Morsel-Driven Parallelism: A NUMA-Aware Query Evaluation Framework for the Many-Core Age

Viktor Leis, Peter Boncz, Alfons Kemper, Thomas Neumann

Lunhao Liang, Dezhou Wang

Agenda

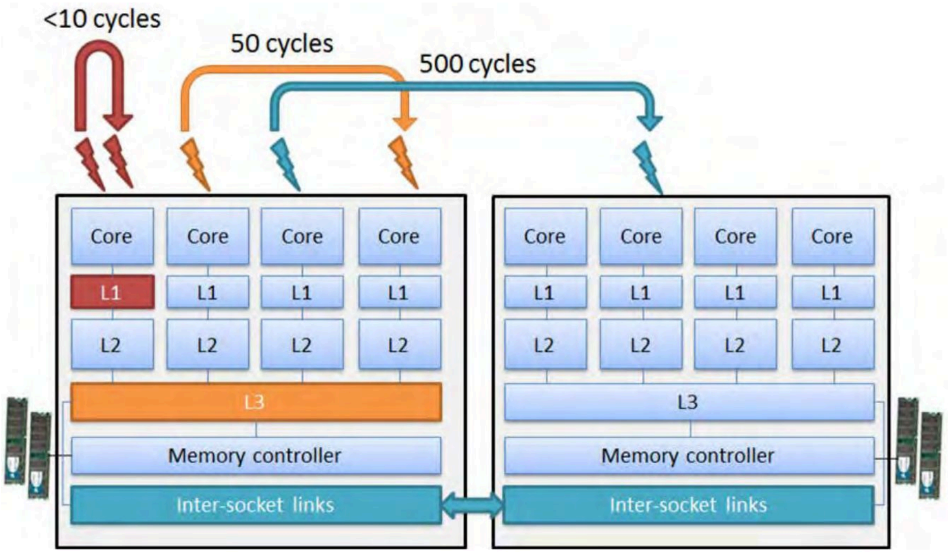
- What's the Morsel-Driven Parallelism?
- How does Morsel-Driven execute
- How does Dispatcher schedule parallel pipeline tasks
- Parallel Operator Details
- Evaluation
- Conclusion

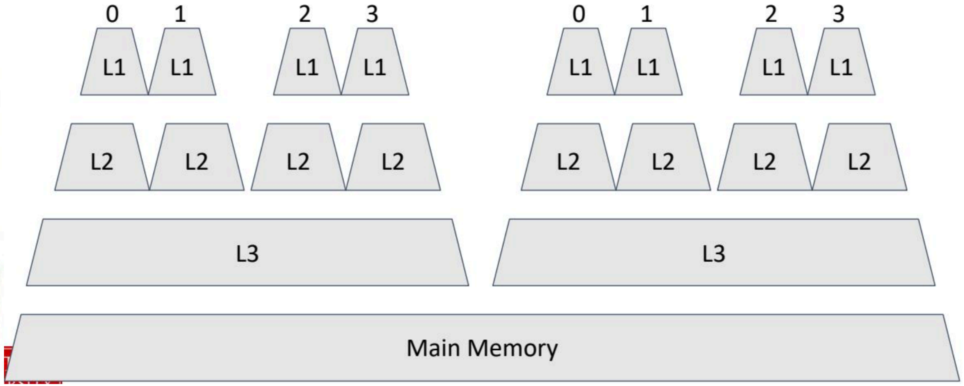
What’s the Morsel-Driven Parallelism?

