COLUMBIA UNIVERSITY
IN THE CITY OF NEW YORK

A Comprehensive Study of Main-Memory Partitioning and its Application to Large-Scale Comparison- and Radix-Sort

Orestis Polychroniou

Kenneth A. Ross

Usage of partitioning

- Joins
  - **Hash** partition to small (cache-resident) pieces
    - Build & probe (shared-nothing) hash tables **in-cache**
    - Zero cache misses in the final phase
    - Best approach on **single-core** [Manegold et al. VLDB ’00]
    - Best approach on **multi-core** [Kim et al. VLDB ’09]

- Aggregation
  - **Hash** partition to small (cache-resident) pieces
    - **Update** partial aggregates **in-cache**
    - Avoid synchronization between threads [Ye et al., DaMoN ’11, Raman et al. VLDB ’13]
    - Avoid contention of hot aggregates [Cieslewicz et al., SIGMOD ’10]

Usage of partitioning

- Sorting
  - A sub-problem of all other problems …
    - Sort-merge-join
    - Sort-aggregation
    - Compression, …
  - **Radix**-sort
    - Faster than merge-sort [Satish et al. SIGMOD ’10, Wassenberg et al. EuroPar ’11]
  - Hybrid approaches in related work
    - First range partition the data (using MSB radix)
    - Then sort using quick-sort & heap-sort [Albutiu VLDB ’12]
    - Then sort using merge-sort [Balkesen VLDB ’14]

Outline

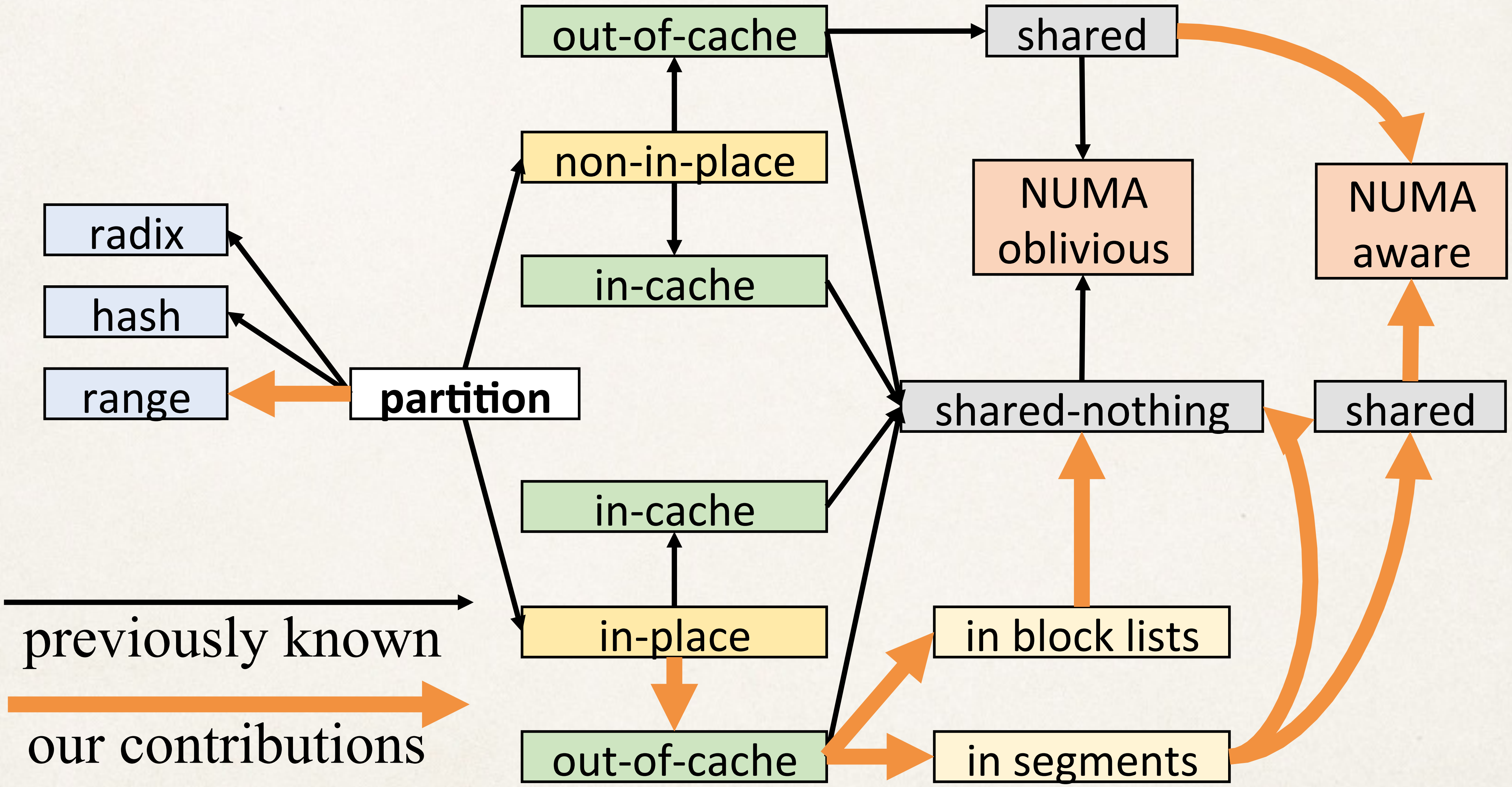
- Discuss partitioning
  - Categorization
  - Shared-nothing partitioning
    - In-cache
    - Out-of-cache
  - Parallel in-place partitioning
  - Range partitioning

- Apply partitioning to sorting
  - Mix **all** partitioning variants to create sorting algorithms with good properties
    - Each with **different** characteristics
    - Minimize NUMA transfers
    - Ensure load-balancing & skew-awareness

Categories of partitioning

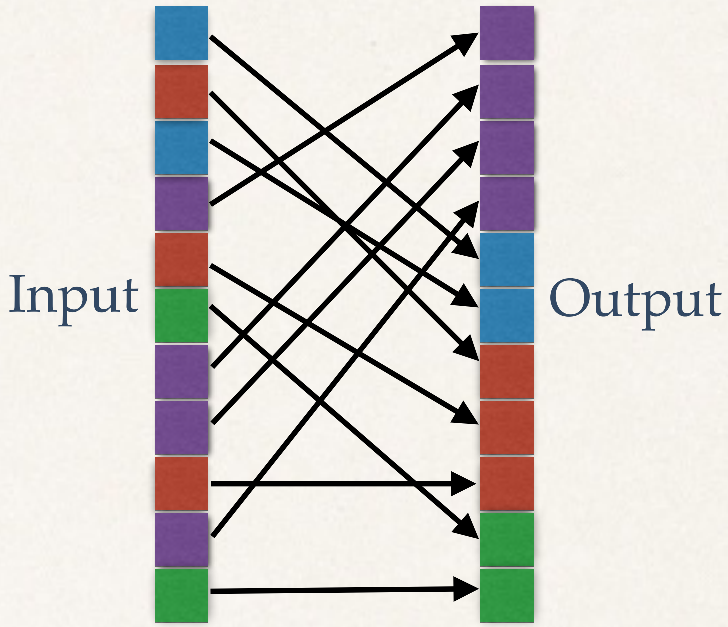
- Types of partitioning
    - Hash / radix / range
- Memory usage
    - Non-in-place / in-place
- Parallelization model
    - Shared / shared-nothing
- Memory hierarchy layer
    - In-cache / out-of-cache / out-of-CPU
- NUMA awareness
    - NUMA aware / NUMA oblivious

