Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited

Authors: Cargi Balkesen, Gustavo Alonso, Jens Teubner, M. Tamer Ozsu
Presenter: Terryn Brunelle

Background

- Multi-core join algorithms
    - Sort-merge
    - Hash-join
- Want to understand performance of parallel data operators on new hardware
- Sort-merge claimed better, but there are new optimizations for hash-join

Sort vs. Hash

Sort

- Massively Parallel Sort-Merge Join (MPSM)
- SIMD data parallelism

Hash

- Preferable on single core
- Partitioning

Sort-Merge Joins

Sort-Merge Strategies: Run Generation

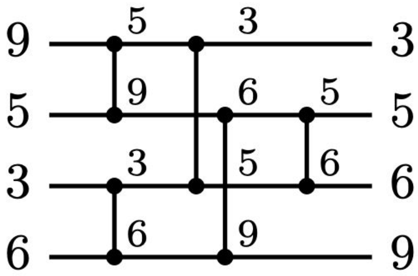


Figure 1: Even-odd network for four inputs.

Sort-Merge Strategies: Merging Sorted Runs

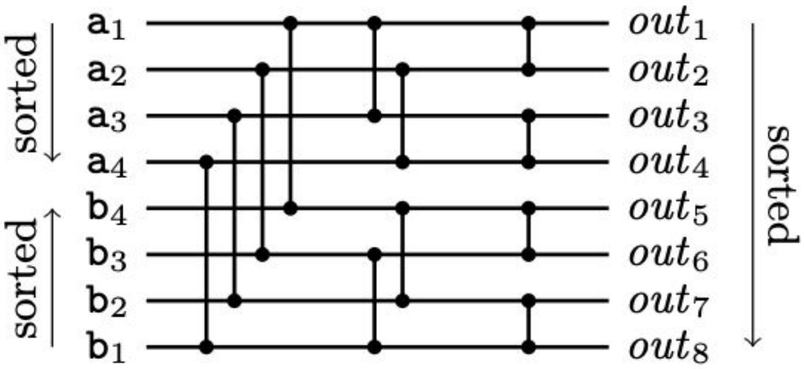
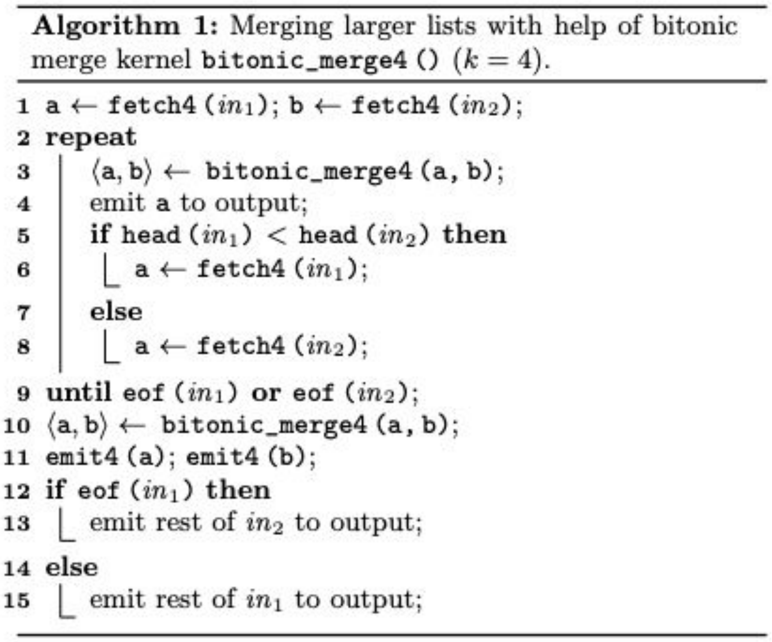


Figure 2: Bitonic merge network.