Problems for modern parallel query

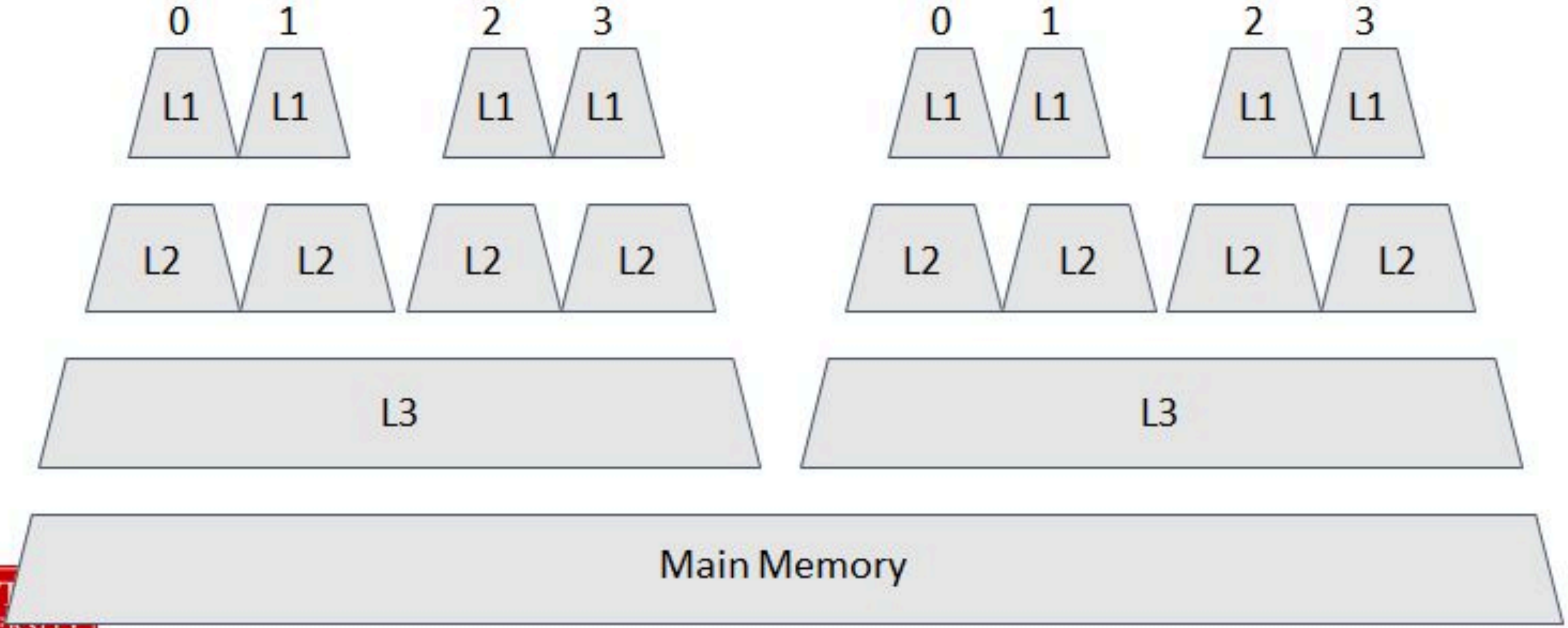
Modern computer architecture evolves

Multi Cores CPU	Cache Hierarchy	NUMA

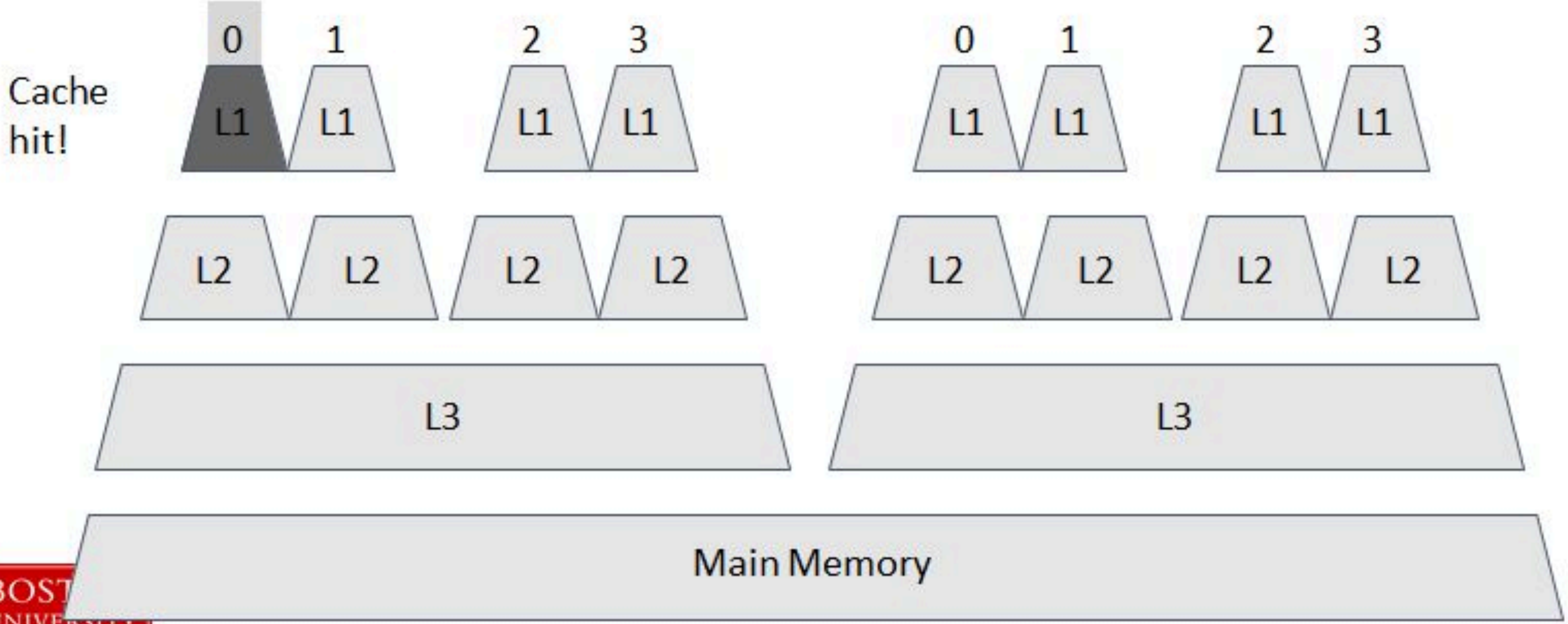




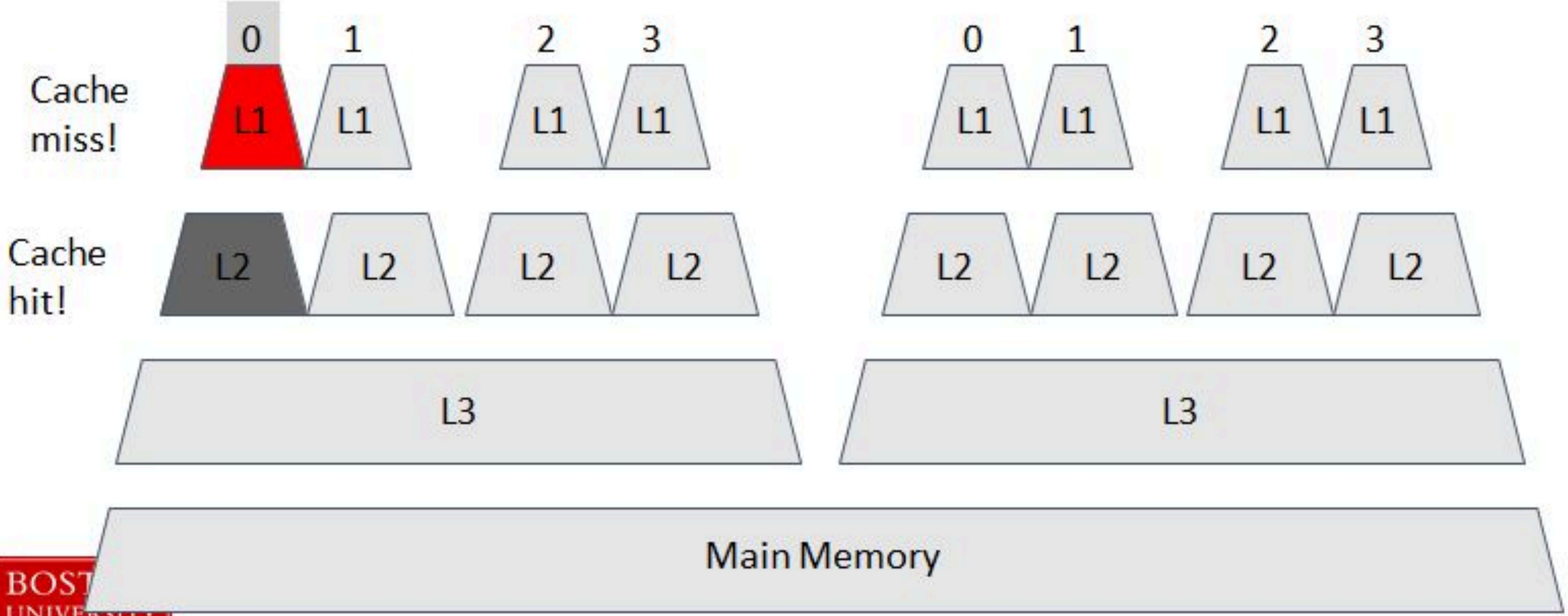
Non Uniform Memory Access (NUMA)



Non Uniform Memory Access (NUMA)

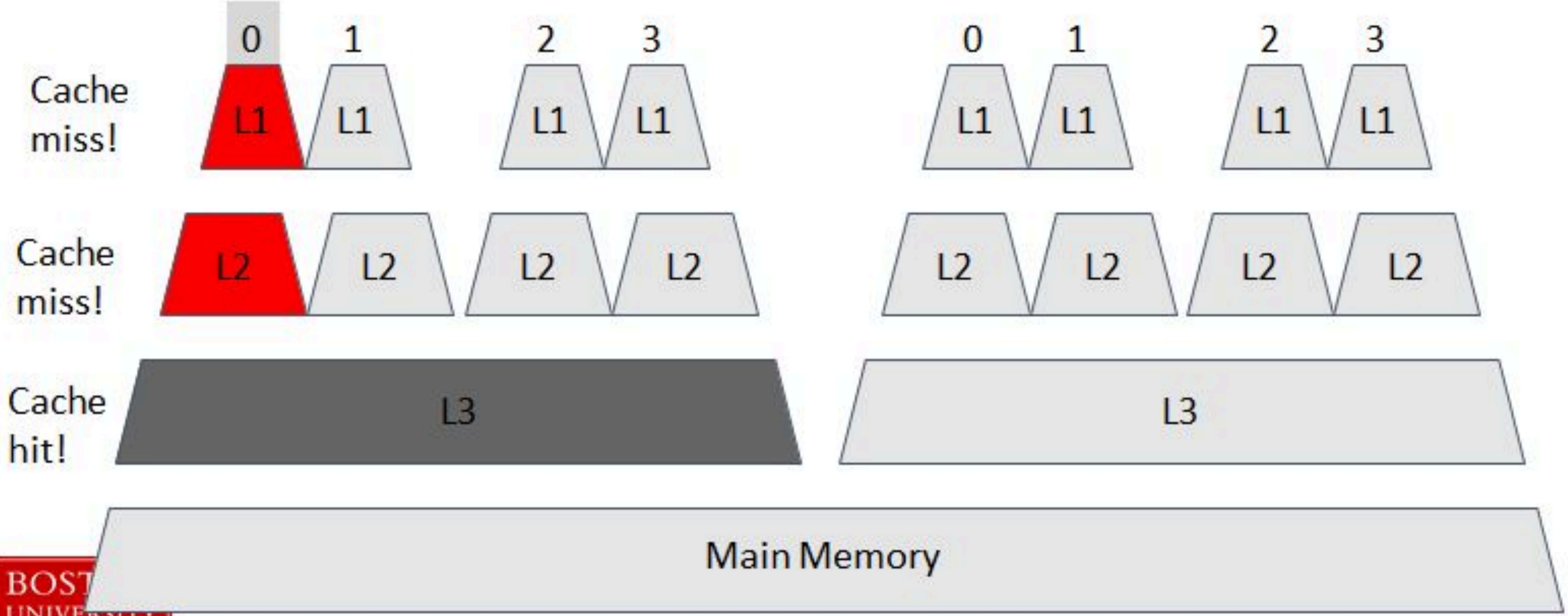


Non Uniform Memory Access (NUMA)