Categories of partitioning

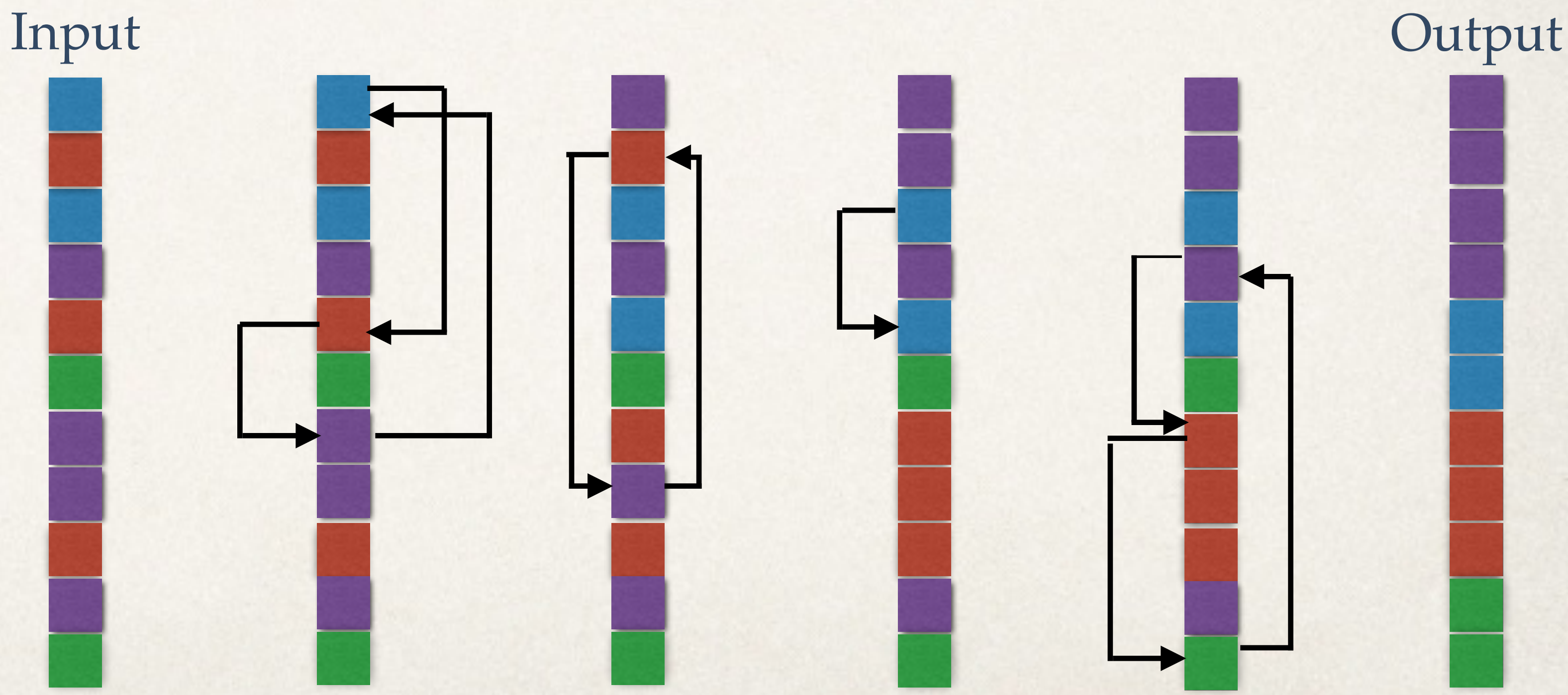


Partition in-cache

- Non-in-place
    - Compute **histogram**
        - Prefix sum to offsets
    - Transfer each tuple once
        - Input to output (separate array)



- In-place
    - Compute histogram
    - Transfer in-place
        - Swap tuples in-place
        - Minimize "swap cycles"

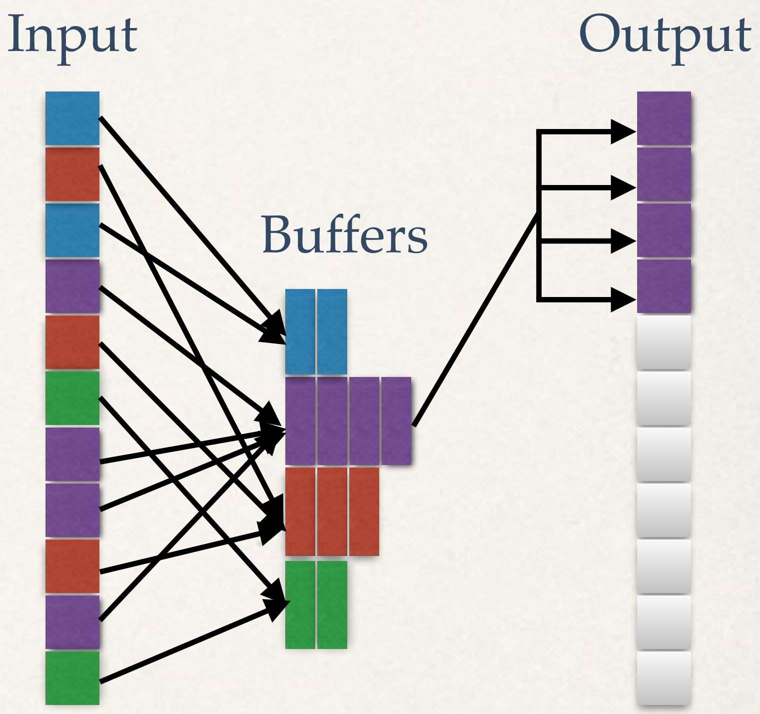


Partition in-cache

- On large working sets (larger than the cache)
  - TLB thrashing [Manegold et al. VLDB ’00]
    - Best case: fanout ~ L1 TLB capacity (64 in Intel CPUs)
    - Otherwise TLB miss for every tuple
  - Cache conflicts [Satish et al. SIGMOD ’10]
    - Worst case: fanout ~ cache set-associativity (8-way in Intel CPUs)
    - Otherwise cache miss for every tuple (on top of TLB miss)
  - Cache pollution [Wassenberg et al. EuroPar ’11]
    - Minimize output caching & write-combining