Merging Larger Lists



Algorithm 1: Merging larger lists with help of bitonic merge kernel `bitonic_merge4()` ($k = 4$).

```
a ← fetch4($in_1$); b ← fetch4($in_2$);
repeat
    ⟨a, b⟩ ← bitonic_merge4(a, b);
    emit a to output;
    if head($in_1$) < head($in_2$) then
        a ← fetch4($in_1$);
    else
        a ← fetch4($in_2$);
until eof($in_1$) or eof($in_2$);
⟨a, b⟩ ← bitonic_merge4(a, b);
emit4(a); emit4(b);
if eof($in_1$) then
    emit rest of $in_2$ to output;
else
    emit rest of $in_1$ to output;
```

Cache-Conscious Sort-Merge

Separate sorting into phases to optimize cache access

1. In-register sorting
2. In-cache sorting
3. Out-of-cache sorting

Out-of-Cache Sorting

- Use two-way merge units connected via FIFO queues
- All queues fit in CPU cache
- Avoids memory bottlenecks even across NUMA boundaries

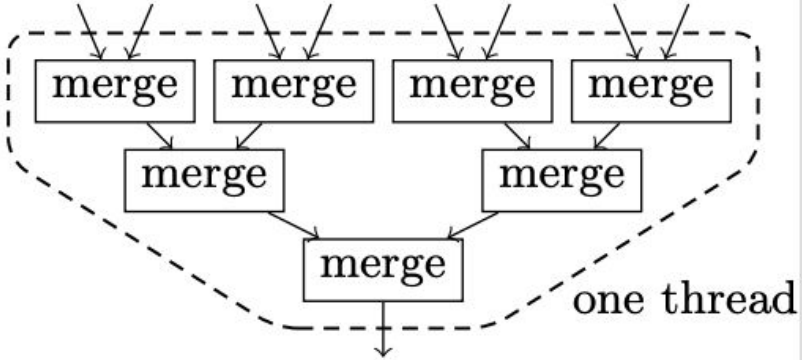
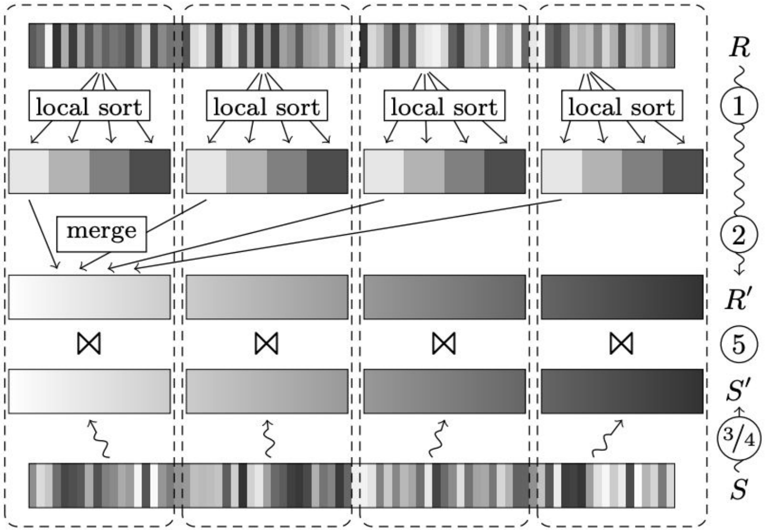


Figure 3: Multi-way merging.

M-Way and M-Pass Sort-Merge Join

1. Threads range-partitions local chunks
2. Multi-way merging to obtain R’ (globally sorted copy of R)
3. Same as 1 but for S
4. Obtain S’ from S in same way as 2
5. Single-pass merge join to find matching pairs

