Parallel Joins in Main-Memory Data(base) Processing

Alfons Kemper, et al.

Fakultät für Informatik, TUM

Joining two Relations R and S: R $\bowtie$ S

R		
A	B	C
a_1	b_1	c_1
a_2	b_2	c_2
a_3	b_3	c_1
a_4	b_4	c_2
a_5	b_5	c_3
a_6	b_6	c_2
a_7	b_7	c_6

$\bowtie$

S		
C	D	E
c_1	d_1	e_1
c_3	d_2	e_2
c_4	d_3	e_3
c_5	d_4	e_4
c_7	d_5	e_5
c_8	d_6	e_6
c_5	d_7	e_7

$=$

$R \bowtie S$				
A	B	C	D	E
a_1	b_1	c_1	d_1	e_1
a_3	b_3	c_1	d_1	e_1
a_5	b_5	c_3	d_2	e_2

Algorithmics:

$R \cap S$

R	S
2	44
3	17
44	97
5	5
78	6
90	27
13	2
17	13
42	9
89	

Algorithmics:

$R \cap S$

R

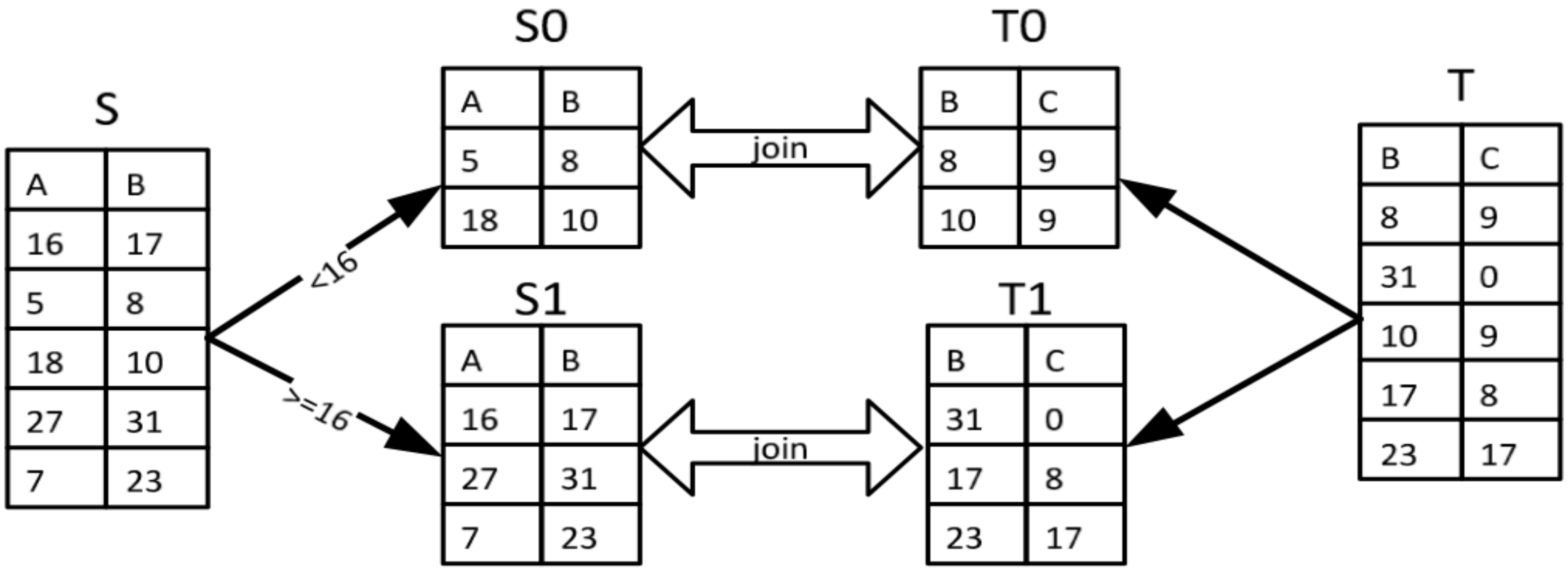
2
3
44
5
78
90
13
17
42
89

- Nested Loop: $O(N^2)$
- Sort-Based: $O(N \log N)$
- Partitioning and Hashing