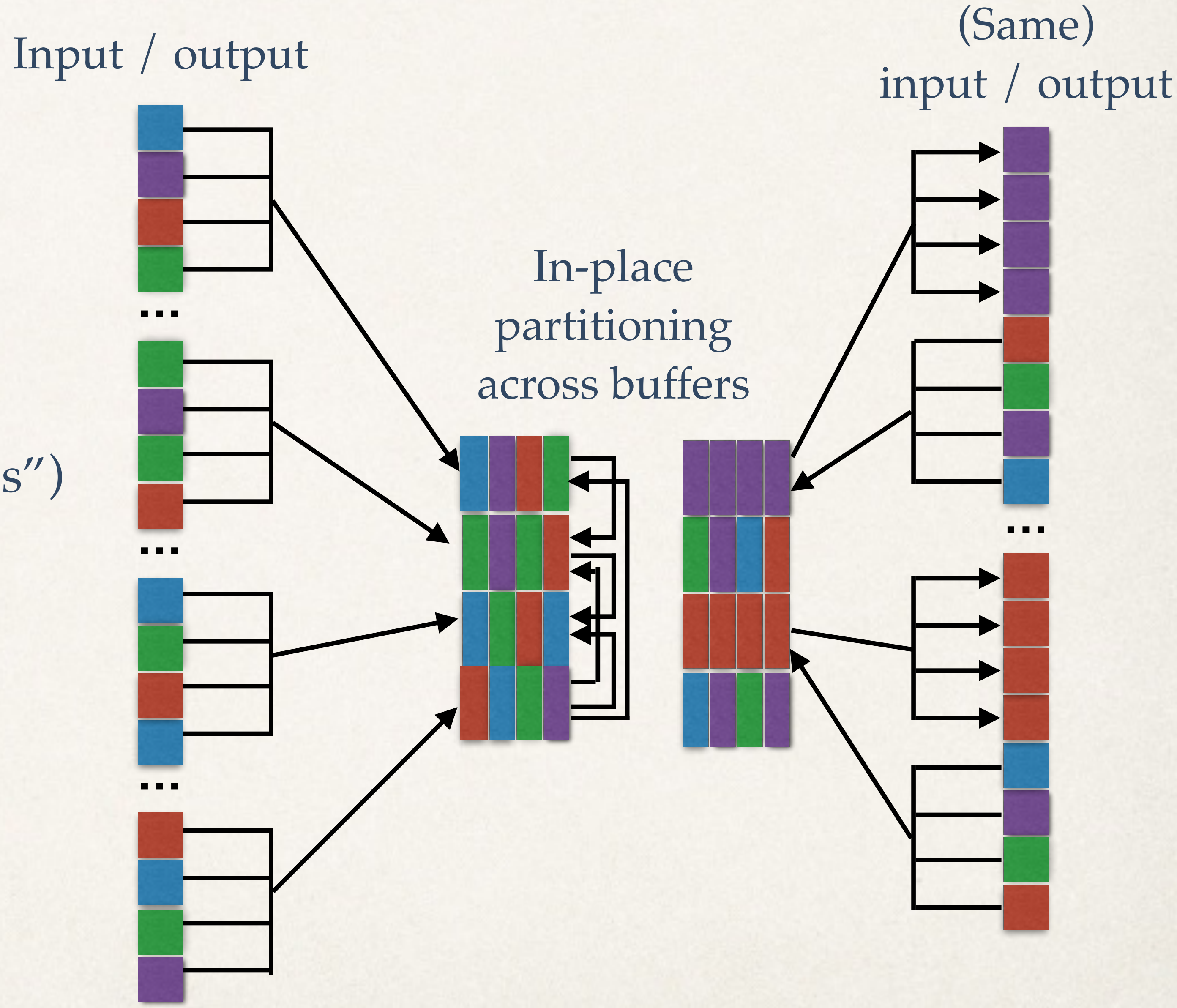
Partition out-of-cache

- Adjust in-cache version
  - **Buffer** each partition in-cache
    - Maintain one buffer per partition
  - TLB thrashing reduced L times
    - Only **1** access **out-of-cache** (TLB miss)
    - For every **L** accesses **in-cache** (TLB hit)
  - Cache conflicts reduced L times
    - Associativity **irrelevant** for buffer accesses
    - Write-combining **bypasses** private caches