M-Pass: successive two-way bitonic merging in phase 2



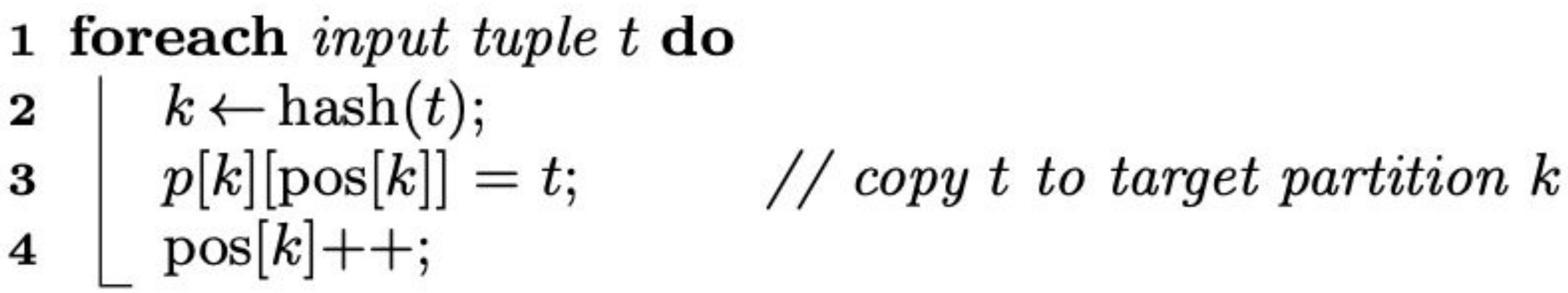
Massively Parallel Sort-Merge Join (MPSM)

1. Globally range-partition R
2. Obtain globally sorted R’
3. Sort S partially without prior partitioning
4. Merge-join run of R with all NUMA-remote runs of S

Good if S is substantially larger than R

Hash-Based Joins

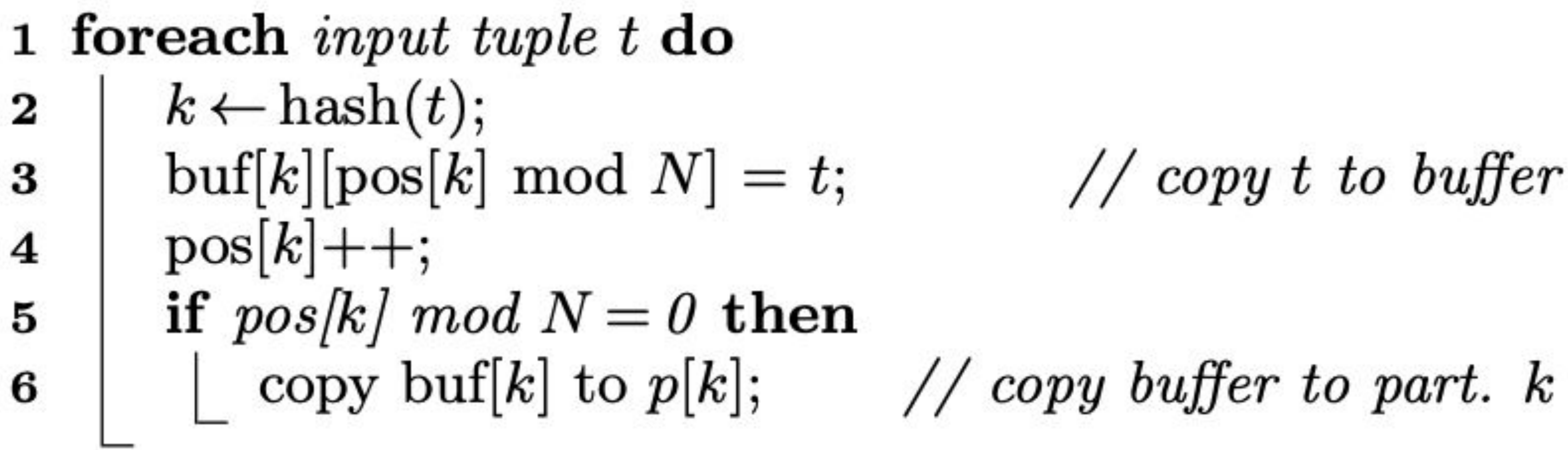
Radix Partitioning



```
foreach input tuple $t$ do
    $k \leftarrow \mathrm{hash}(t)$;
    $p[k][\mathrm{pos}[k]] = t$;          // copy t to target partition k
    $\mathrm{pos}[k]$++;
```

Reduce cache misses and TLB miss effects

Software-Managed Buffers



```
foreach input tuple $t$ do
    $k \leftarrow \mathrm{hash}(t)$;
    $\mathrm{buf}[k][\mathrm{pos}[k] \bmod N] = t$;            // copy $t$ to buffer
    $\mathrm{pos}[k]{+}{+}$;
    if $pos[k]\ mod\ N = 0$ then
        copy $\mathrm{buf}[k]$ to $p[k]$;        // copy buffer to part. $k$
```