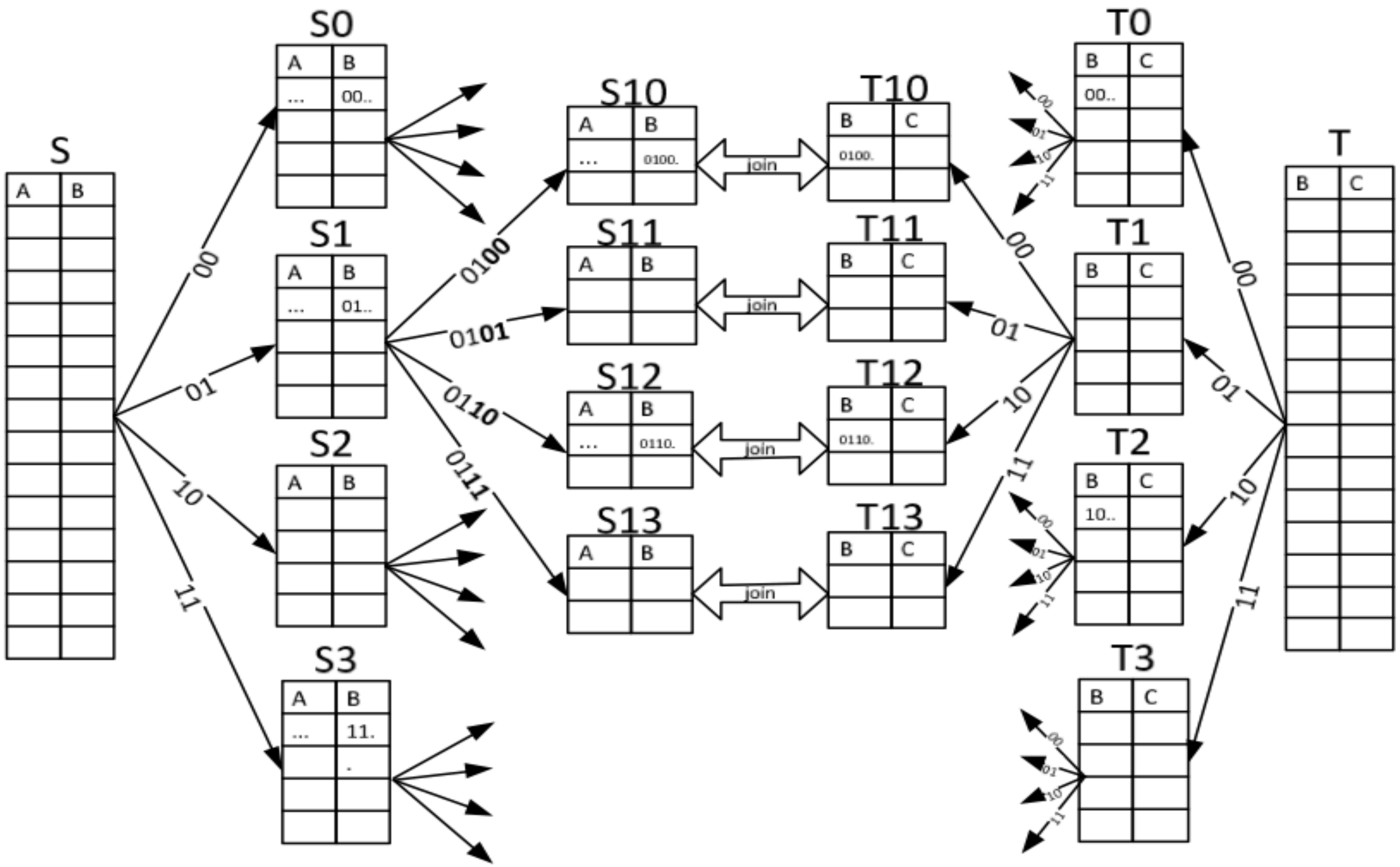
S

44
17
97
5
6
27
2
13
9

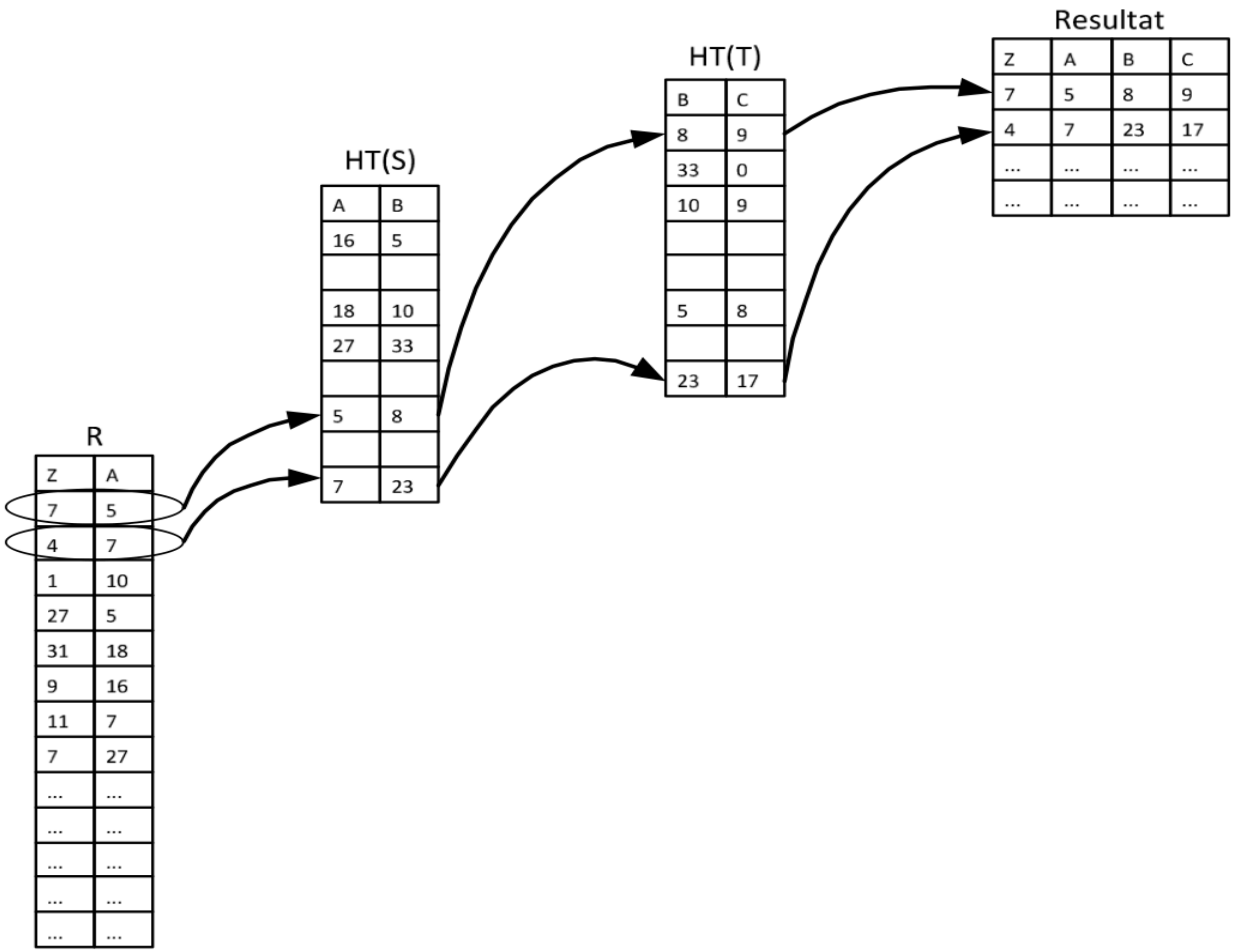
Parallel Radix-Join



Multiple Phase Radix-Joins: Cache-Locality



Hash-Join-Teams: Global Hash Table



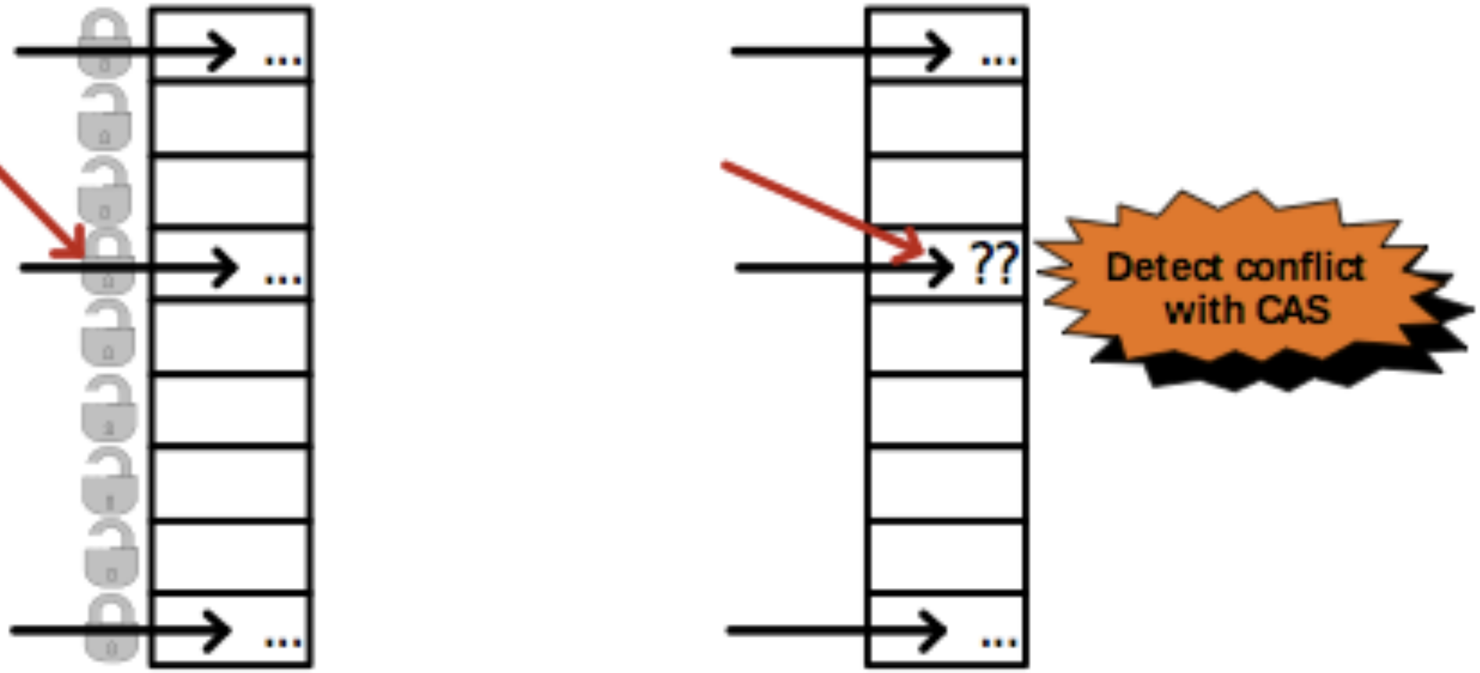
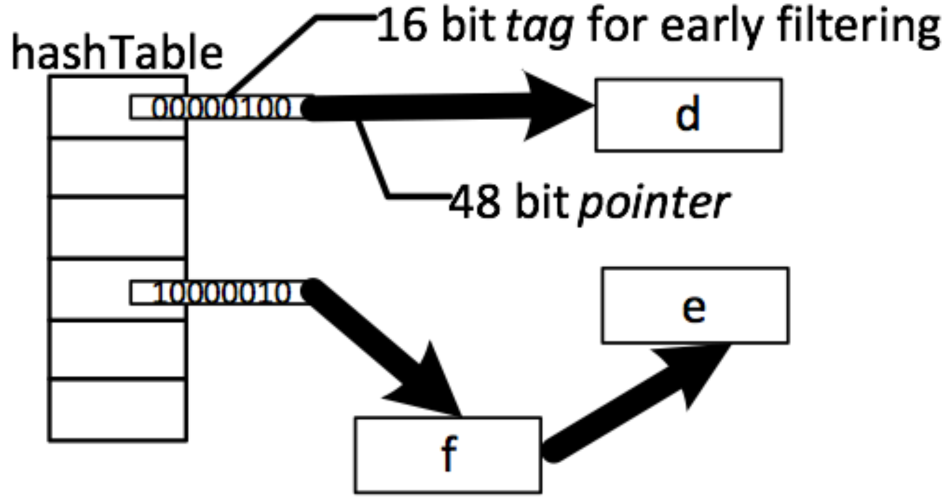


Fig. 1: Pessimistic vs. optimistic write access to a hash table

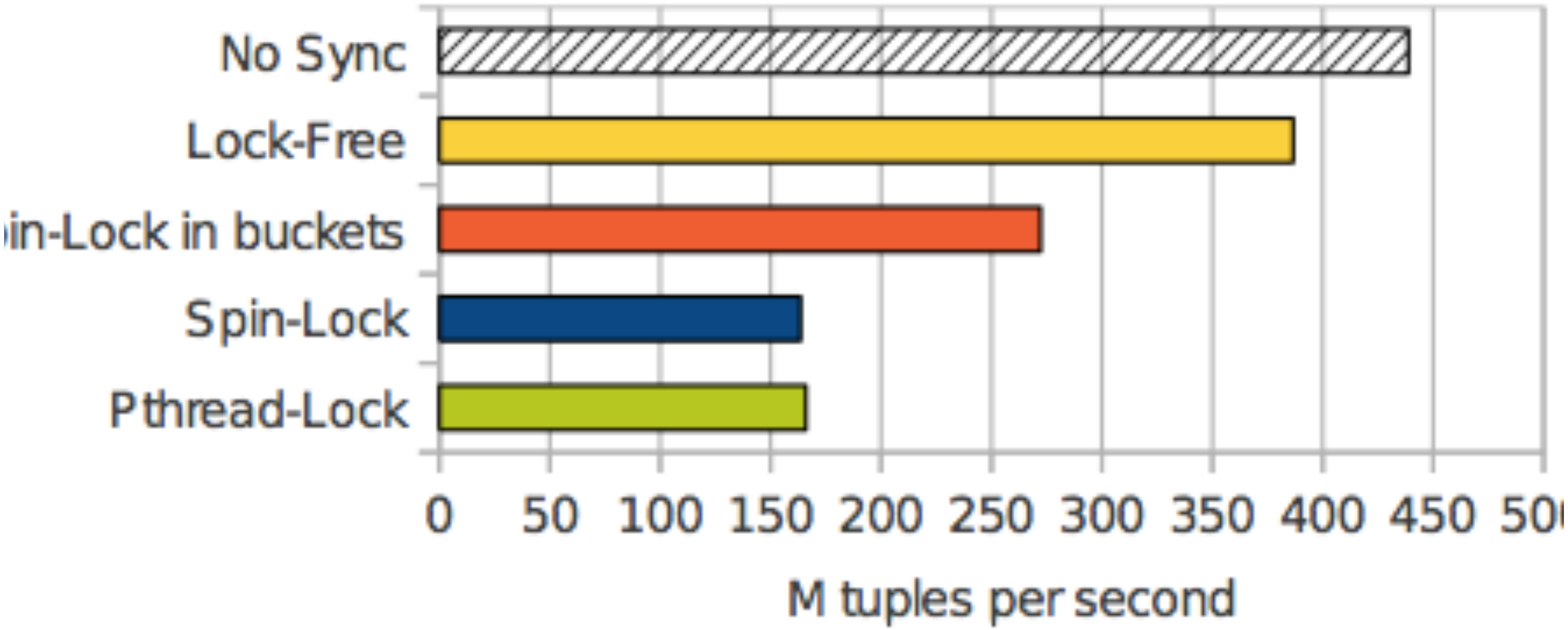
Listing 1.1: Atomic insert function

```
insertAtomic(uint64_t key, uint64_t value) {
  uint64_t hash = hashFunction(key);
  uint64_t pos = hash & mask;
  while (table[pos].h != 0
      || (! CAS(&table[pos].h, 0, hash))) {
    pos = (pos + 1) & mask;
  }
  table[pos].k = key;
  table[pos].v = value;
}
```