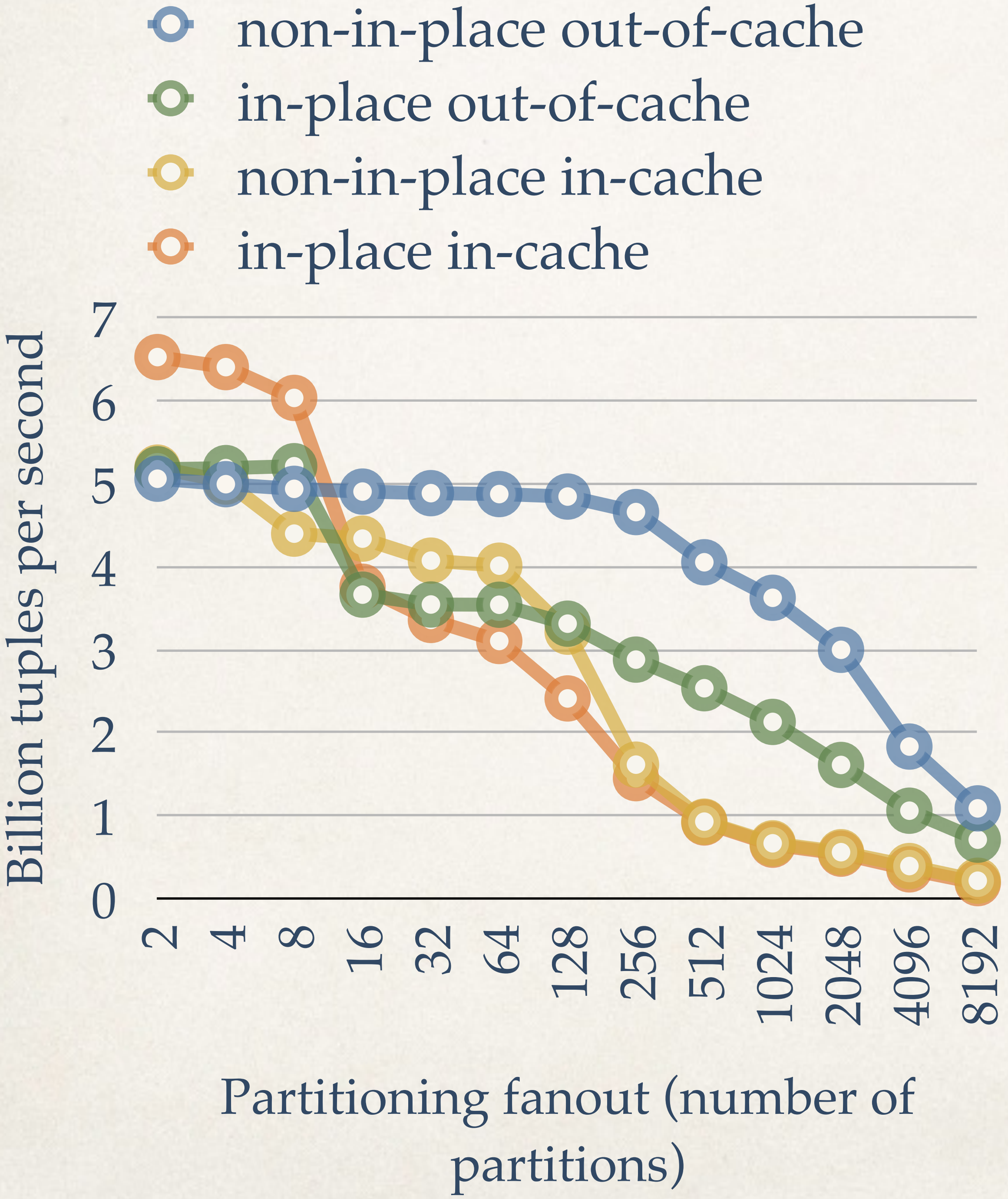


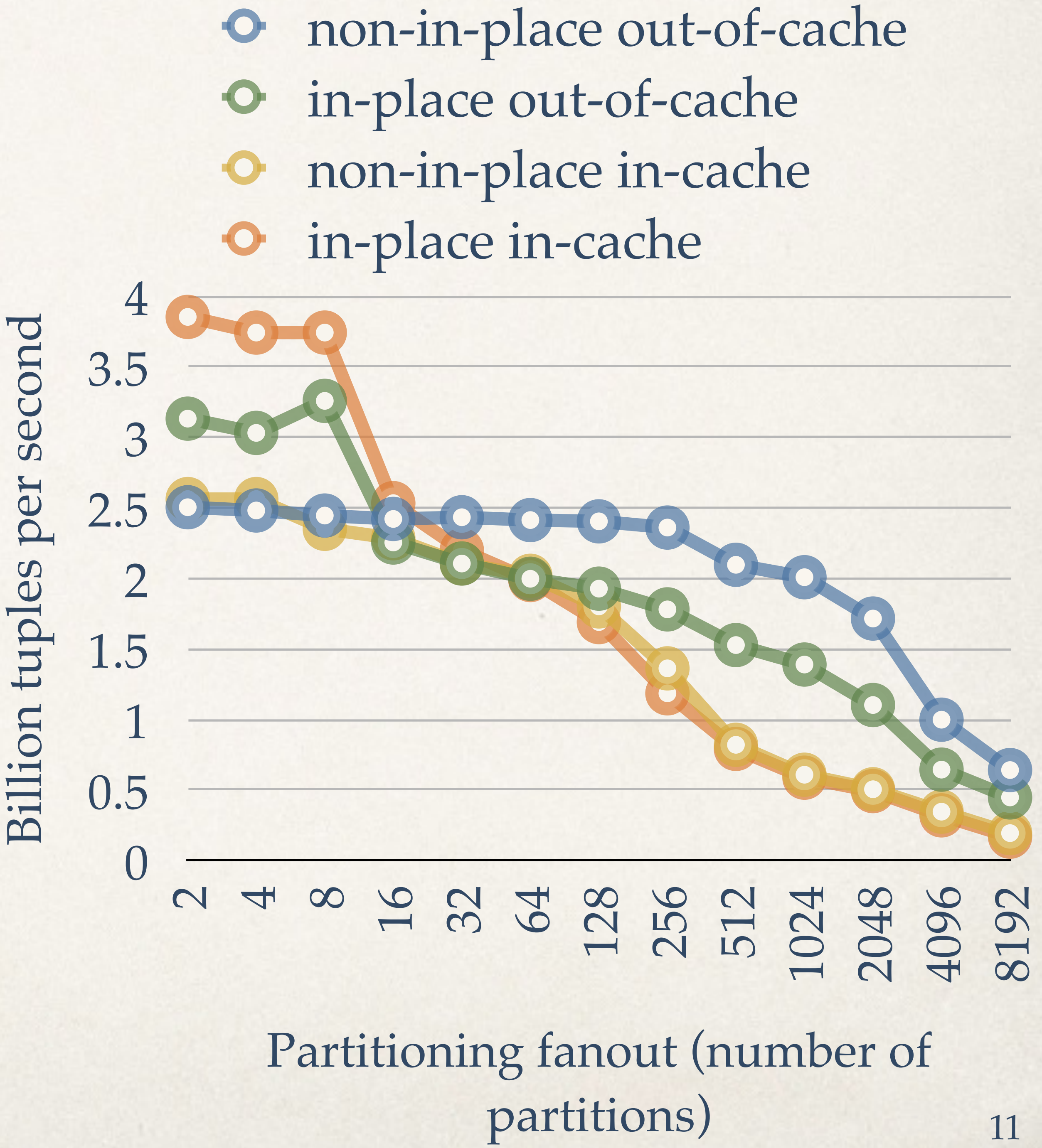
Partition out-of-cache

- Adjust to do in-place
  - Transfer data in **cache lines**
    - **Amortize** out-of-cache accesses
  - "Work" on the cached buffers
    - Similar to in-cache ("swap cycles")
    - Data transferred **across buffers**
  - Recycle buffers when done
    - **Flush** buffer when filled
    - **Refill** buffer with next data



