda's sort
- This effort was available as the undocumented 'twoArrayDistributedRadixSort'

I revisited the question in December 2024.

New idea: instead of tracking subproblems with a queue, track bucket boundaries with an additional array

- I came across this strategy in the RadixSA implementation [2]

With this new idea, I was able to make something that performs better than Arkouda's LSD sort at scale!



Distributed MSD Radix Sort: Building Blocks

Use an array *R* of size *n* that stores 1 byte per element

- Indicates if a bucket boundary, and if so, bucket type (sorted? Unsorted? Equal elements?) and in A or B

For a small bucket, we can easily scan to find the end of the bucket, and don't care about digit

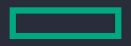
For a large bucket, we have space to store additional info in *R*, including digit position and bucket end

Now all progress towards sorting is recorded in *R*

- R can be distributed just like *A* and *B*
- We can scan forward in *R* to find the start of the next unsorted bucket within a particular region

sortStep is a building block:

- Given a bucket start position, make progress sorting it
- If small, comparison sort; otherwise partition and update the information in *R*



Distributed MSD Radix Sort: Algorithm

Arrays A, B, C as before (A and B store n elements, C stores an integer for each possible digit)

R is an array containing *n* uint(8) elements (storing bucket boundaries & other bucket information)

1. **Initial partition**

   Do a parallel, distributed partition of the data in A, storing into B with 2^{16} buckets.

2. **Each Locale sorts buckets it owns that have elements in multiple Locales**

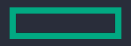
   Further sort any buckets that cross locale boundaries. Each locale is responsible for sorting any buckets that start in the region of the array owned by that locale. Each locale does a parallel sortStep to further sort such buckets, until no such bucket exists.

3. **Each task sorts buckets it owns that have elements in multiple tasks**

   Further sort any buckets that cross task boundaries. We think of each task as responsible for a contiguous block of elements within the elements on its locale. Each task does a serial, local sortStep to further sort buckets that start in the region it is responsible for until no such buckets exist.

4. **Each task sorts all remaining buckets**

   Each task does a serial, local sortStep to partially sort the first unsorted bucket in the region it is responsible for. It repeats this process until there are no unsorted buckets in the region it is responsible for.



Distributed MSD Radix Sort: Tricky Pieces

Partition (and sortStep) need to work at 3 different levels of parallelism:

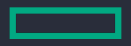
- Initial partition is distributed & parallel on all Locales
- sortStep for buckets spanning Locale boundaries creates many tasks but usually only on 2 locales
- sortStep for buckets spanning task boundaries creates many tasks but all local
- sortStep for buckets within a task's region is serial (already within a parallel loop)

Handling each of these in the same code was a bit awkward

- Let to 2 partition implementations: parallel and serial
- Parallel version needs to handle being distributed or not (depending on provided arrays)
- Lots of 'local' with a condition

I went to some length to avoid operations that unnecessarily do work on each Locale

- I was particularly concerned about this for the part sorting buckets spanning Locale boundaries
- At present, unclear to me how much my efforts here mattered, maybe not so much in practice