Alternative Hash Table with Overflow Bu




```
insert(entry) {
  // determine slot in hash table
  slot = entry->hash >> hashTableShift
  do {
    old = hashTable[slot]
    // set next to old entry without tag
    entry->next = removeTag(old)
    // add old and new tag
    new = entry | (old&tagMask) | tag(entry->hash)
    // try to set new value, repeat on failure
  } while (!CAS(hashTable[slot], old, new))
}
```



Build performance using different synchronization mechar

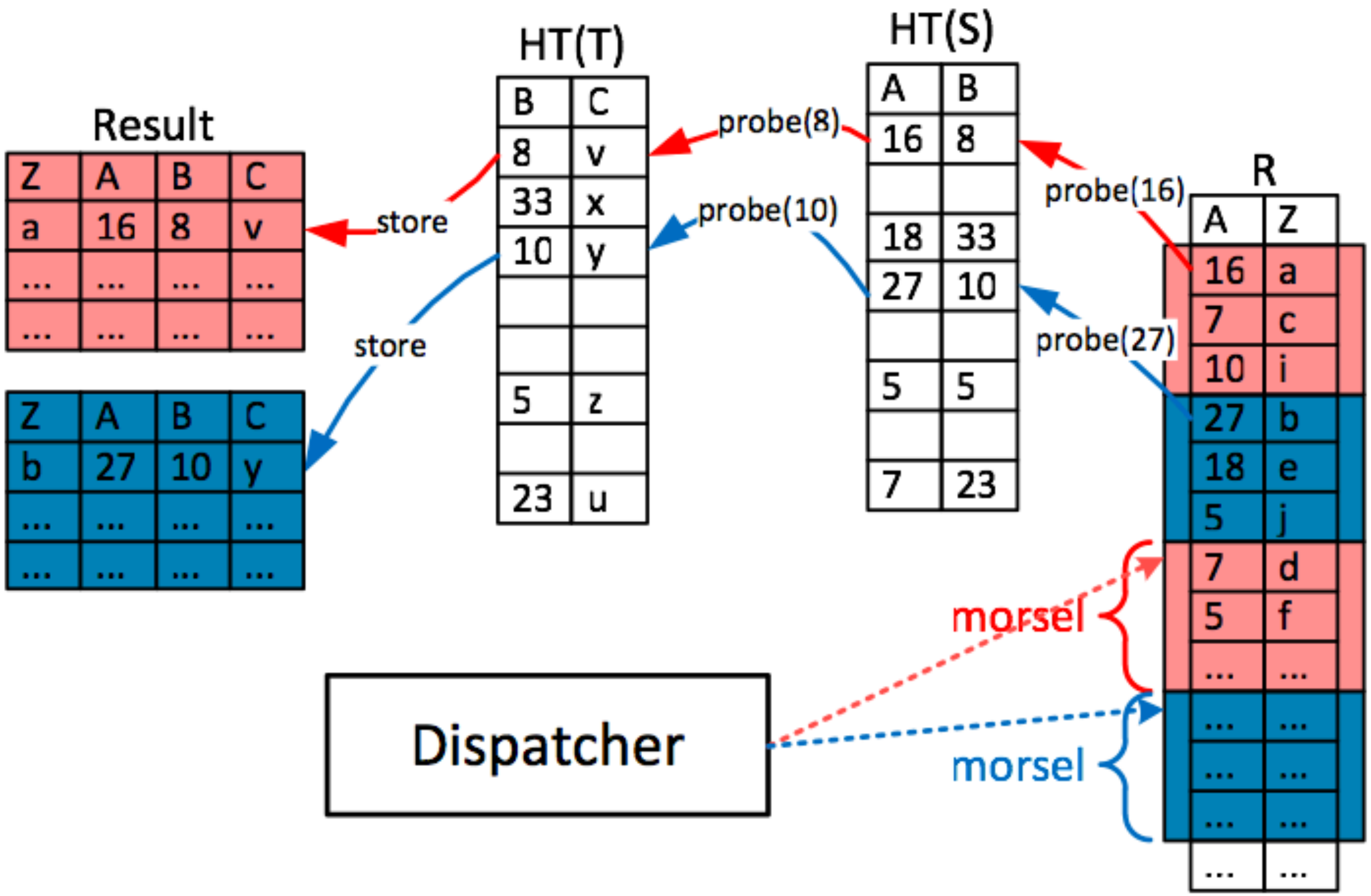
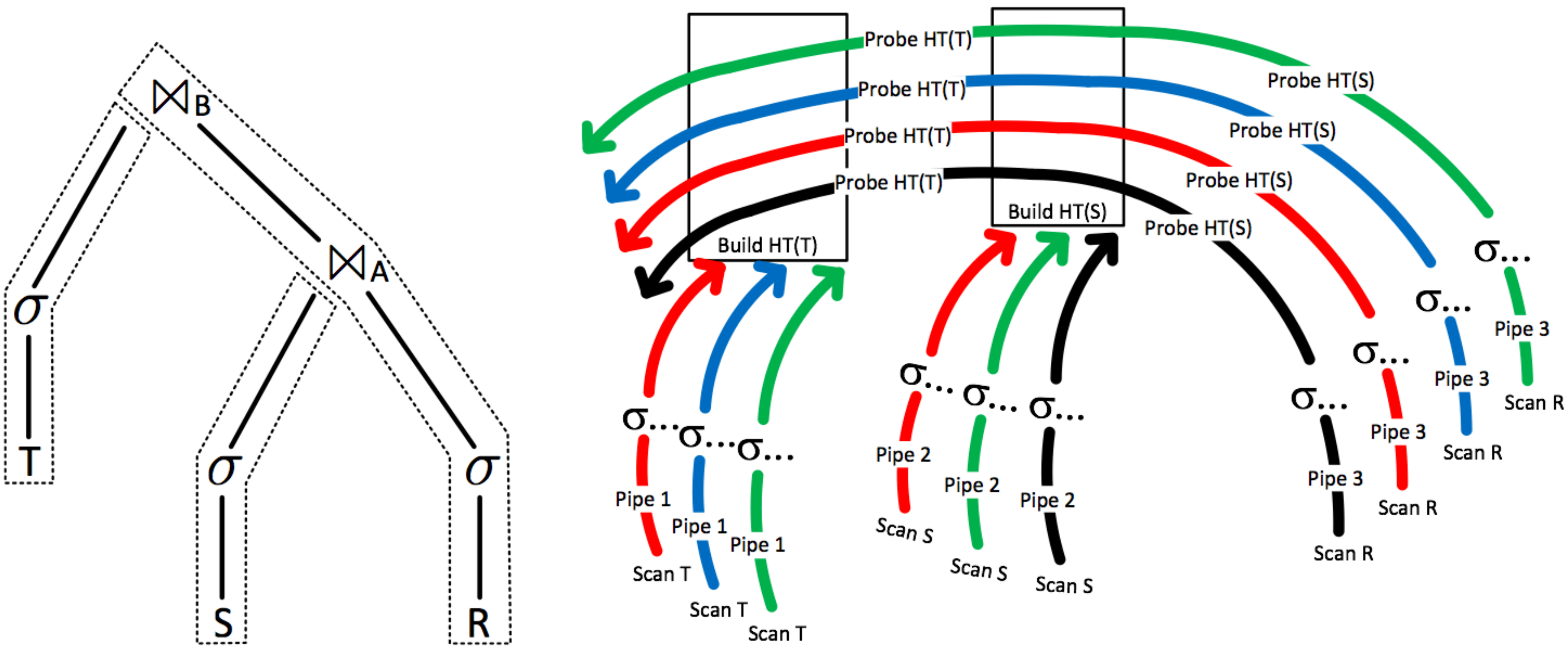