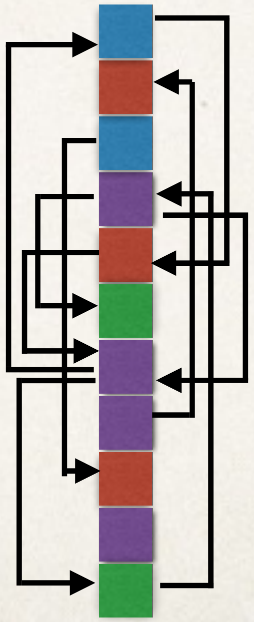
Shared-nothing partitioning


32-bit key & 32-bit payload


64-bit key & 64-bit payload

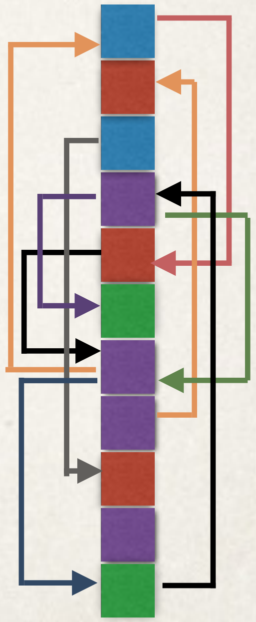
Parallel in-place partitioning

- Partitioning job shared across threads
  - Non-in-place ? Easy.
    - "Interleave" histograms using prefix-sum
    - Common approach for LSB radix-sort
    - Coarse grain granularity synchronization (barriers)
  - In-place ? Hard !
    - As before "swap" items in-place
    - Ensure "safe" swapping (with atomics)
    - **Fine** grain granularity synchronization
    - **Impractical** to synchronize for every tuple



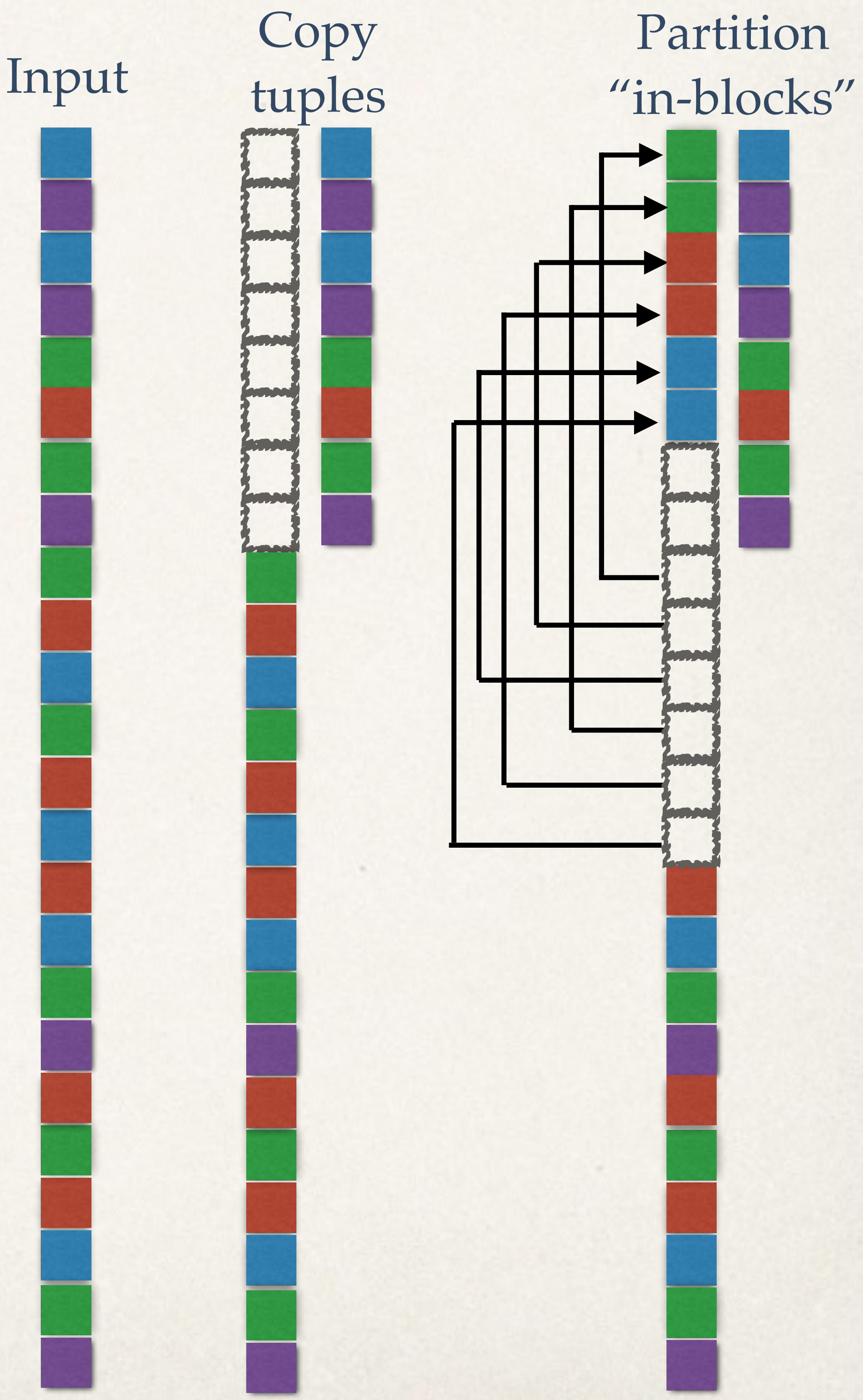
Swaps by 1 thread

Swaps by N threads



Parallel in-place partitioning

- Split in two steps
  - Partition in-place and generate "blocks"
    - Contiguous segments are not the only way
    - A "block" contains tuples from 1 partition only
    - Traverse list-of-blocks: amortized random access
    - Can be done in-place: **re-use input space**
  - Partition blocks in-place
    - "Swap" blocks in-place (not tuples)
    - No buffering needed since blocks are large
    - Synchronization cost **amortized**



Radix / hash / range function

- Radix partitioning
  - Trivial to compute
    - 1 shift & 1 logical-and (or 2 shifts)

```
(key >> shift) & mask
```

- Hash partitioning
  - Using **multiplicative** hashing
    - 1 multiplication & 1 shift
    - Minimum **collisions** are not useful for partitioning

```
(key * factor) >> shift
```

- Range partition function
  - **Binary** search on sorted array of delimiters
    - Very slow compared to the previous even if L1 cache resident
    - Data dependent cache lookups —> L1 **latency** fully exposed