Only need to access TLB once every Nth tuple

Radix Hash Join (radix)

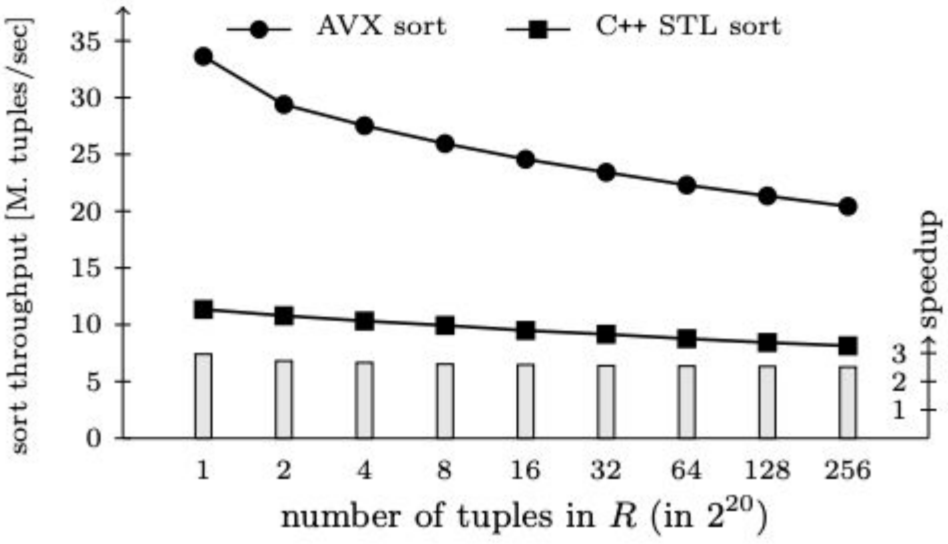
- Apply radix partitioning
- Break the smaller input into pieces that fit into caches
- Run cache-local hash join on individual partition pairs

No-Partitioning Hash Join (n-part)

- Parallel version of hash join
- Divide input relations evenly across worker threads
- Build phase: Workers populate shared hash table with R tuples
- Probe phase: Workers find matching join partners for S portions using hash table