Figure 1: Idea of morsel-driven parallelism: $R \bowtie_A S \bowtie_B T$

Massively parallel processing



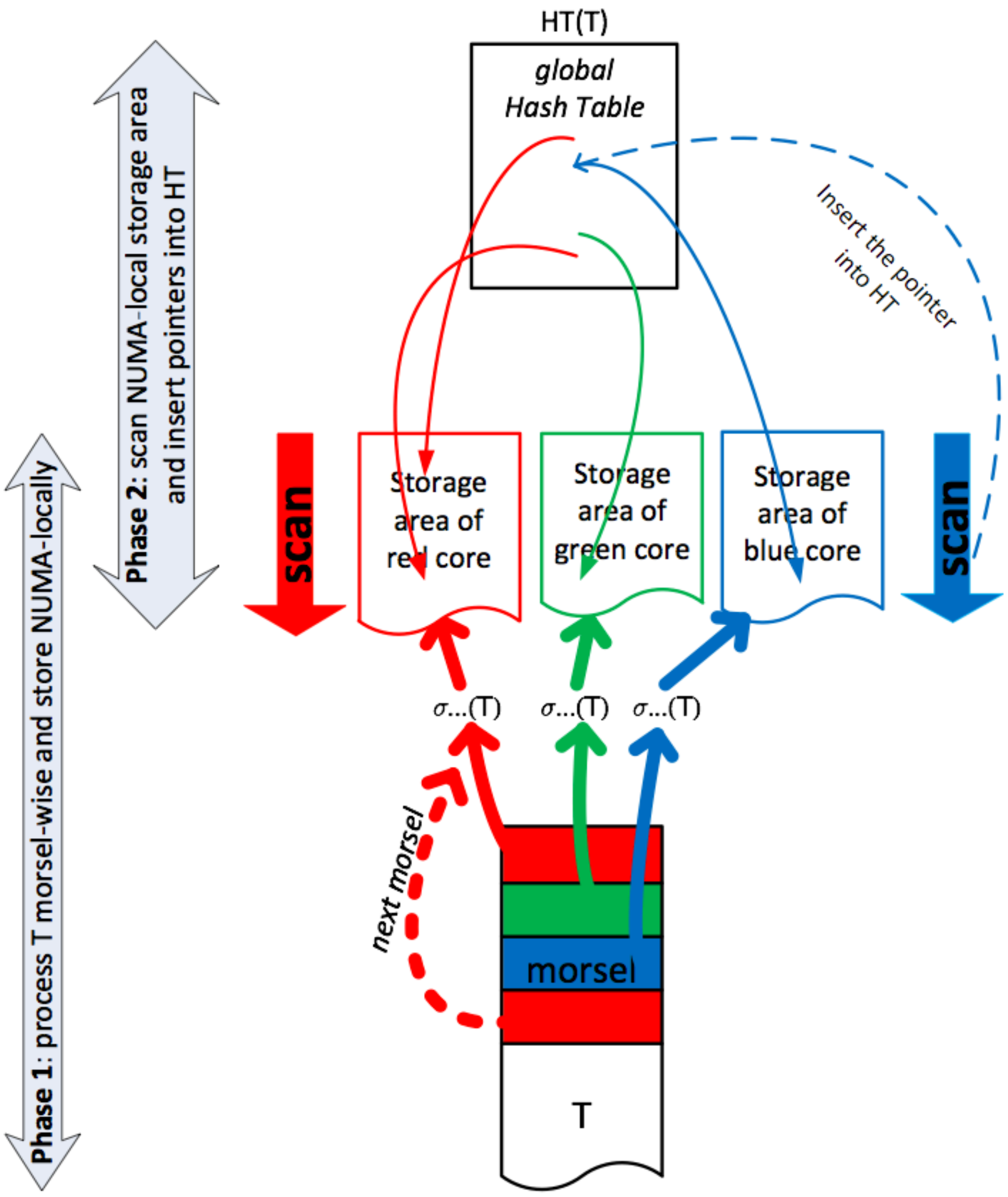


Figure 3: NUMA-aware processing of the build-phase

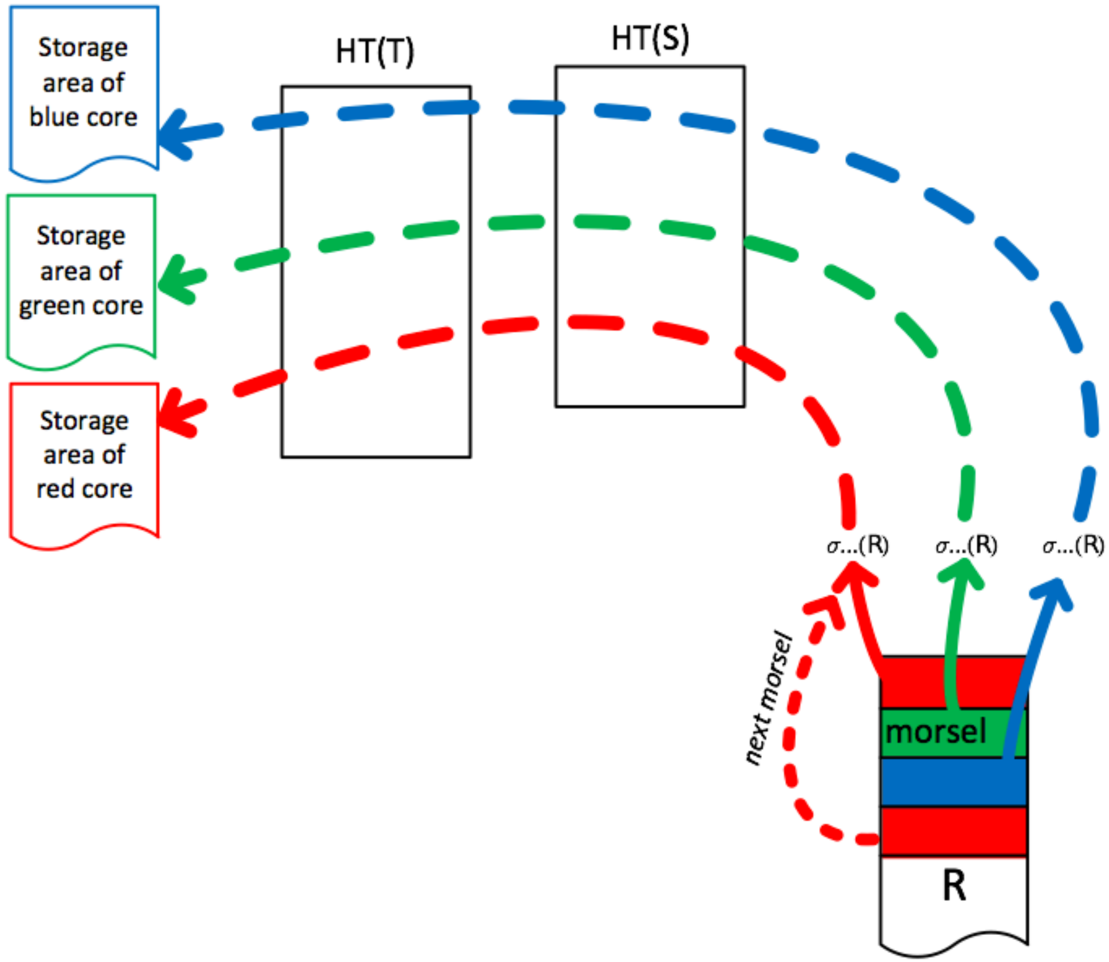
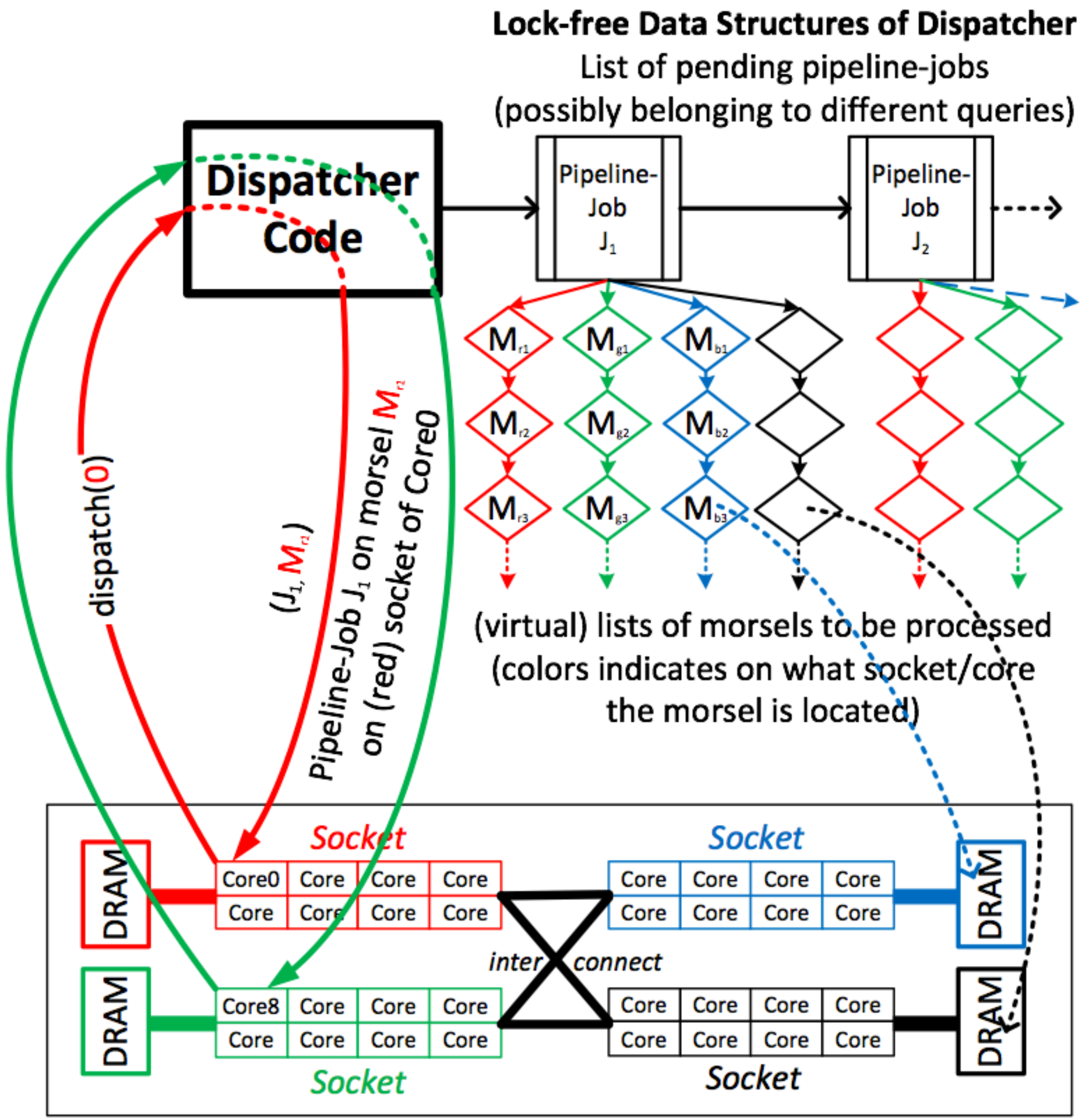