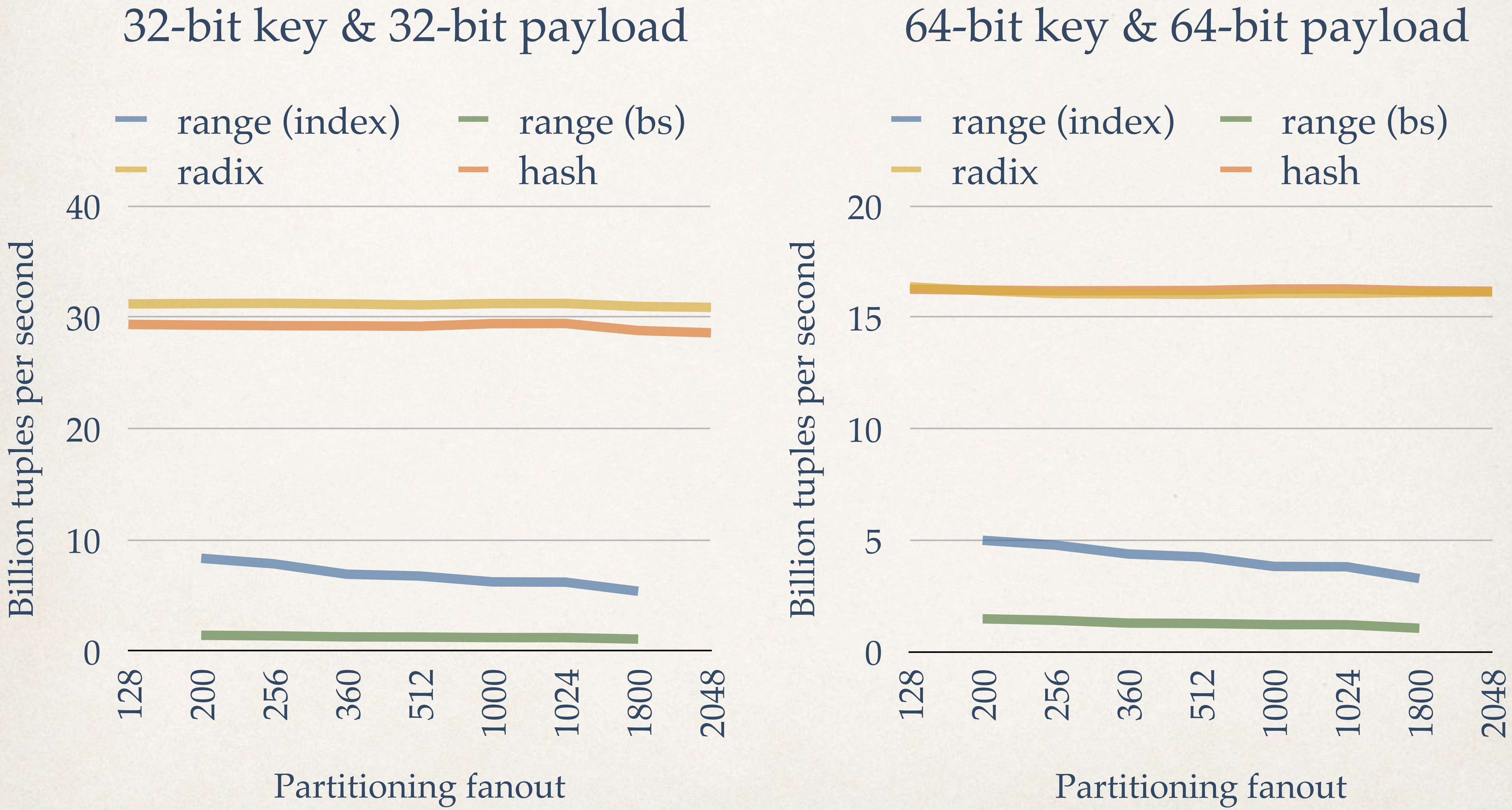
```
lo = 0;
hi = N;
do {
    mid = (lo + hi) >> 1;
    if (key > delim[mid])
         lo = mid + 1;
    else
         hi = mid;
} while (lo < hi);
```

Range partitioning function

- Compute using cache-resident SIMD range tree **index**
  - Index design
    - Store only keys = range splitters
    - Store no pointers
  - Use SIMD to do comparisons
    - On root: “Vertical” SIMD search (see paper)
    - On nodes: “Horizontal” SIMD search: k **SIMD comparisons** to find which path to follow
  - Optimize for range partitioning
    - **Unroll** access to each tree level
    - Use different fanout per tree level

```
dwords_1 = _mm_cmpeq_epi32(x, del_ABCD);
dwords_2 = _mm_cmpeq_epi32(x, del_EFGH);
dwords_3 = _mm_cmpeq_epi32(x, del_IJKL);
dwords_4 = _mm_cmpeq_epi32(x, del_MNOP);
words_1 = _mm_pack_epi32(dwords_1, dwords_2);
words_2 = _mm_pack_epi32(dwords_3, dwords_4);
bytes = _mm_pack_epi16(words_1, words_2);
bits = _mm_movemask_epi8(bytes);
dest = trailing_zero_count(bits);
```

Histogram Generation



Sorting

- Applying partitioning to sorting
  - Sorting is ubiquitous in OLAP
    - Sub-problem of joins
    - Sub-problem of aggregations
  - NUMA-aware setup
    - Array equally split in N parts, one per NUMA region
  - Sorting algorithms
    - Stable LSB radix-sort
    - In-place MSB radix-sort
    - Comparison-sort

(Our) LSB Radix-sort

- Stable algorithm
  - Parallel LSB-radix & range partition
    - Shared across threads of **same CPU** (NUMA region) only
    - Sample and use C range partitions for C NUMA regions (C CPUs)
  - Shuffle data across NUMA regions using C range partitions
    - The C range partitions used with the MSB radix bits
  - Parallel radix partition <u>iteratively</u>
    - Shared across threads of **same CPU** only
    - Skip single key range partitions
    - Always saturate partitioning fanout to minimize passes