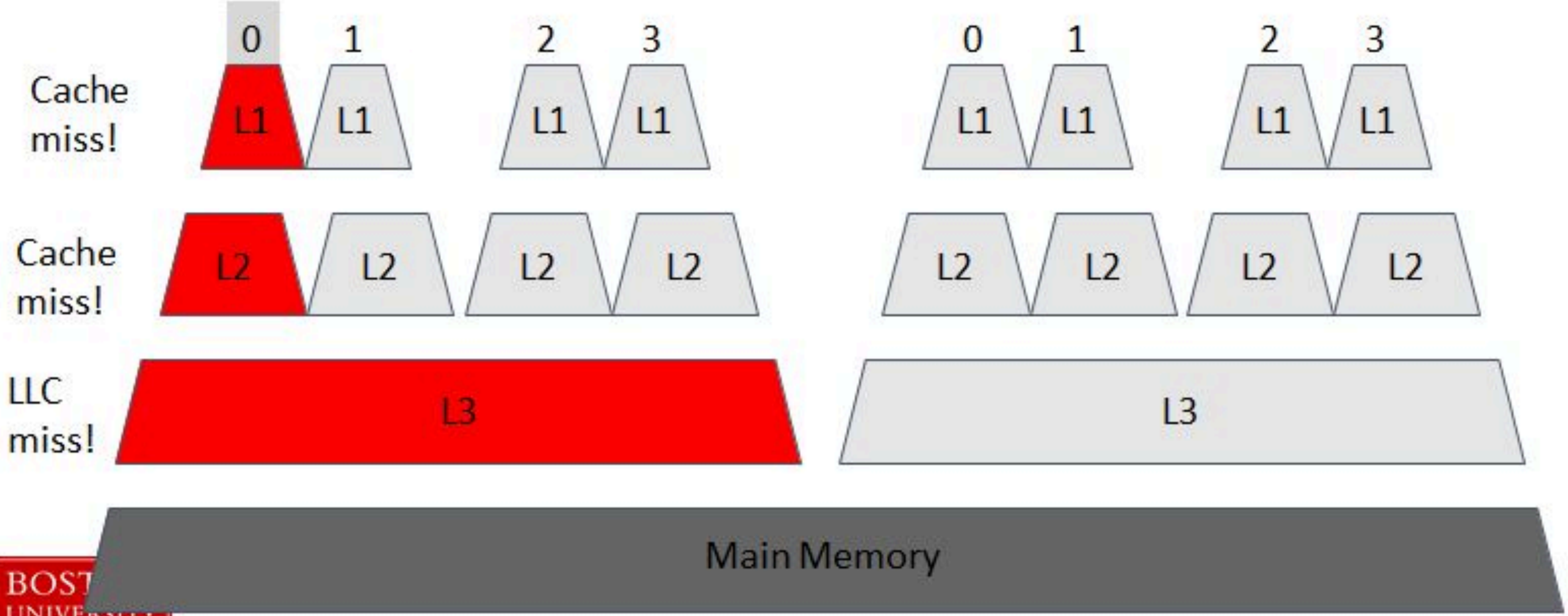


BOSTON UNIVERSITY

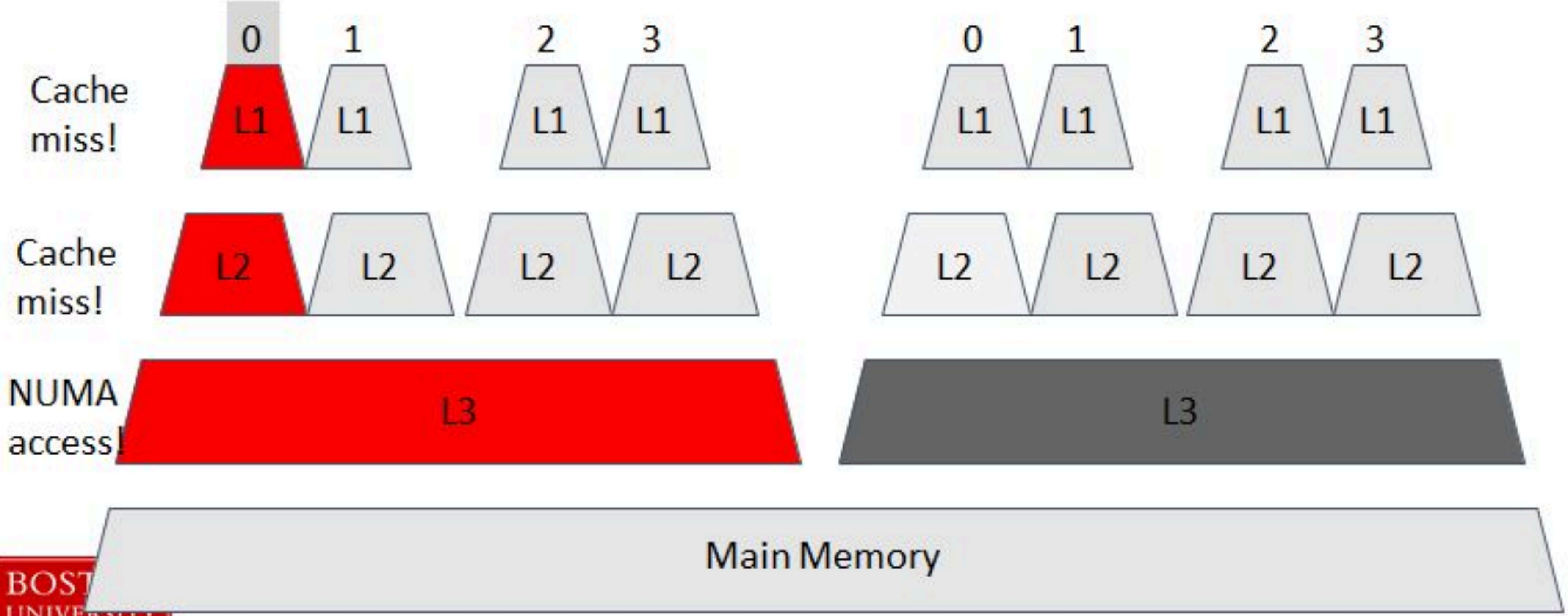
Non Uniform Memory Access (NUMA)



Non Uniform Memory Access (NUMA)



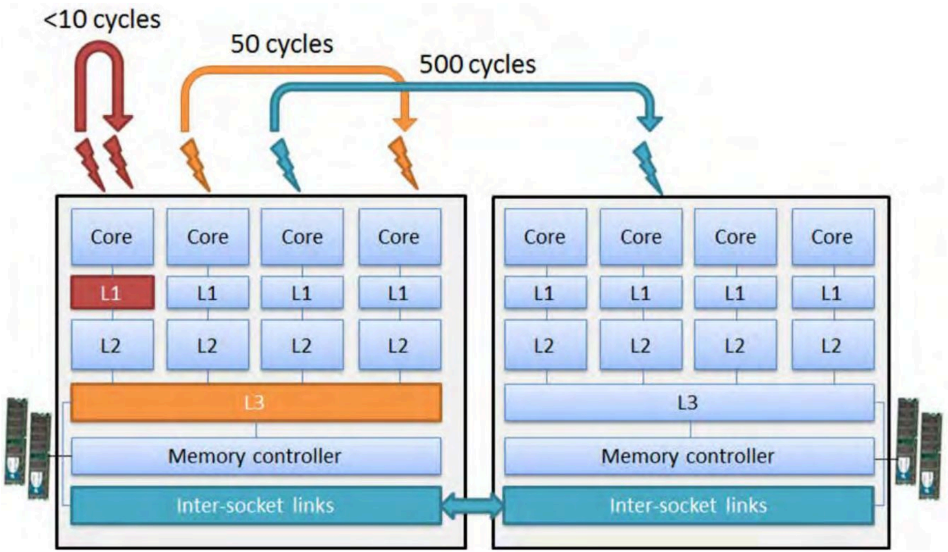
Non Uniform Memory Access (NUMA)