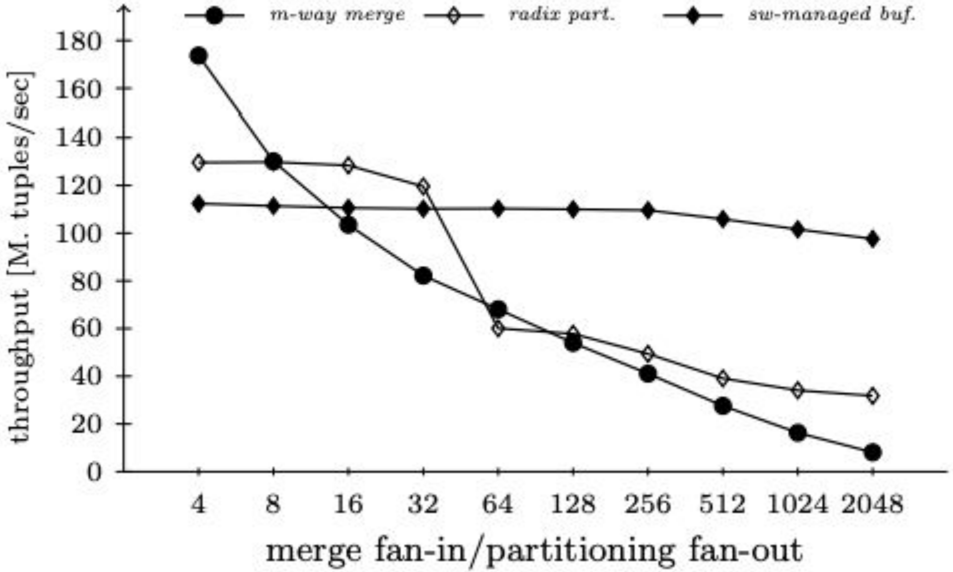
Experimental Results

Sort Phase



AVX sort is 2.5 to 3x faster than C++ STL sort

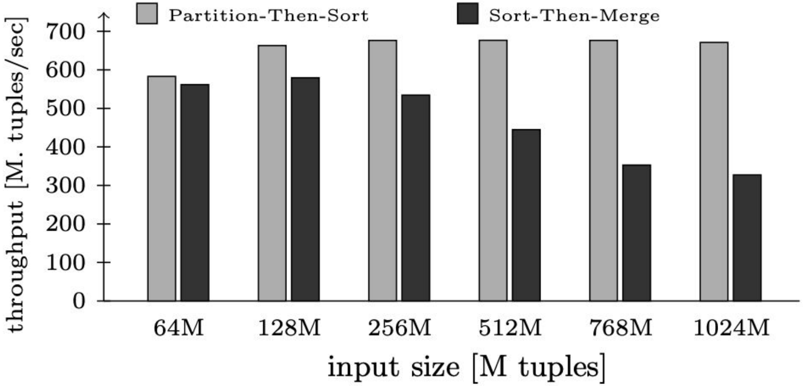
Merge/Partition Phase

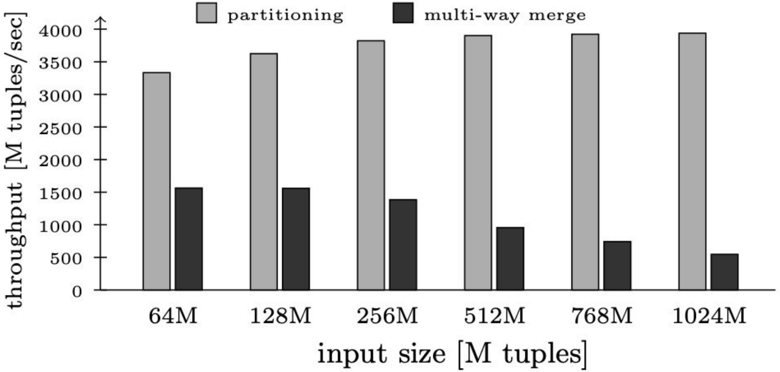


Using Partition with Sort

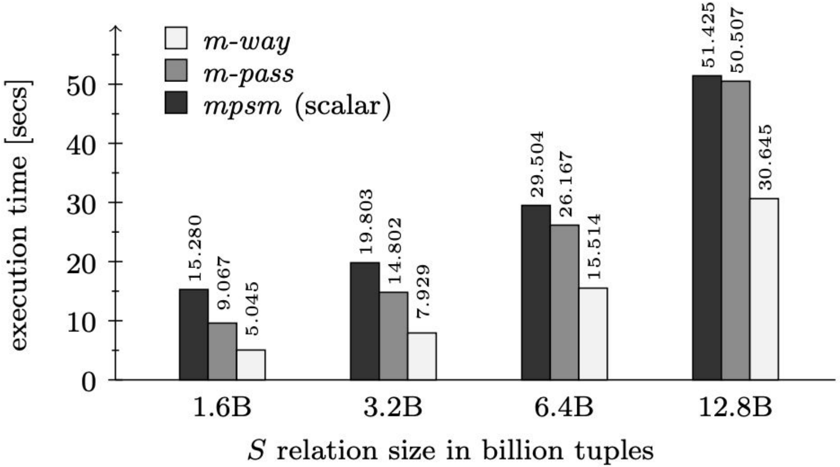
- **Partition-then-Sort** range partitions the input
    - Each partition is individually sorted using AVX sort
- **Sort-then-Merge** creates cache-sized sorted runs
    - Merge sorted runs via multi-way merge