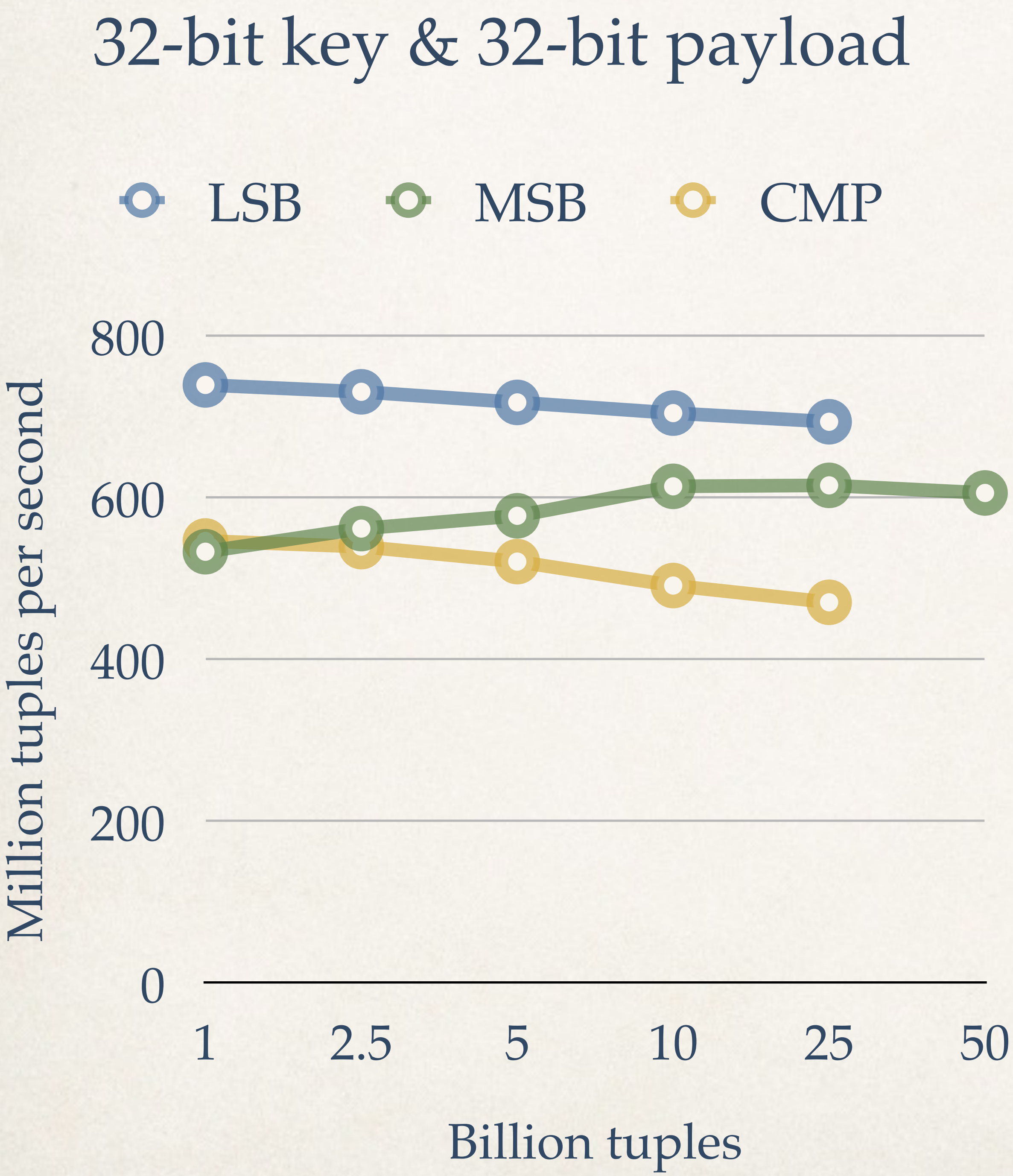
(Our) MSB Radix-sort

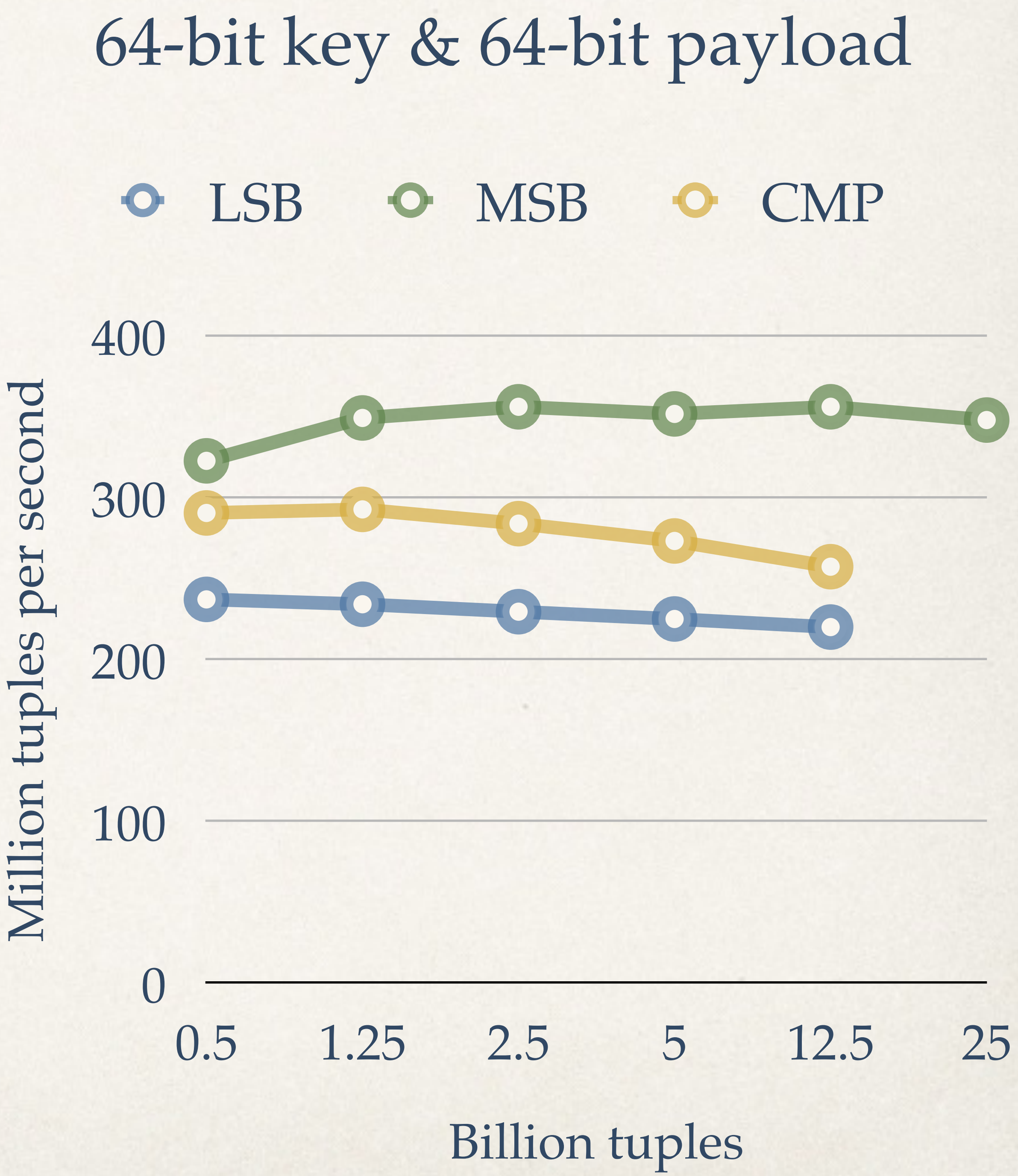
- In-place algorithm
  - Parallel in-place range partition to split across T threads
    - Sample T range delimiters and create T delimiters using MSB radix
    - Range partition locally using 2T delimiters **in-blocks**
  - Shuffle range (& radix) partitioned blocks across NUMA
    - Move **blocks** (not tuples) to amortize synchronization cost
  - In-place radix partition <u>recursively</u> per thread
    - Starting with **out-of-cache** until parts can fit in the cache
    - Switch to **in-cache** and use wider fanout to create very small parts
    - Switch to **insert-sort** for very small parts of items (if radix bits not covered yet)