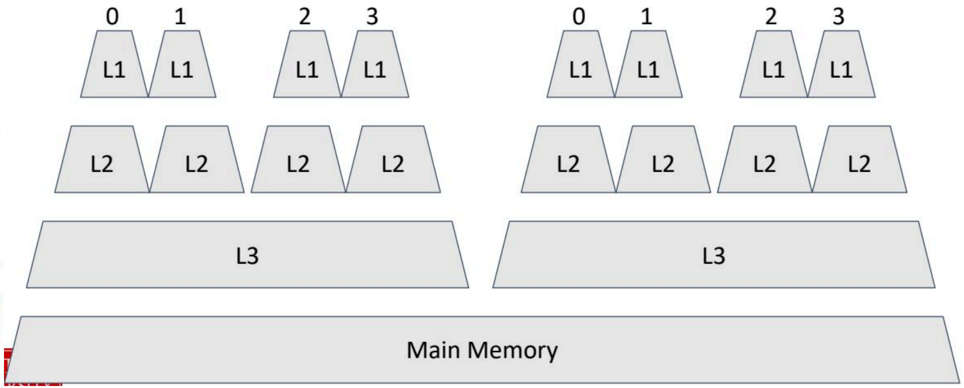


Problems for modern parallel query

Problems for modern parallel query

Only use one core for query pipeline	Algorithm can not take advantage of multi cores	Dividing the work evenly in CPU is difficult





Related research on parallel query

Volcano parallelism framework for database in 1990s (Plan-Driven)

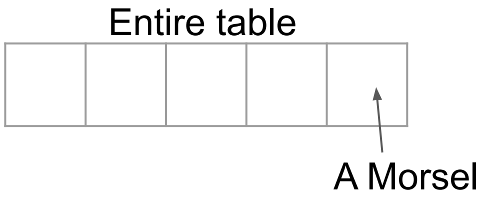
- ***Statically*** determine at query compiling time how many threads should be use
- Instantiates one query operator plan for each thread at first for one time

Core 0	Core 1	Core 2	Core 3

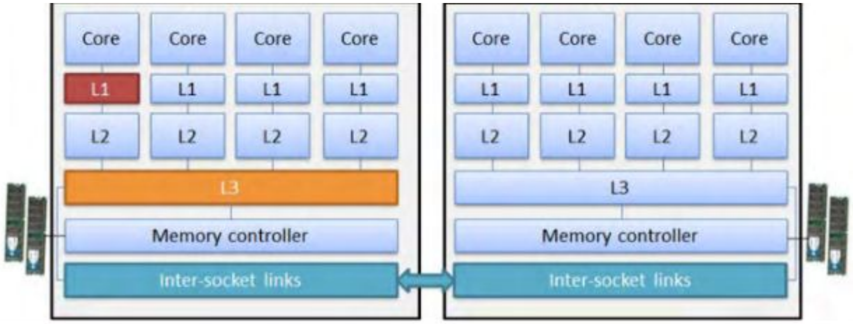
Input Data

Morsel-Driven Parallelism

The paper presents the “morsel-driven” query to solve the problem for the database “HyPer”



- ✓ Divide input data into ***small fragments*** a.k.a. “***Morsels***”
- ✓ A “Dispatcher” dispatches morsels to cores
- ✓ ***Dynamically*** and ***Elastically*** change ***parallelism*** during query execution by dispatcher
- ✓ Use ***NUMA-local*** memory



How does Morsel-Driven execute?

Morsel-Driven Parallelism

The Three-Way-Join query example in paper

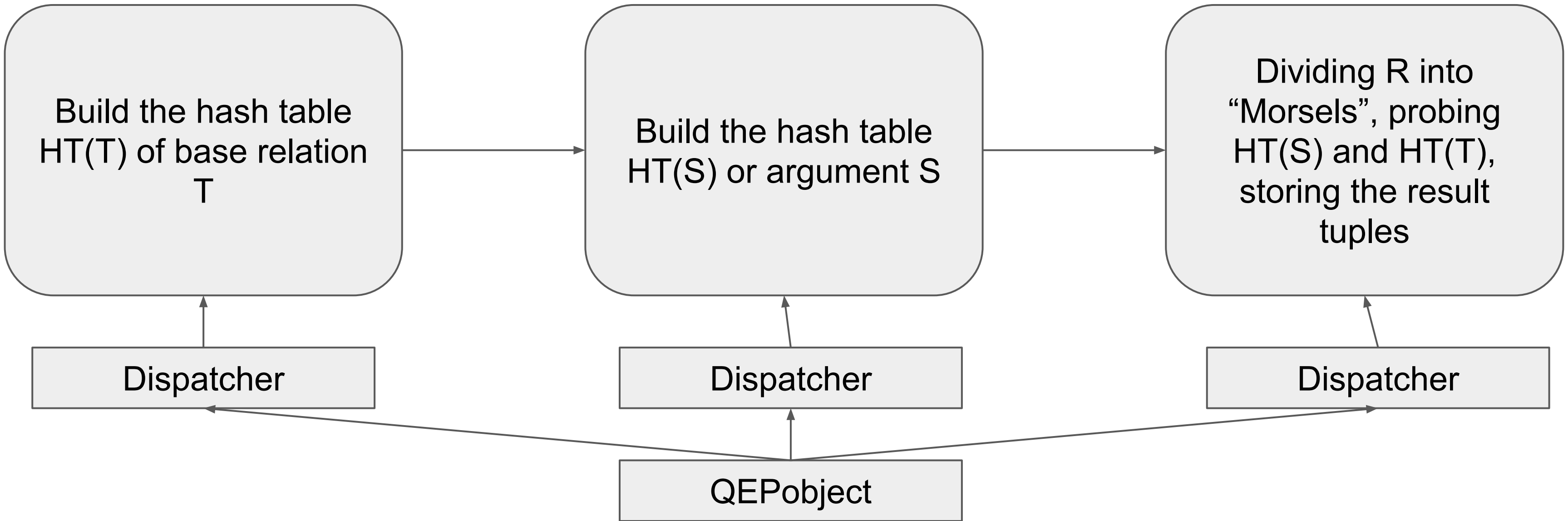
$$\sigma_{\ldots}(R) \bowtie_A \sigma_{\ldots}(S) \bowtie_B \sigma_{\ldots}(T)$$

- Table R is the largest table (optimizer chooses R as probe input)
- T is the Base Relation table, S is the Argument table
- Slicing the R in to small fragments “Morsels” stored in NUMA-local storage and build other two hash tables HT(S) and HT(T) based on T and S by using a tool called “QEPobject”