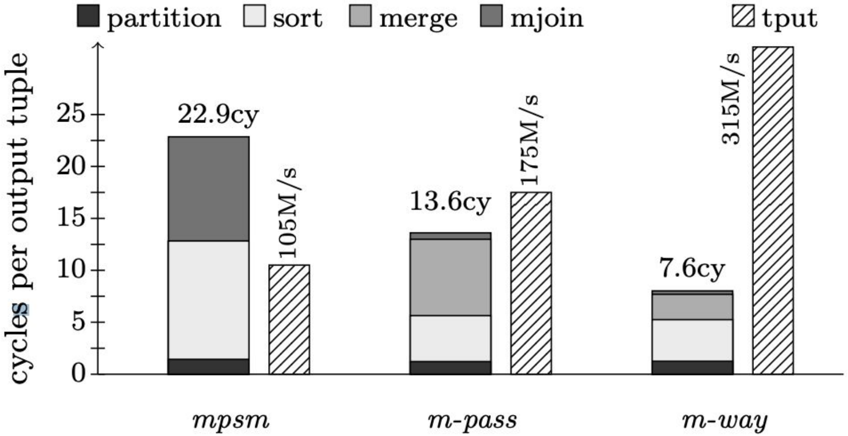
Using Partition with Sort



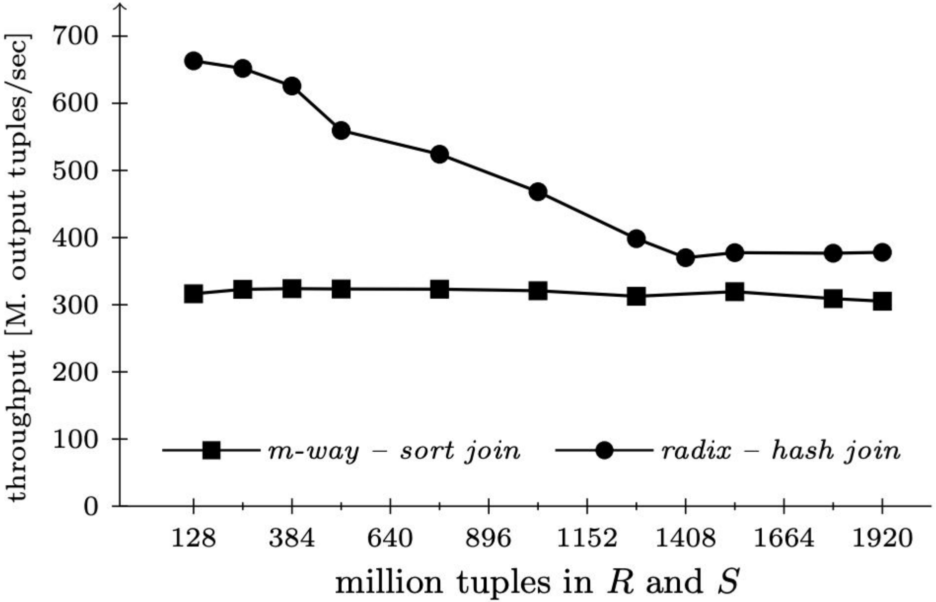


Sort-Merge Joins



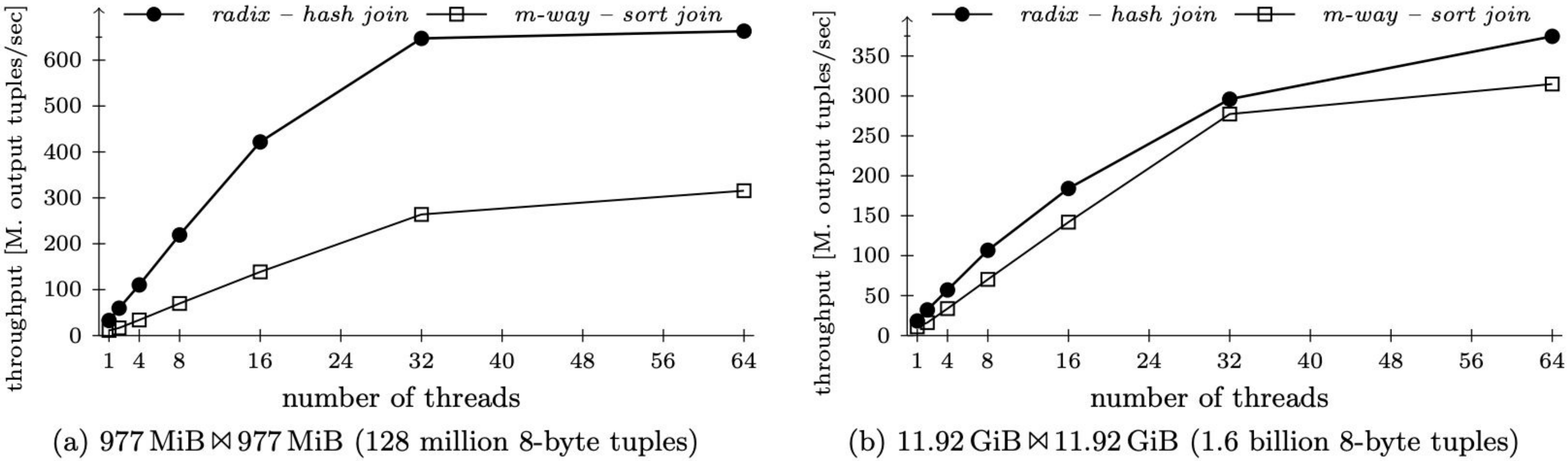


Sort vs. Hash



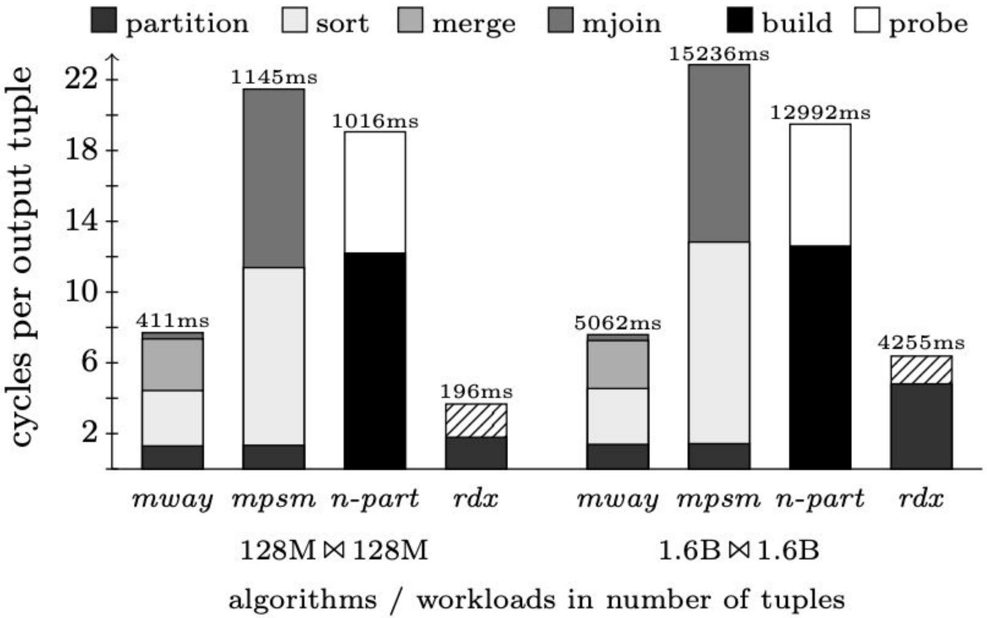
Input Size: Radix seems better

Sort vs. Hash



Scalability: Both exhibit almost linear scalability

Sort vs. Hash


partition
sort
merge
mjoin
build
probe
cycles per output tuple
22
18
14
10
6
2
411ms
1145ms
1016ms
196ms
5062ms
15236ms
12992ms
4255ms
mway
mpsm
n-part
rdx
mway
mpsm
n-part
rdx
128M ⋈ 128M
1.6B ⋈ 1.6B
algorithms / workloads in number of tuples

Summary of Results

- Input sizes have a big effect on performance
- Winner: hash-join (for now)

Concluding Thoughts

Strengths/Weaknesses

Strengths

- Develop fastest sort-merge and hash-join algorithms
- Hash join buffers enable partitioning larger data in single pass

Weaknesses

- Would have been nice to see evaluation of partition sort
- Paper layout could be more clear

Discussion Questions

- How would you expect the results of sort with partition to compare to sort-merge?
    - How would the results compare with hash-join?
- What implications do you think future hardware developments will have on the choice between sort-merge and hash-join?
- How do you view the fate of hash-join as hardware advancements result in wider registers?