Distributed MSD Radix Sort: Example

Weak Scaling on an HPE Cray Supercomputing XC

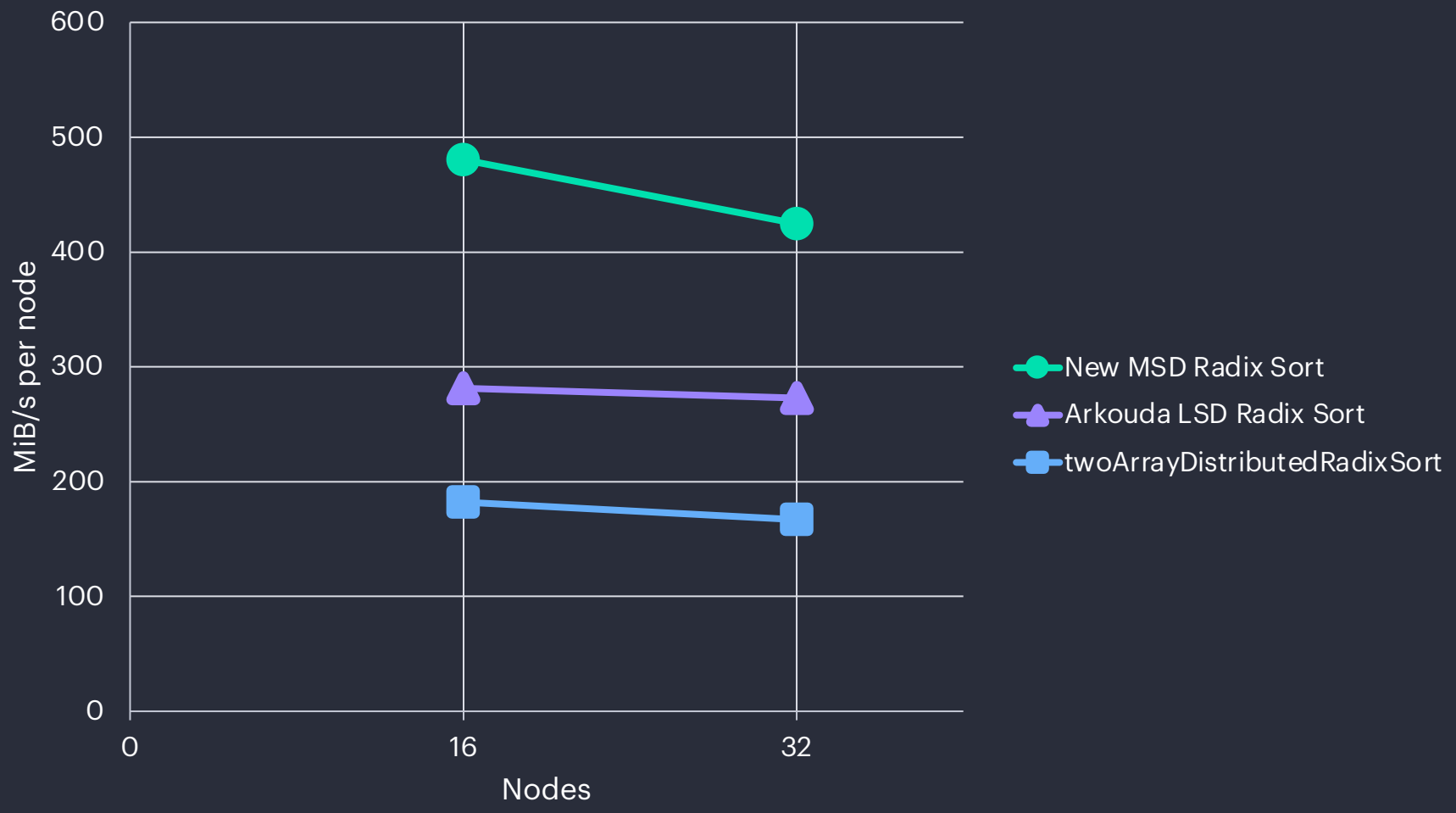
When sorting 2^{31} integers per node (16 GiB per node)

All 3 show pretty good weak scaling on XC

New sort is significantly faster at 32 nodes:

2.5x faster than previous MSD sort

1.6x faster than Arkouda LSD sort



Weak Scaling on HPE Cray Supercomputing EX

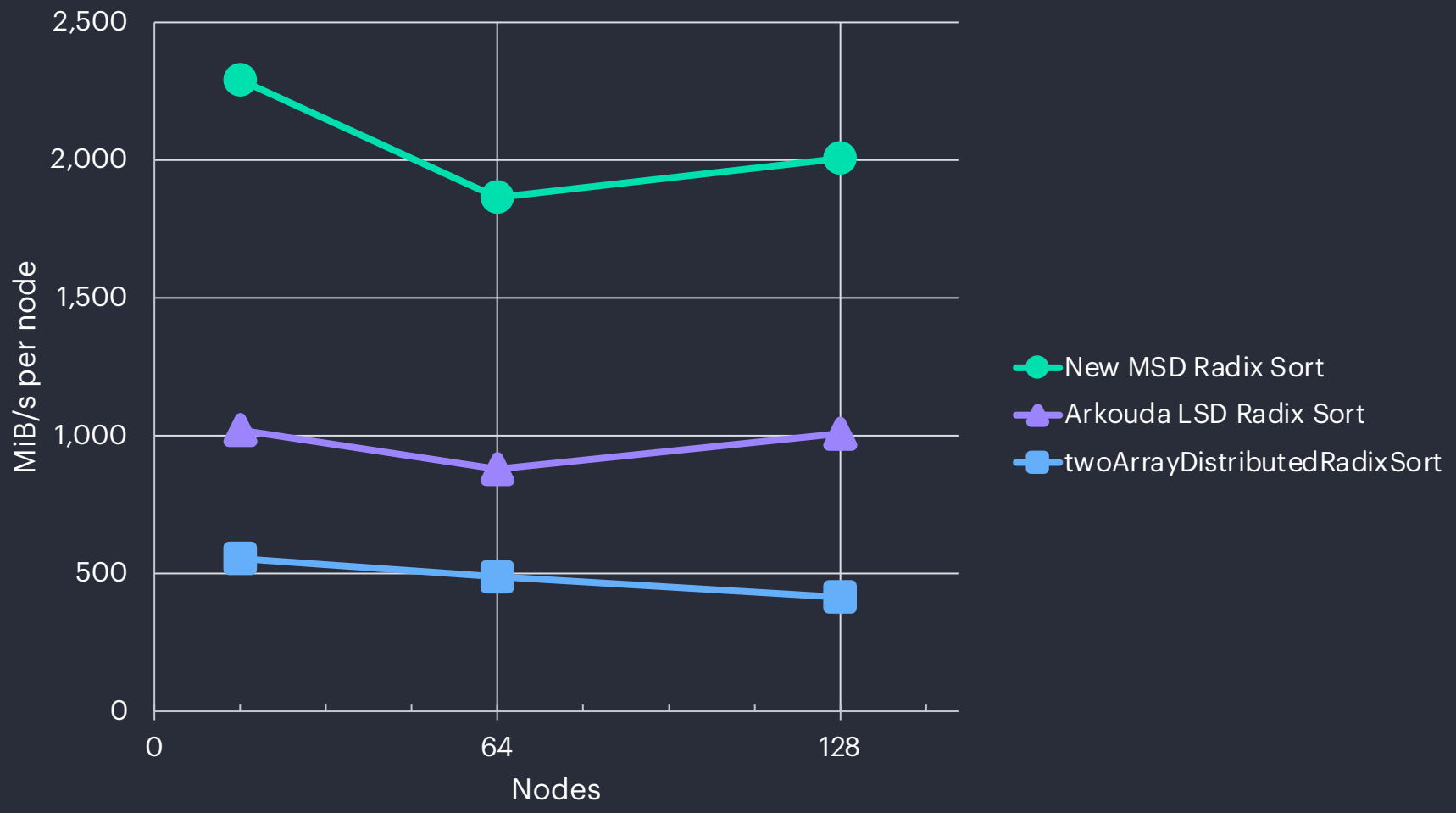
When sorting 2^{31} integers per node (16 GiB per node)

All 3 show pretty good weak scaling on EX

New sort is significantly faster at 128 nodes:

2x faster than Arkouda

5x faster than previous MSD sort



Performance on 128 nodes of HPE Cray Supercomputing EX

Aggregate performance when sorting 2^{37} (int,int) elements total

Here, the new sort is

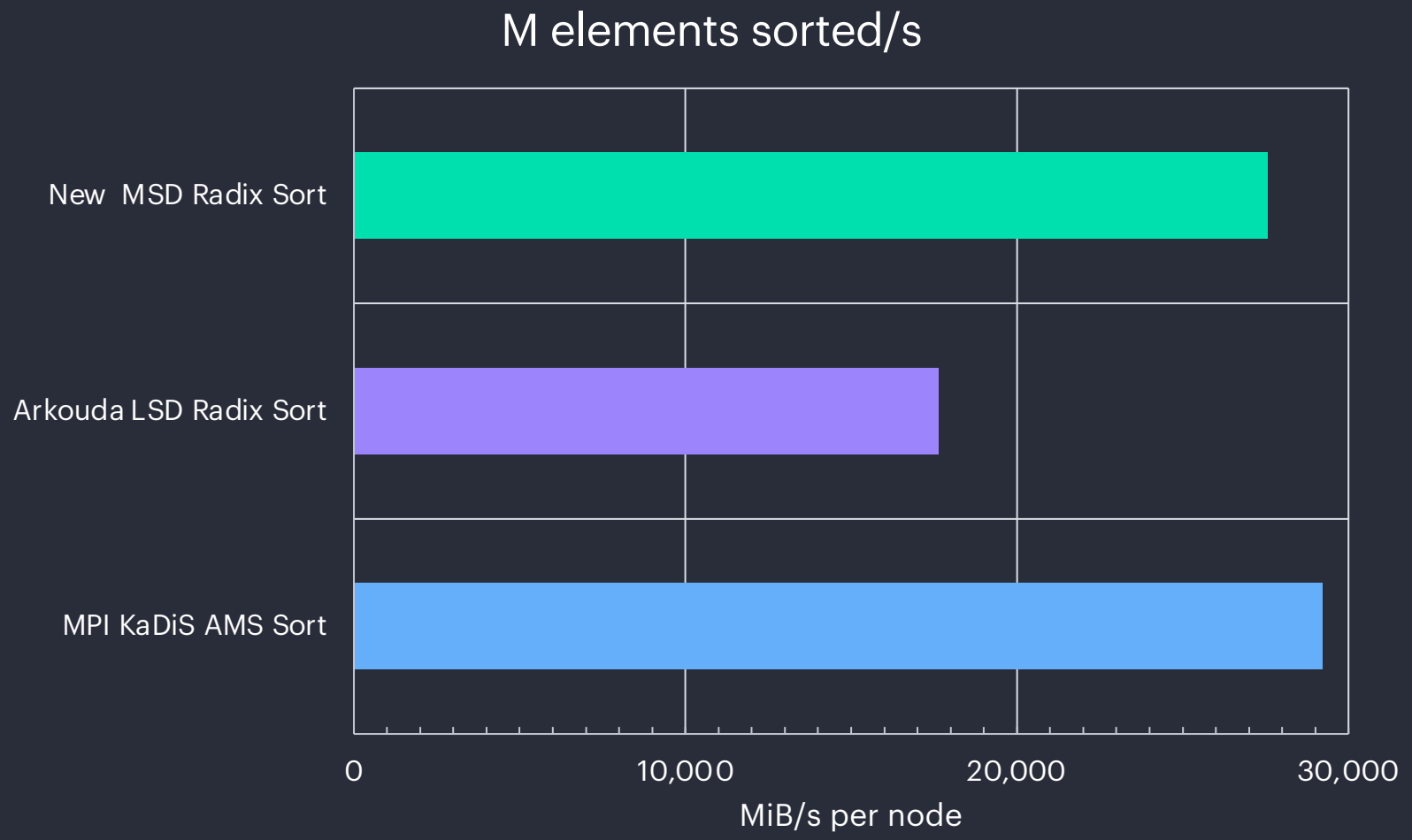
1.5x faster

than the Arkouda LSD Sort

The new sort is also

competitive with an MPI sort

developed as part of a doctoral thesis



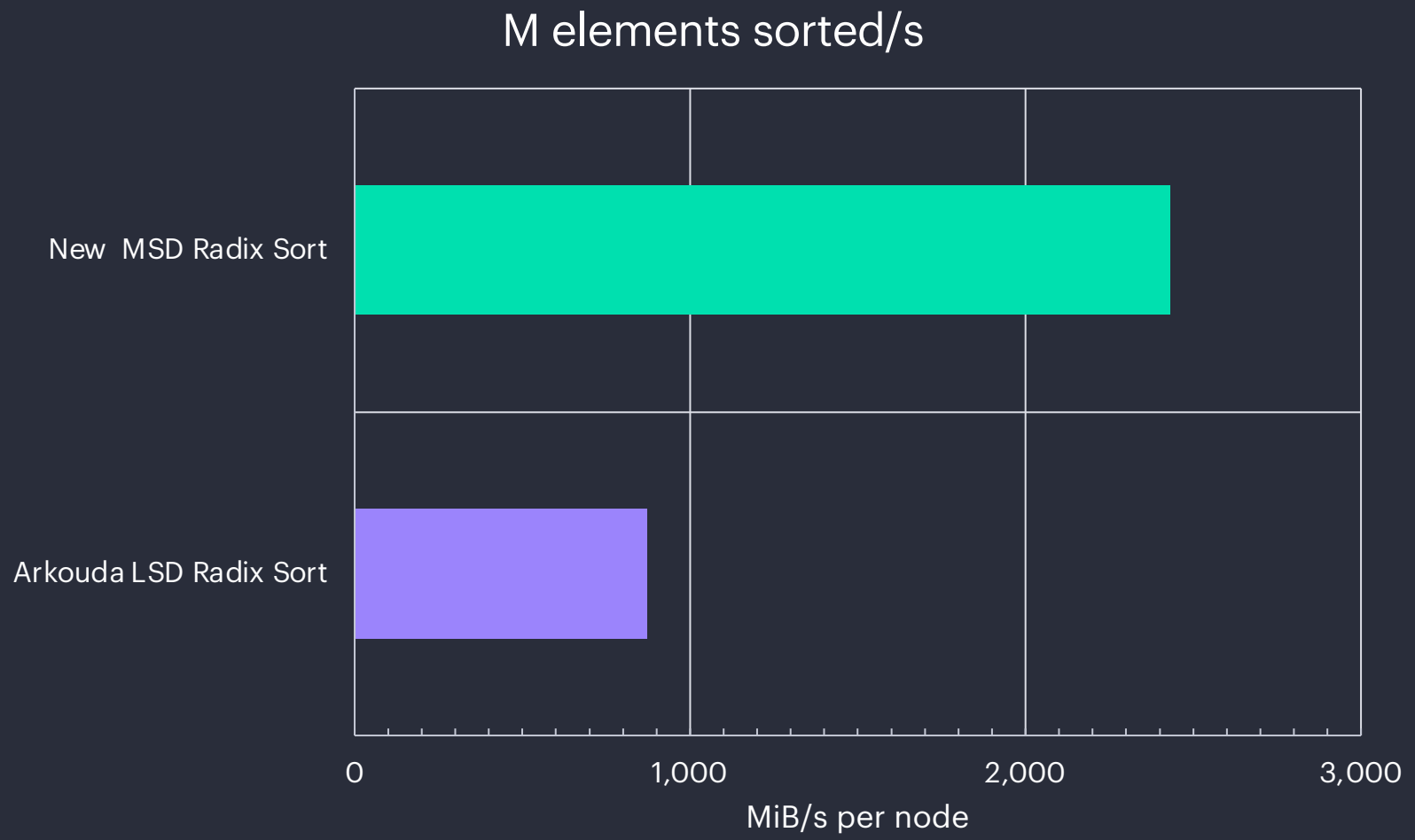
Performance on 32 nodes of an InfiniBand cluster

Aggregate performance when sorting 2^{34} (int,int) elements total

On 32 nodes on an InfiniBand system, the new sort is

2.8x faster

than Arkouda LSD Sort



Conclusion & Next Steps

Since Chapel 2.5, the standard library ‘sort’ has been able to sort a Block-distributed array!

A different way of managing sub-tasks enabled better distributed performance vs previous distributed MSD sorts in Chapel

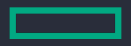
- **5x faster** than previous (undocumented) MSD distributed sort in the standard library

New implementation is flexible and fast

- Supports sorting with ‘.key’, ’.keyPart’, or ‘.compare’
- Competitive with a well-known MPI sort implementation

Next Steps

- Improve Chapel domains & arrays to avoid involving Locales not relevant in a distributed operation
  - E.g. creating a distributed domain, array, or slice that only involves 2 locales -- examples in this gist
- Further optimize the new sort



References

[1] Arkouda's LSD Radix Sort Implementation https://github.com/Bears-R-Us/arkouda/blob/main/src/RadixSortLSD.chpl

[2] S. Rajasekaran and M. Nicolae, "An elegant algorithm for the construction of suffix arrays," Journal of Discrete Algorithms, vol. 27, pp. 21–28, 2014. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1570866714000173