Figure 4: Morsel-wise processing of the probe phase



Example NUMA Multi-Core Server with 4 Sockets and 32 Cores

Massively Parallel Hash Aggregation

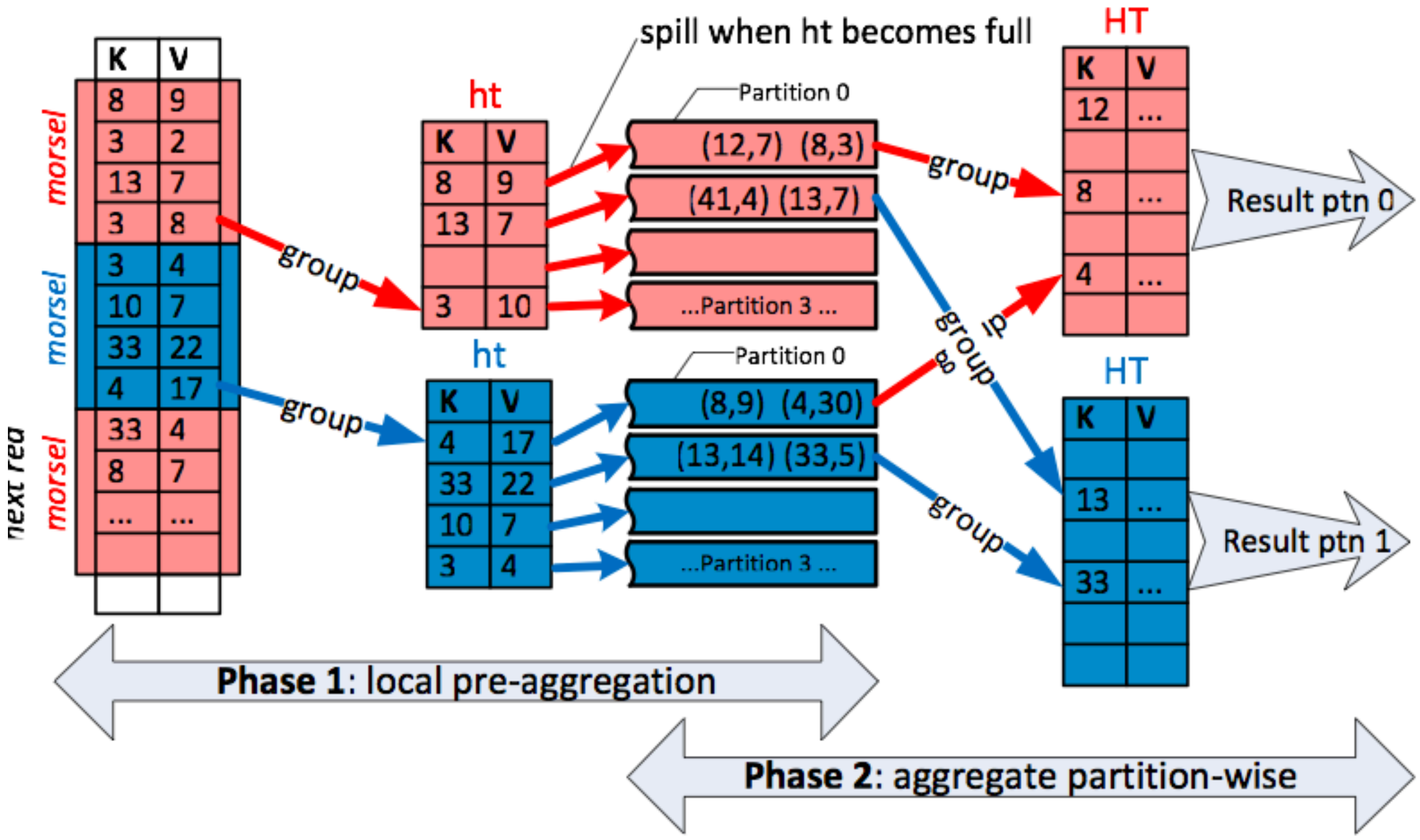


Figure 8: Parallel aggregation

Sorting in Parallel

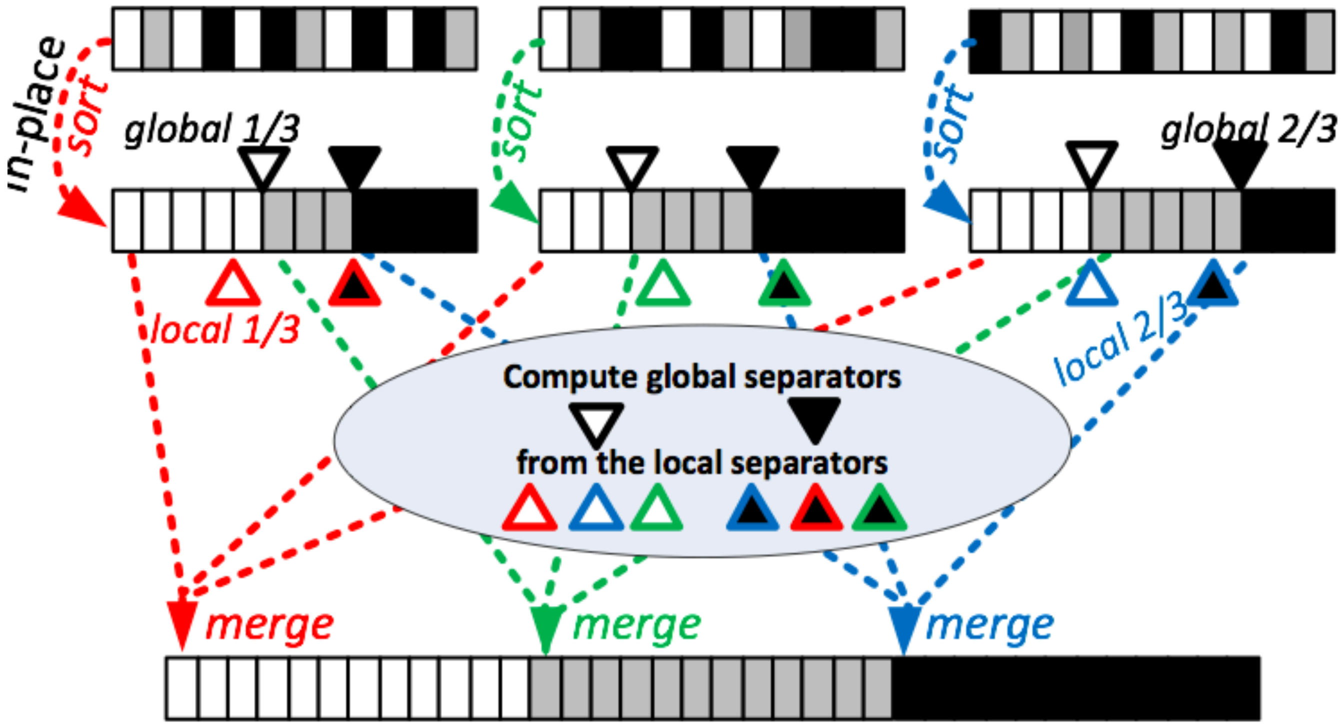


Figure 9: Parallel merge sort

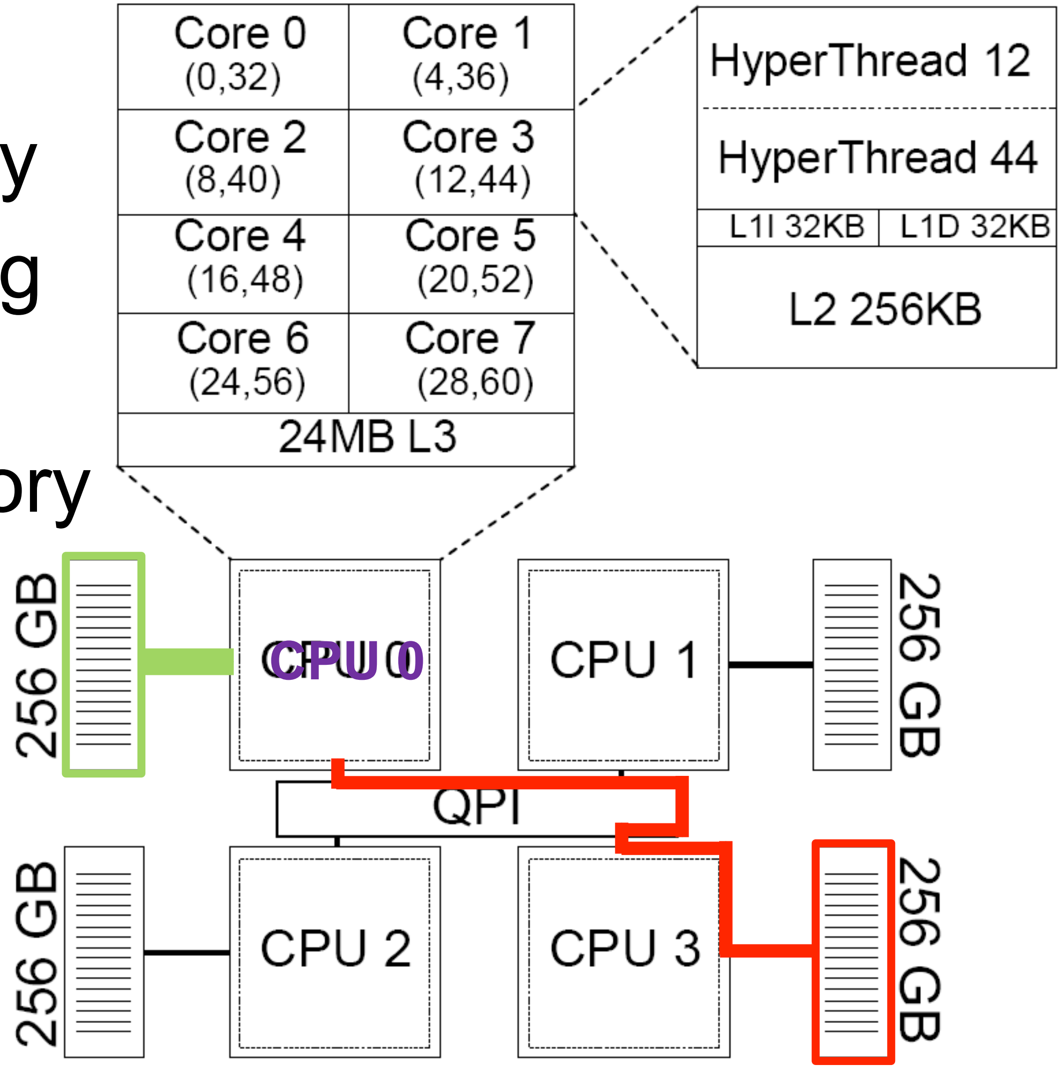
Massively Parallel Sort-Merge Joins (MPSM) in Main Memory Multi-Core Database Systems