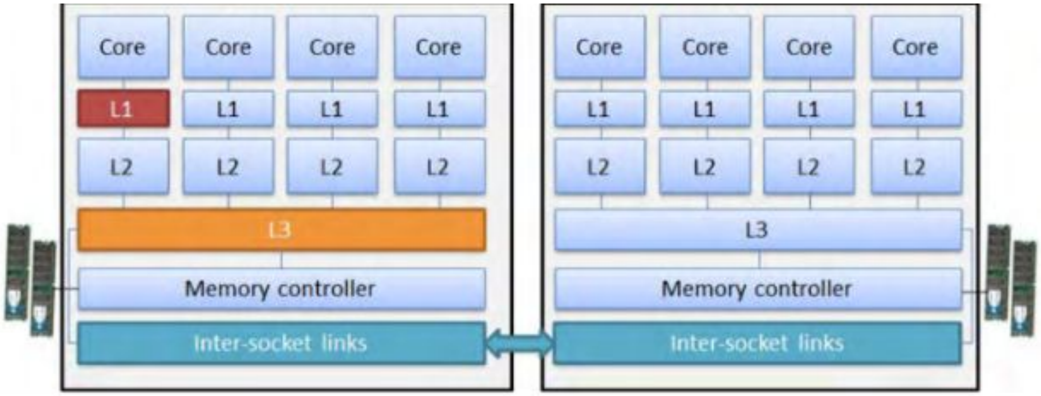
Morsel-Driven Parallelism

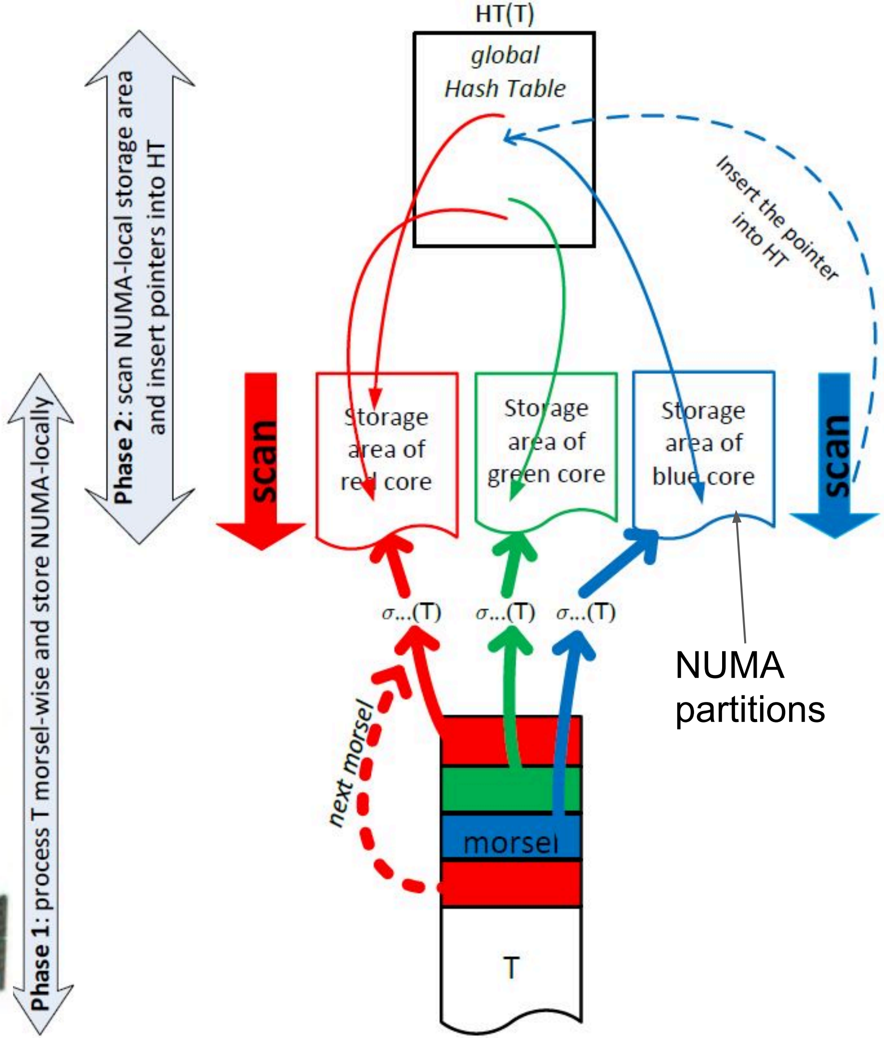
The Three-Way-Join query example in paper

$$\sigma_{\dots}(R) \bowtie_A \sigma_{\dots}(S) \bowtie_B \sigma_{\dots}(T)$$

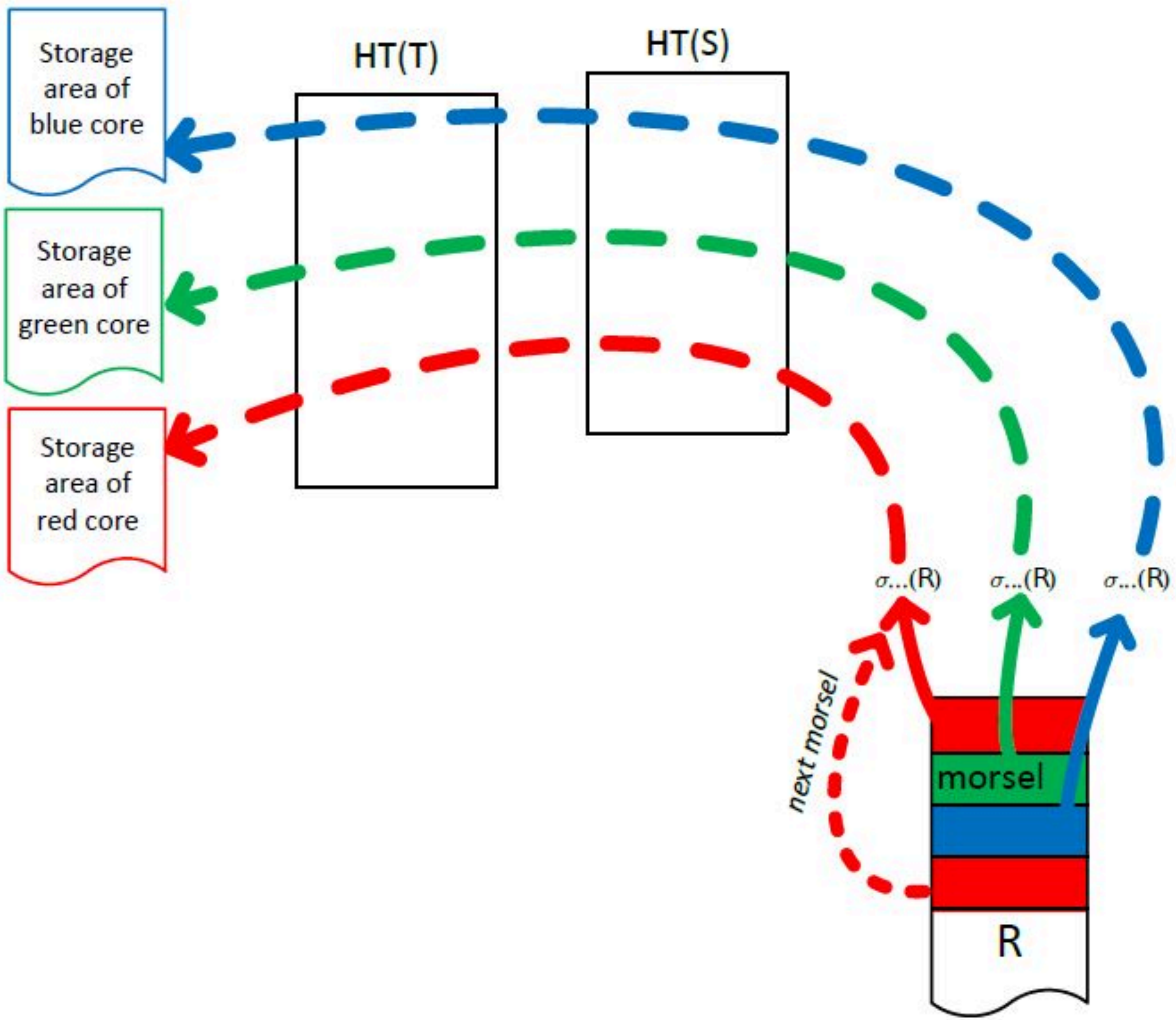


How does QEPobject build hash table H(T) and H(S)?