(Our) Comparison-sort

- Algorithm (non-stable, non-in-place)
  - Parallel range partition & shuffle across NUMA regions
    - Shared across threads of **same CPU** (NUMA region) only
  - Range partition iteratively per thread
    - Dynamically share partitions across threads of same CPU
    - Sample range delimiters (load balancing)
    - Skip single key range partitions (skew efficiency)
  - When in-cache, switch to SIMD comb-sort
    - SIMD comb-sort [Inoue et.al. PACT '07] > SIMD bitonic sort [Chhugani et.al. VLDB '08]
    - On W-wide SIMD: $(n/W) \log n < (n/W) \log(n/W) + n \log W < (n/W) \log 2n$

Sorting Results





Comparison of Sorting Algorithms

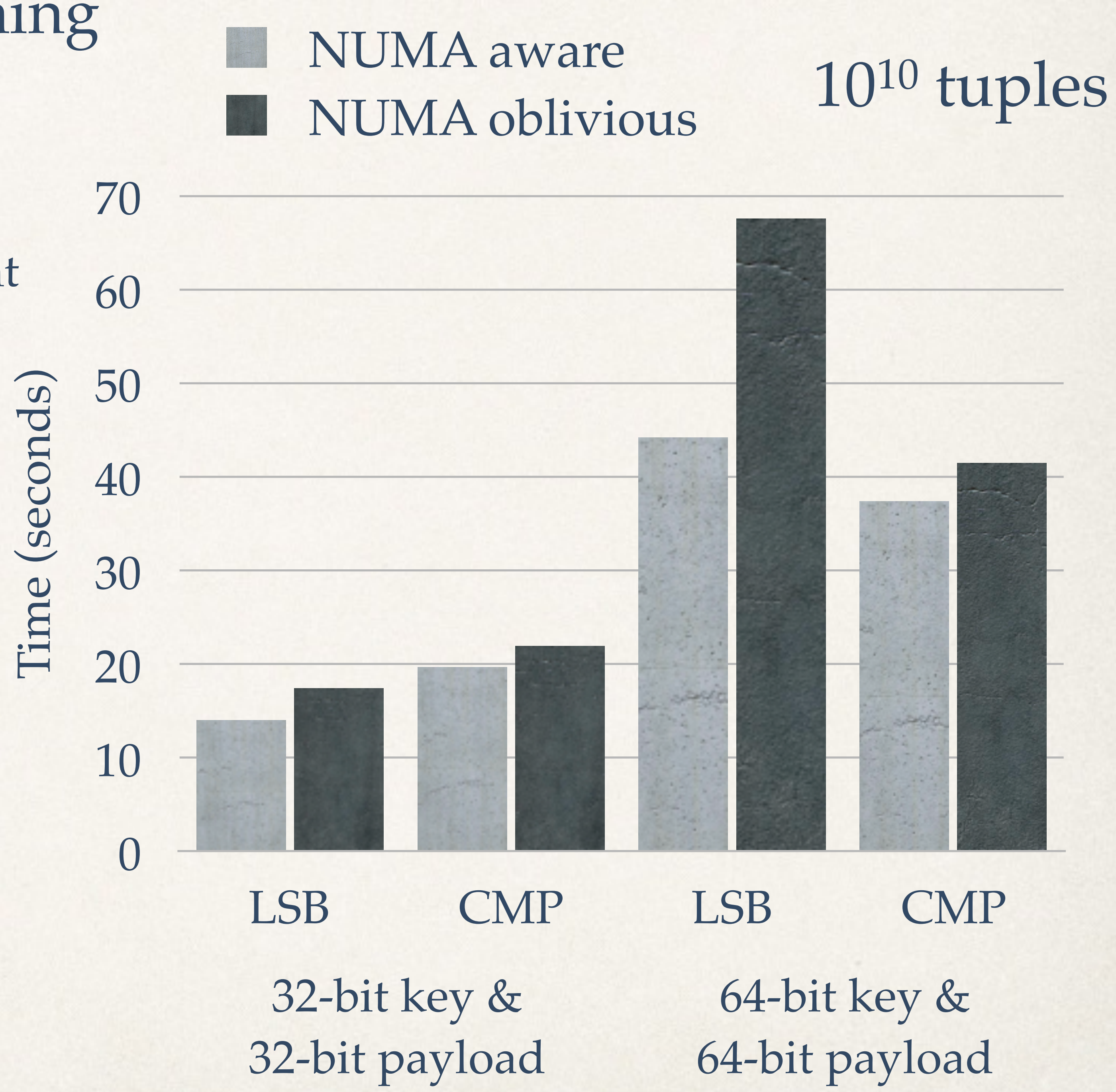
- Our sorting algorithms
  - Stable LSB radix-sort
    - Best for small key domains (LSB)
    - Immune to skew
  - In-place MSB radix-sort
    - Best for large key domains (MSB)
    - Doubles maximum array size (in-place)
  - Comparison sort
    - Comparably efficient on all domains
    - Faster under skew

Comparison of Sorting Algorithms

- Related work
  - In-place radix partitioning & intro-sort [Albutiu et al. VLDB '12]
    - Using in-cache variant out-of-cache & scalar intro-sort
  - Radix partitioning & merge-sort [Balkesen et al. VLDB '14]
    - Radix-based approach: ~675 million tuples / second (not a radix-sort)
    - Comparison-based approach: ~350 million tuples / second (we sort ~550 million)
  - Range-partitioning is faster than merging
    - -12.4% for 1 GB versus half (0.5 GB) [Chhugani et al. VLDB '08]
    - -25% for 8 GB versus half (4 GB) [Balkesen et al. VLDB '14]
    - Our comparison sort: -13% for 25 billion tuples (~186 GB) versus 1 billion tuples

NUMA Awareness

- NUMA (out-of-CPU) partitioning
  - Using local RAM is **faster**
    - Avoid random NUMA placement
    - Using **out-of-cache** variants
  - **Minimize** NUMA transfers
    - Shuffle across NUMA **once**
    - Make all other passes **local**
  - NUMA aware > oblivious
    - **1.23X** in 3 passes (32-bit LSB)
    - **1.53X** in 6 passes (64-bit LSB)



10^{10} tuples

Conclusions

- Partitioning variants with **different** properties
    - Non-in-place & in-place
    - In-cache & out-of-cache & across-NUMA
    - Range & radix & hash

- Sorting = Partitioning
    - For radix-sort (known)
    - For comparison-sort (our result)

- Combine partitioning variants: **trade-offs**
    - **In-place** partitioning: space/time tradeoff
    - **Range** partitioning: load balancing & skew efficiency
    - **NUMA** optimality: better scalability & performance

Questions