Martina Albutiu, Alfons Kemper, and Thomas Neumann

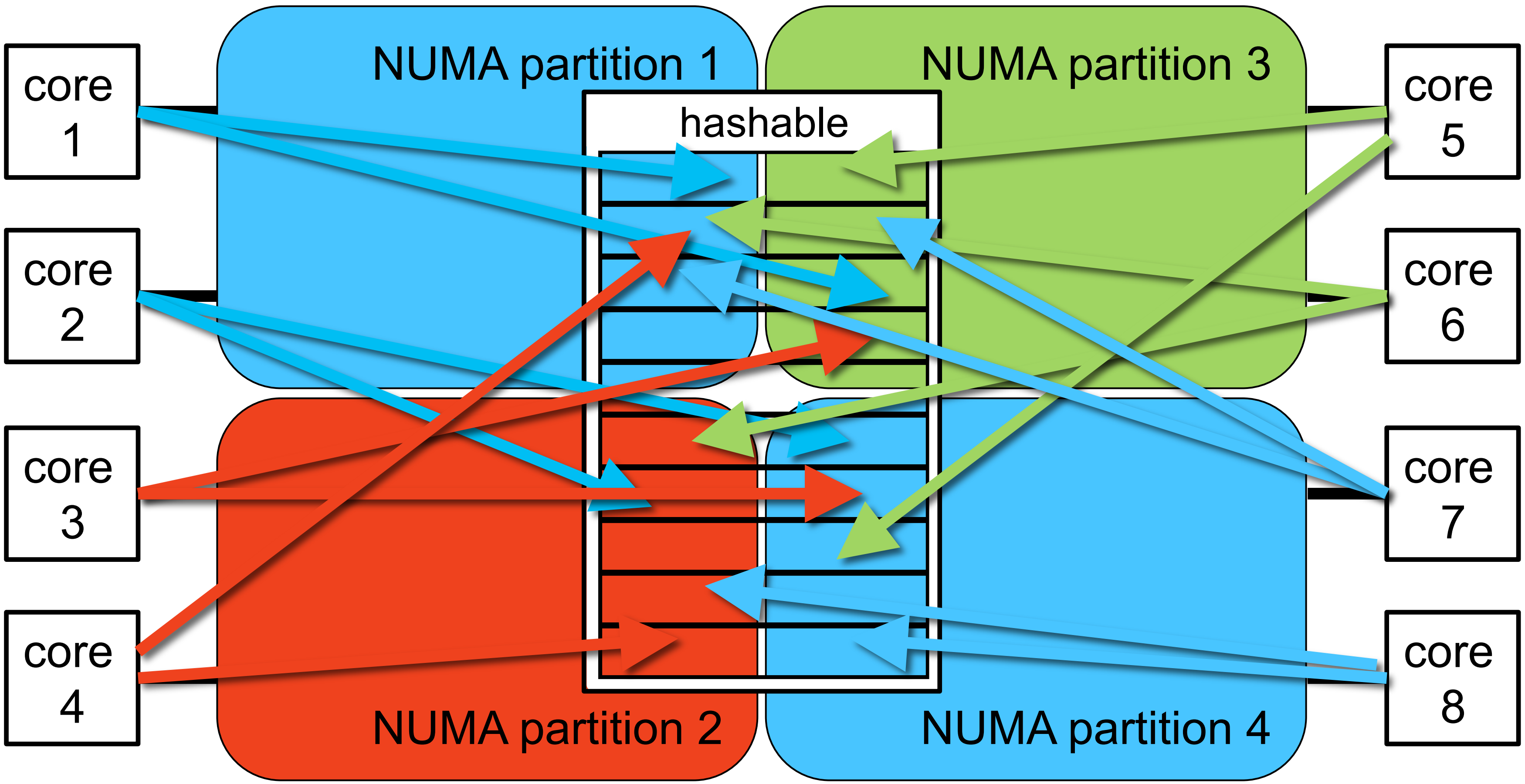
Technische Universität München

Hardware trends …

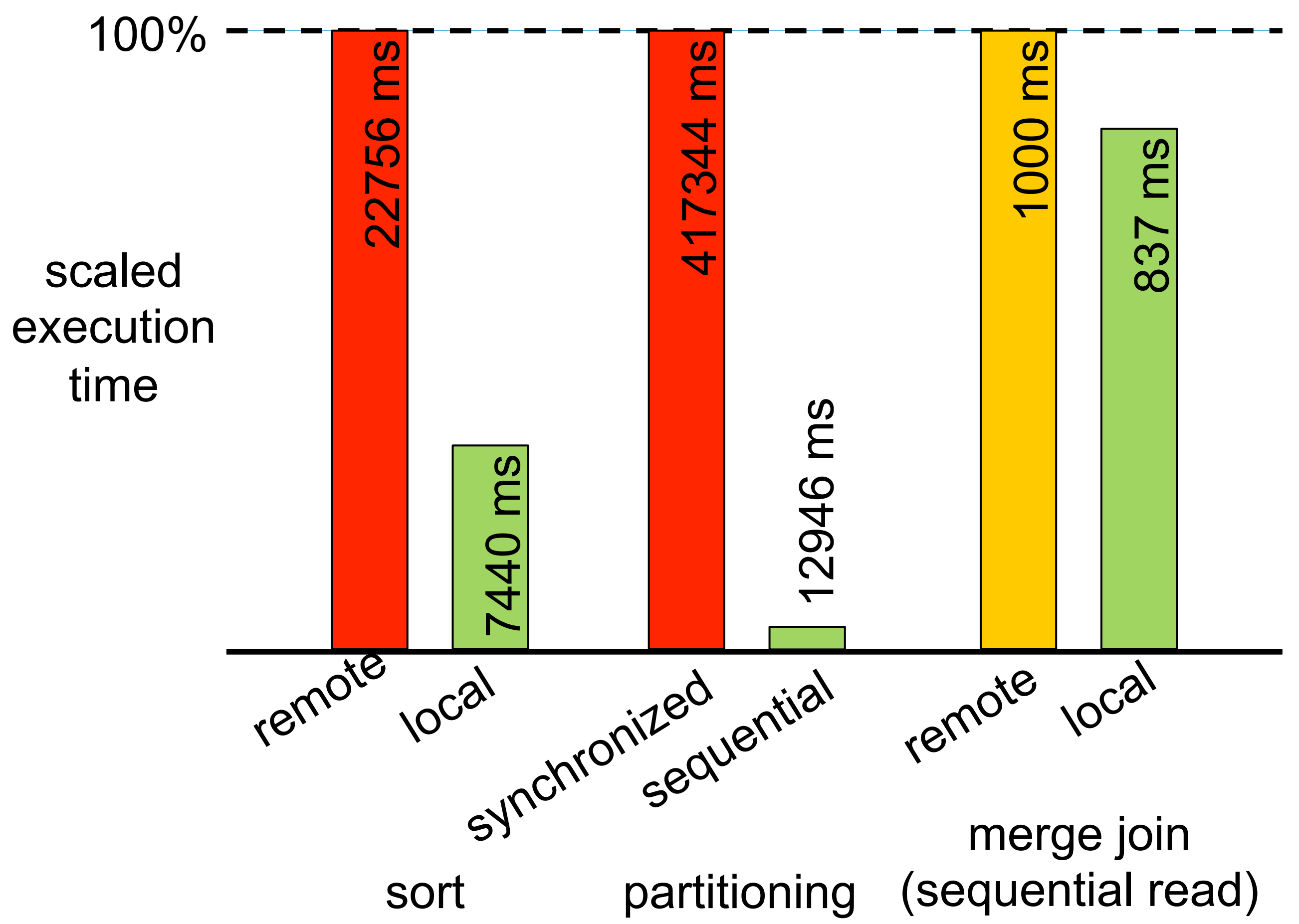
- Huge main memory
- Massive processing parallelism
- Non-uniform Memory Access (NUMA)
- Our server:
  - 4 CPUs
  - 32 cores
  - 1 TB RAM
  - 4 NUMA partitions



Ignoring NUMA



How much difference does NUMA make?