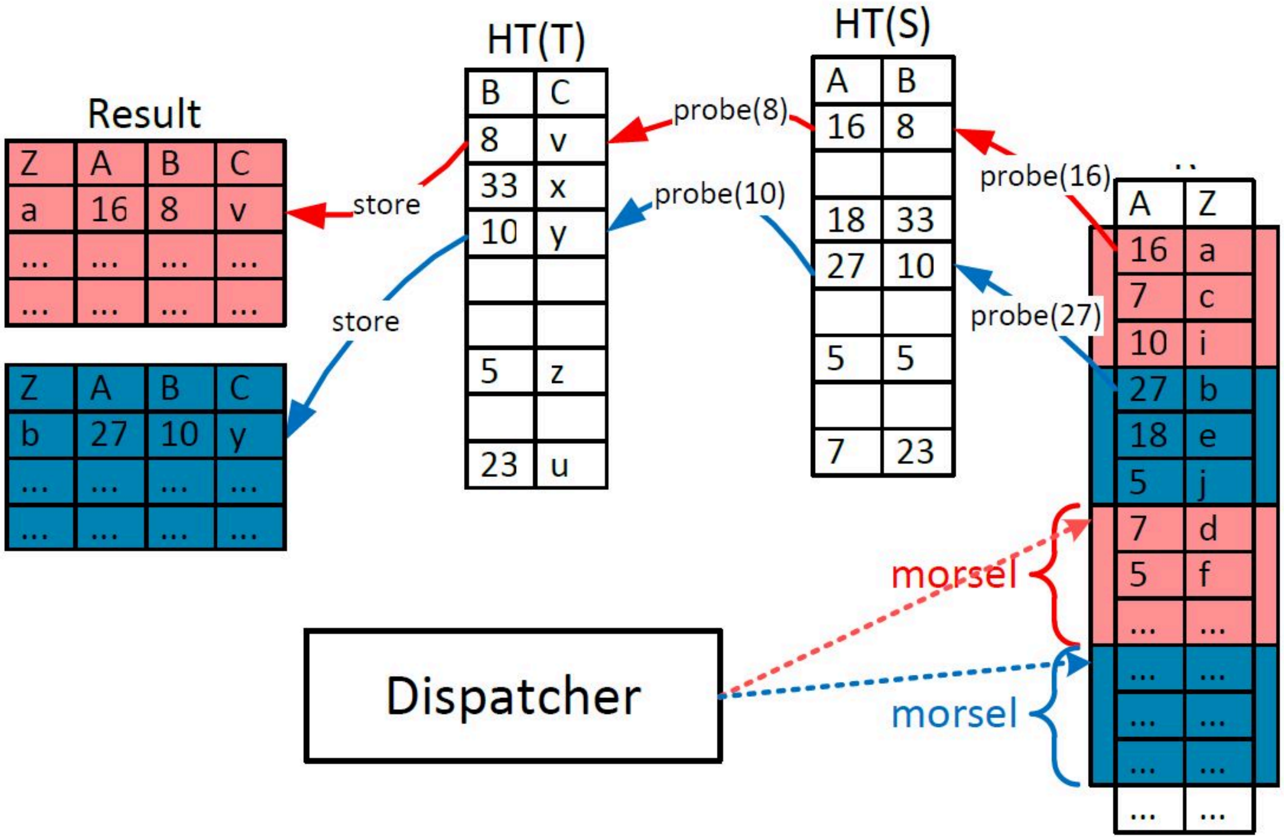




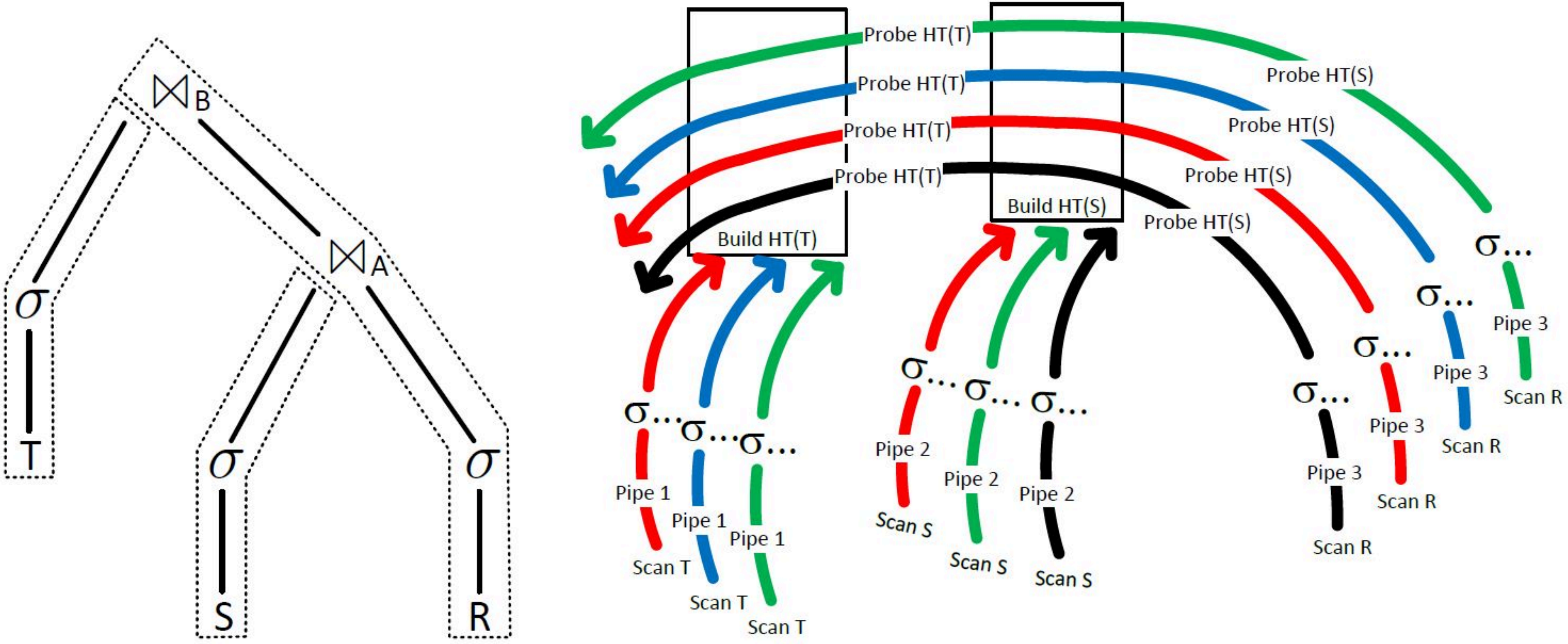
How does probe phase work?



Example of morsel-driven parallelism



Overall Idea of morsel-driven parallelism



How does Dispatcher schedule parallel pipeline tasks?

Goals for Dispatcher

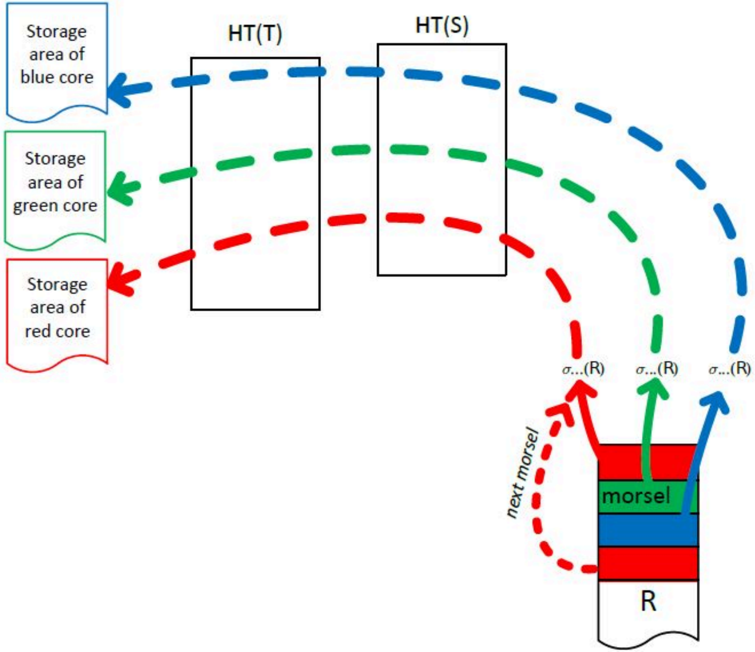
Dispatcher is working to control and assign the compute resources to the parallel pipelines by assigning tasks to different worker threads.