The three NUMA commandments

C1

Thou shalt not write thy neighbor's memory randomly

-- chunk the data, redistribute, and then sort/

C2

Thou shalt read thy neighbor's memory only sequentially

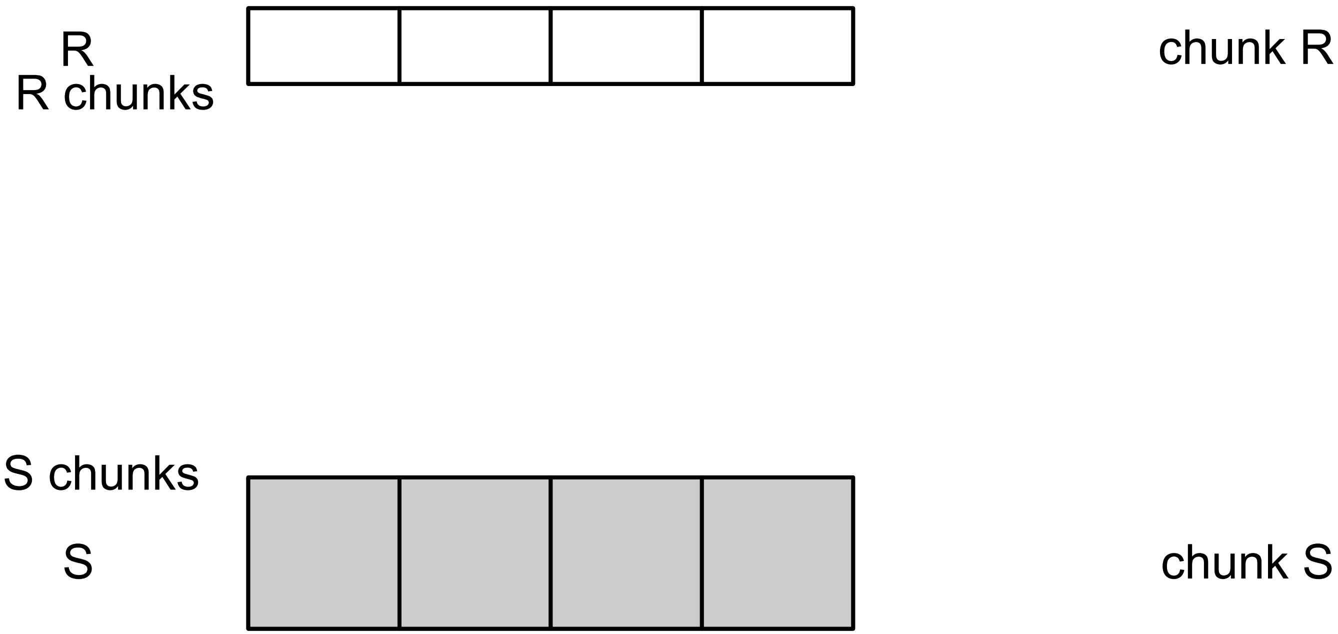
-- let the prefetcher hide the remote access latency.

C3

Thou shalt not wait for thy neighbors

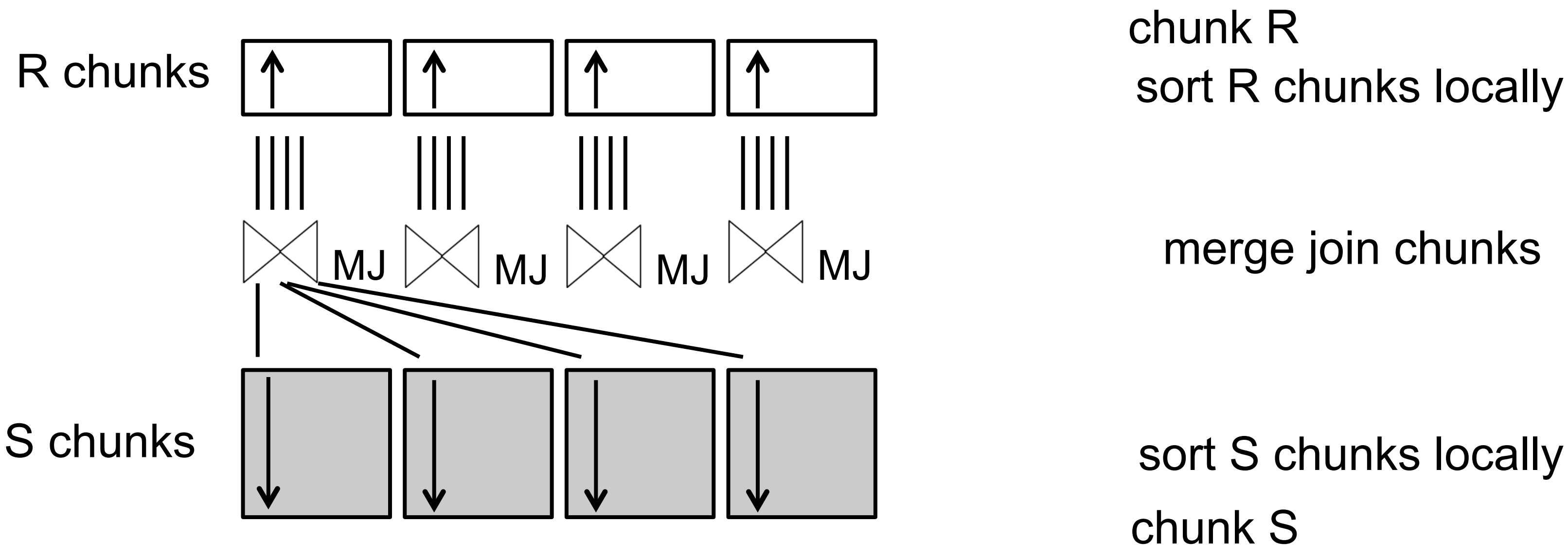
-- don't use fine-grained latching or locking and avoid synchronization points of parallel threads.

Basic idea of MPSM



Basic idea of MPSM

- C1: Work locally: sort
- C3: Work independently: sort and merge join
- C2: Access neighbor's data only sequentially

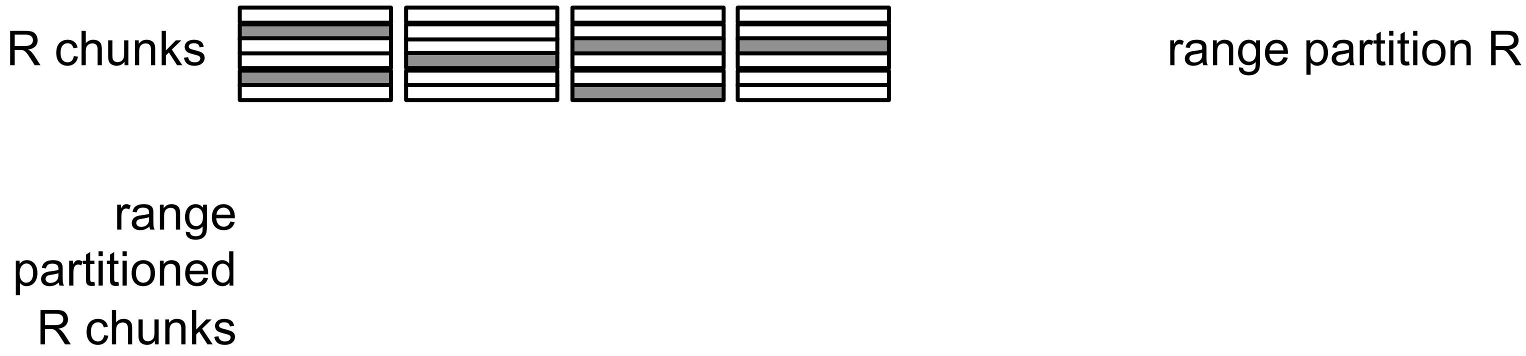


Range partitioning of private input R

- To constrain merge join work
- To provide scalability in the number of parallel workers

Range partitioning of private input R

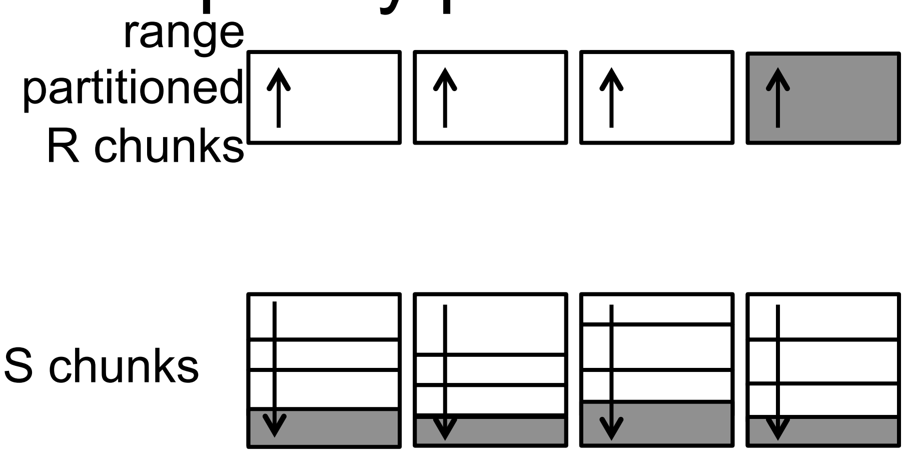
- To constrain merge join work
- To provide scalability in the number of parallel workers



Range partitioning of private input R

- To constrain merge join work
- To provide scalability in the number of parallel workers