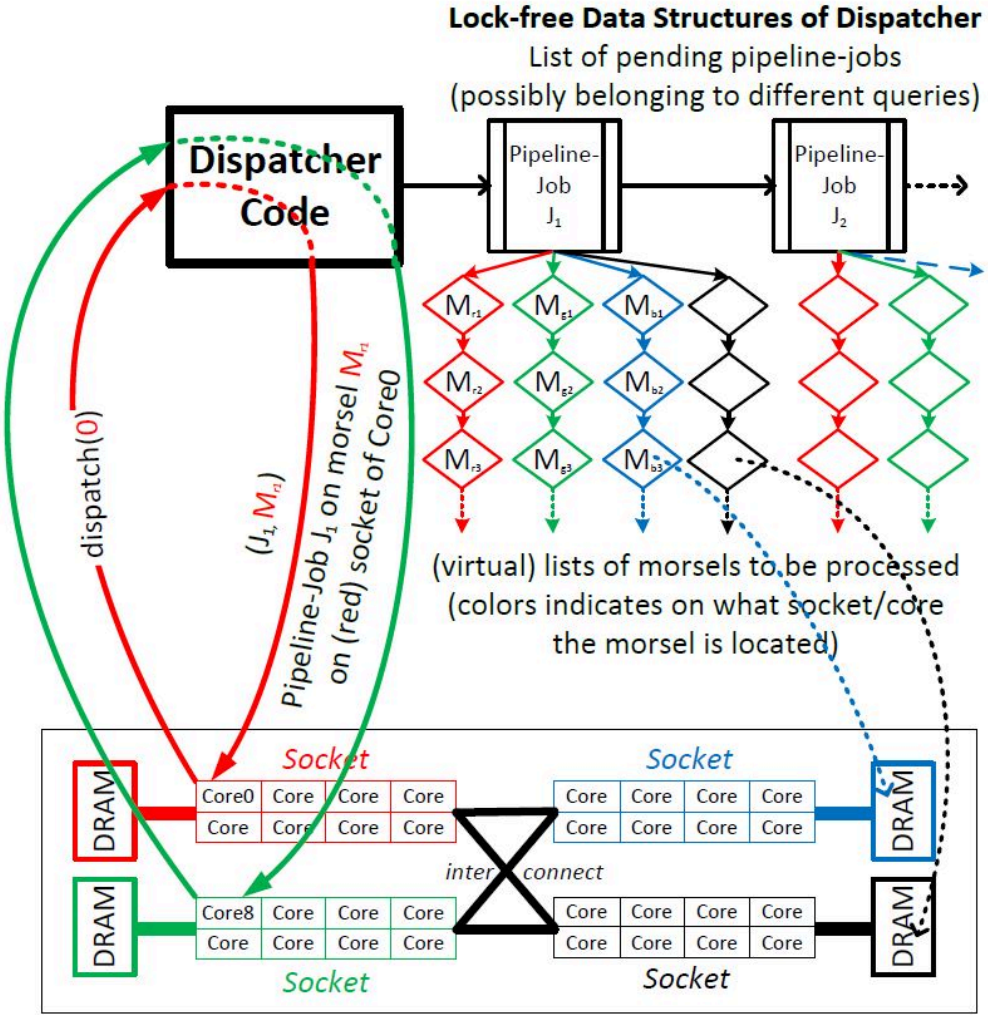
NUMA-Locality	Elasticity	Load balancing

Elasticity of Dispatcher

- Different priority of the tasks will influence the degree of parallelism
- Assume the priority is the same for all tasks in this paper
- Work-stealing mechanism when cores finish in different time

Architecture of the Dispatcher

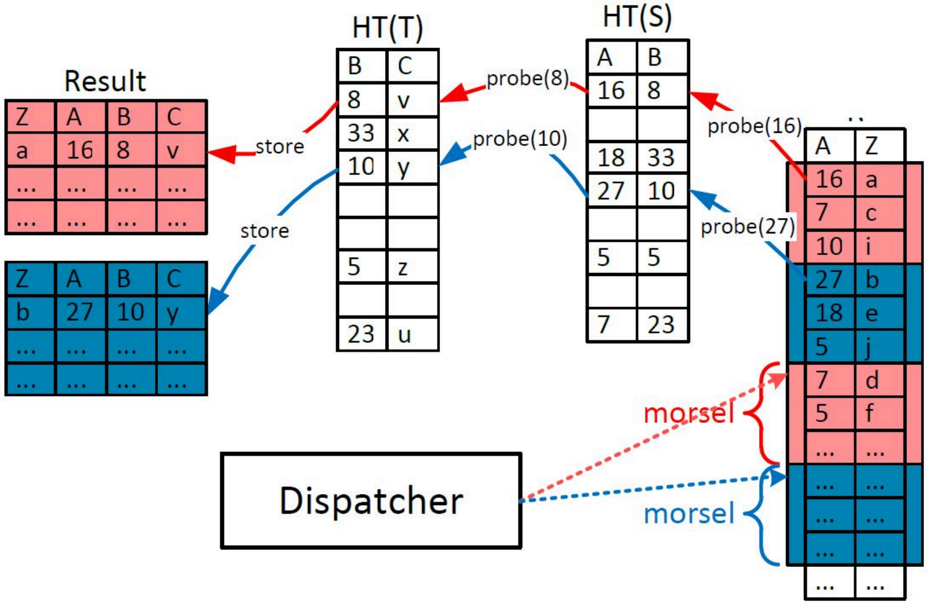




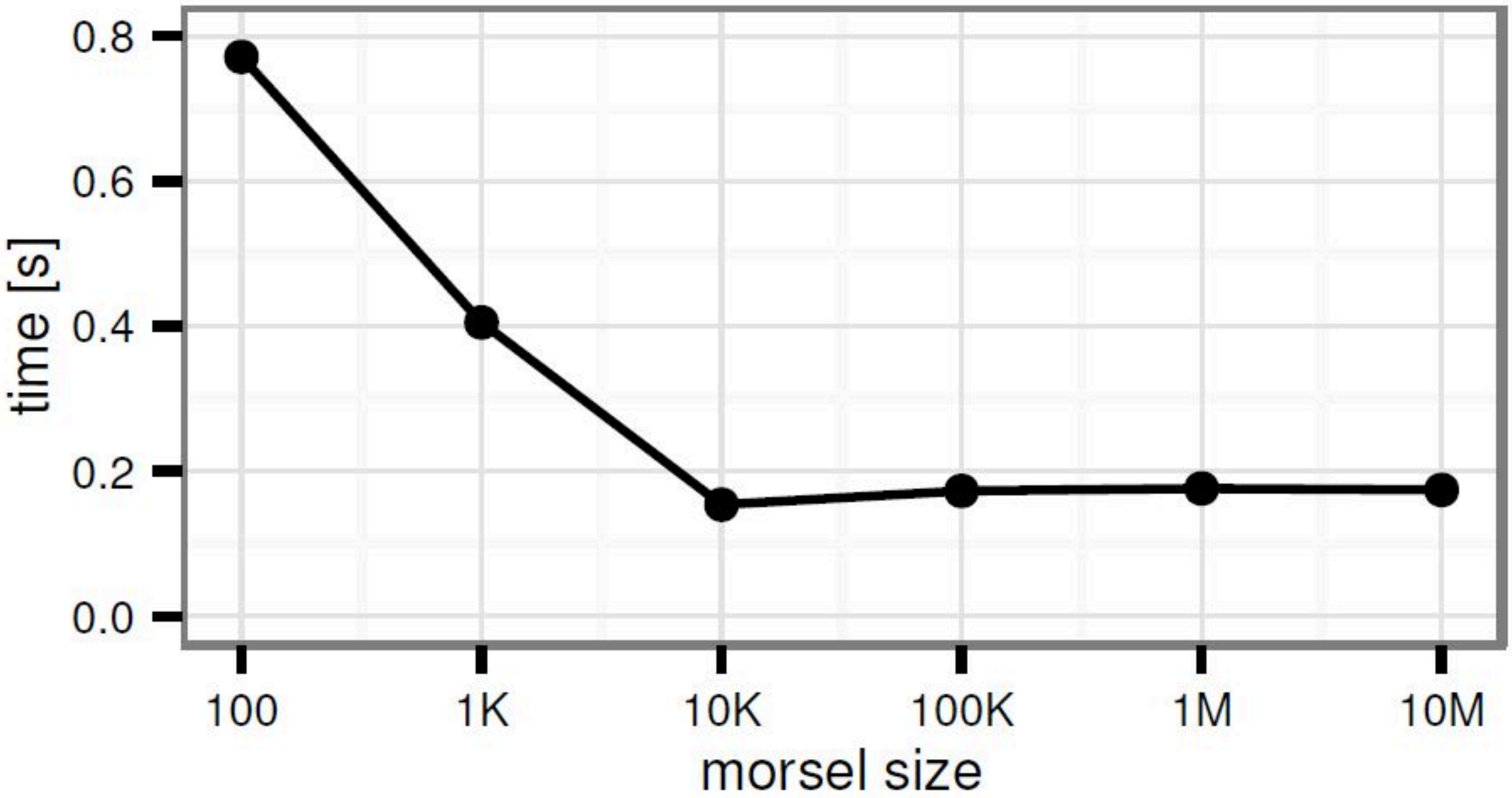
Example NUMA Multi-Core Server with 4 Sockets and 32 Cores

Another question

- What size is the best morsal size to improve this system's performance? Big size morsal or small size morsal?