→ S is implicitly partitioned



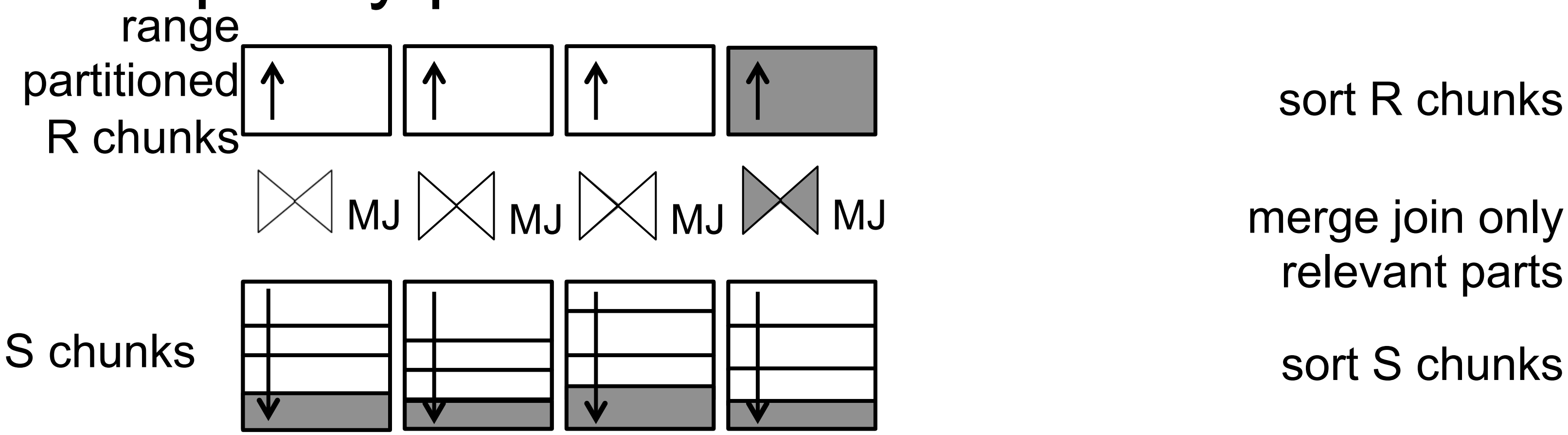
sort R chunks

sort S chunks

Range partitioning of private input R

- To constrain merge join work
- To provide scalability in the number of parallel workers

→ S is implicitly partitioned



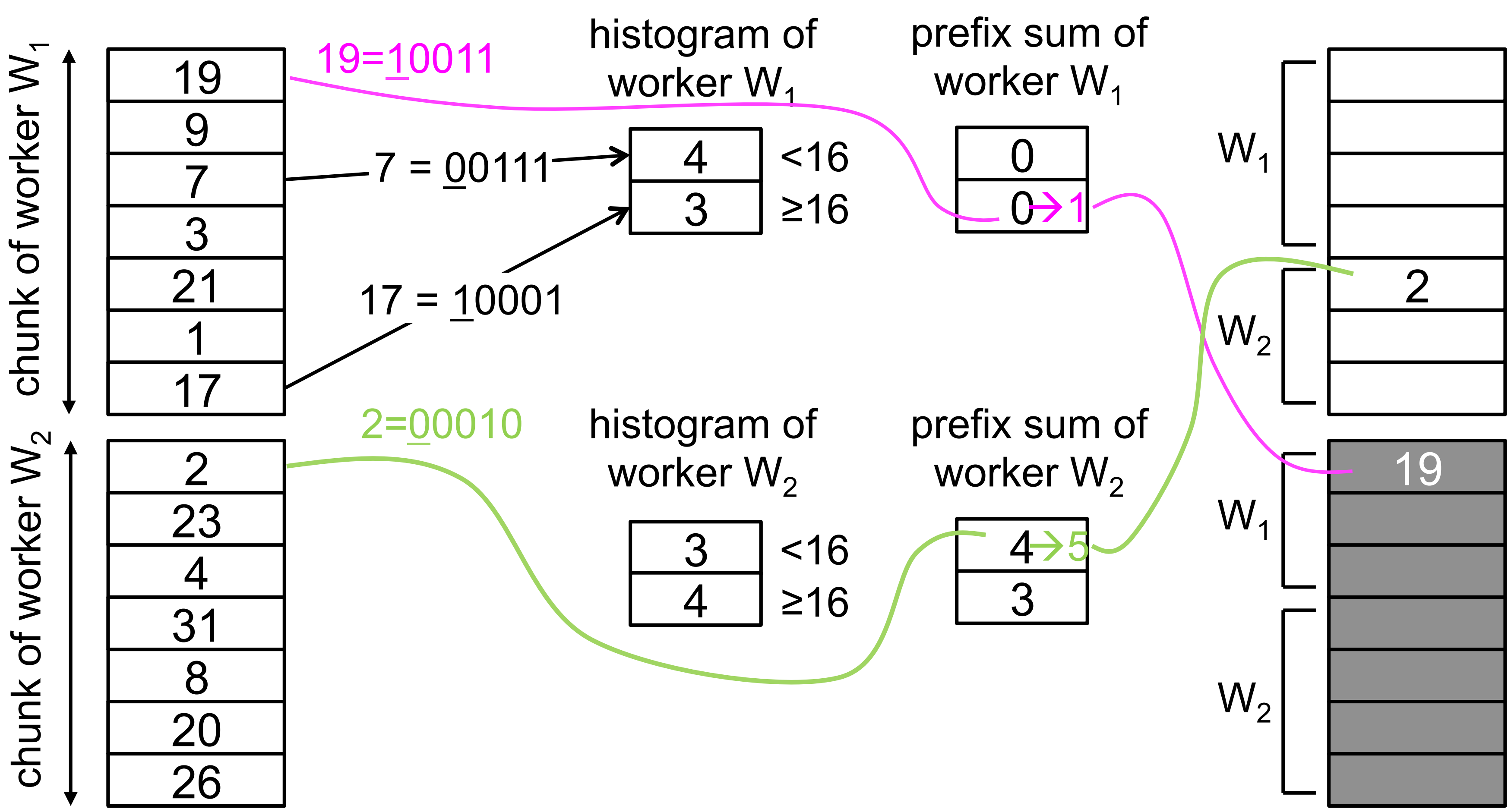
Range partitioning of private input

- Time efficient
  → branch-free
  → comparison-free
  → synchronization-free

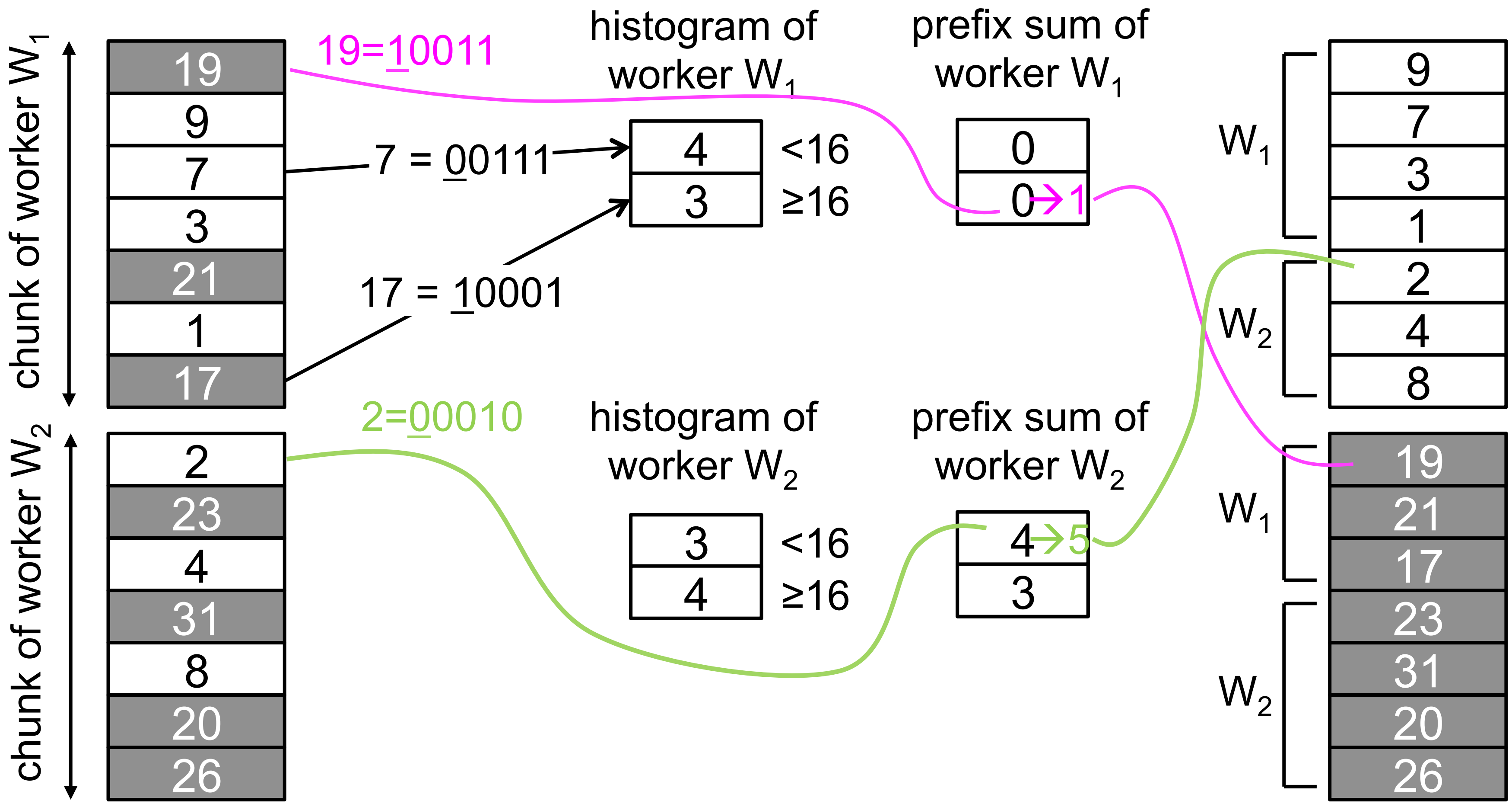
  and
- Space efficient
  → densely packed
  → in-place

by using radix-clustering and precomputed target partitions to scatter data to

Range partitioning of private input



Range partitioning of private input



Real C hacker at work …

$$ps_i[j] = \&R_j[(\sum_{k=1}^{i-1} h_k[j])]$$

$$\mathtt{memcpy}(ps_i[sp[t.key \gg (64 - B)]]\texttt{++}, t, t.size)$$