Morsel size in Dispatcher



Parallel Operator Details

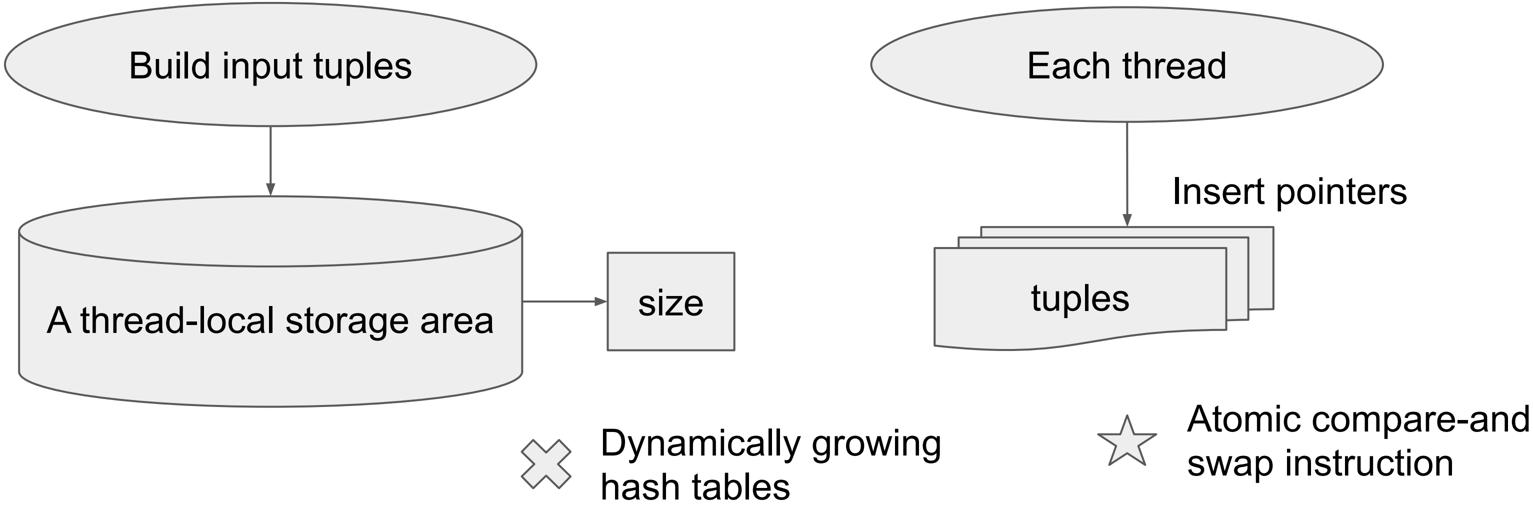
Parallelize each pipeline

Operators that have already started

Operators that start a new pipeline

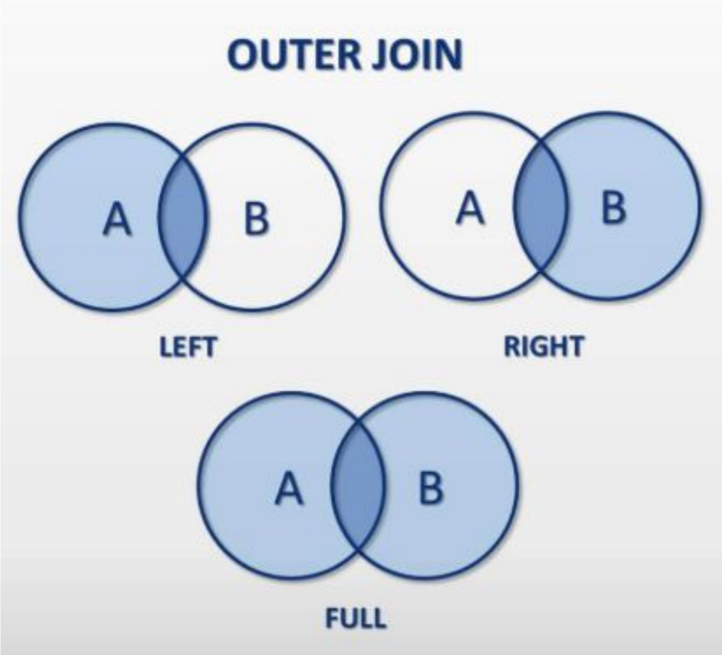
Hash Table Construction

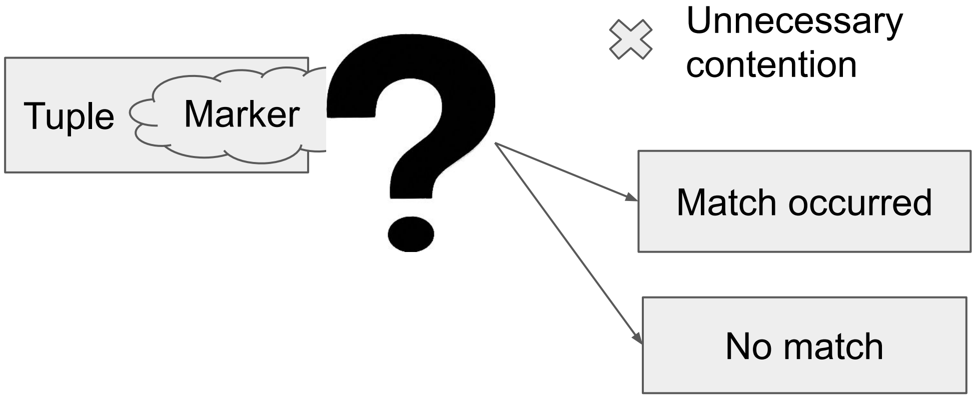
2 phases



Outer Join

Variation of phase 2





Radix Join

http://www.vldb.org/pvldb/vol7/p85-balkesen.pdf

Basic

```
foreach input tuple t do
    k ← hash(t);
    p[k][pos[k]] = t;            // copy t to target partition k
    pos[k]++;
```

optimized

Software-managed buffers

```
foreach input tuple t do
    k ← hash(t);
    buf[k][pos[k] mod N] = t;    // copy t to buffer
    pos[k]++;
    if pos[k] mod N = 0 then
        copy buf[k] to p[k];     // copy buffer to part. k
```

TLB: translation look-aside buffers

Copy overhead ↑

Benefit of single-table hash join

Compared to **Radix Join**

- ✓ Larger input relation
- ✓ Good team player
- ✓ Input & Input
- ✓ Skewed key distributions
- ✓ No care about tuple size
- ✓ No care about hardware

Choose which?

A single-table hash join

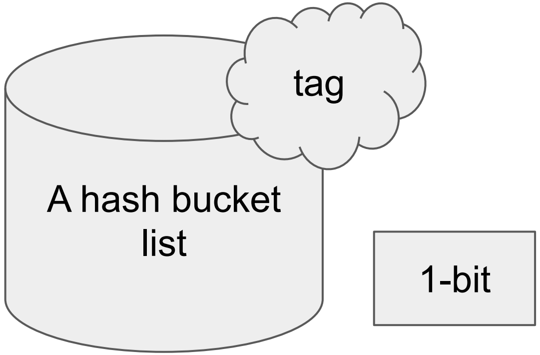
Radix join

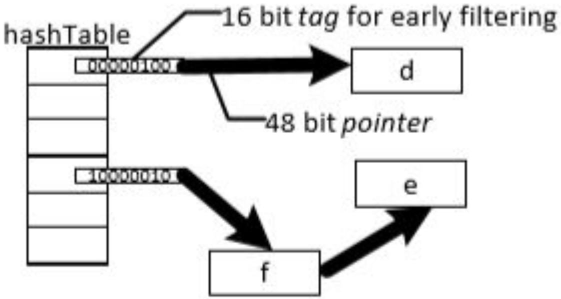
Complex query processing

Higher locality

Lock-Free Tagged Hash Table

Early-filtering optimization



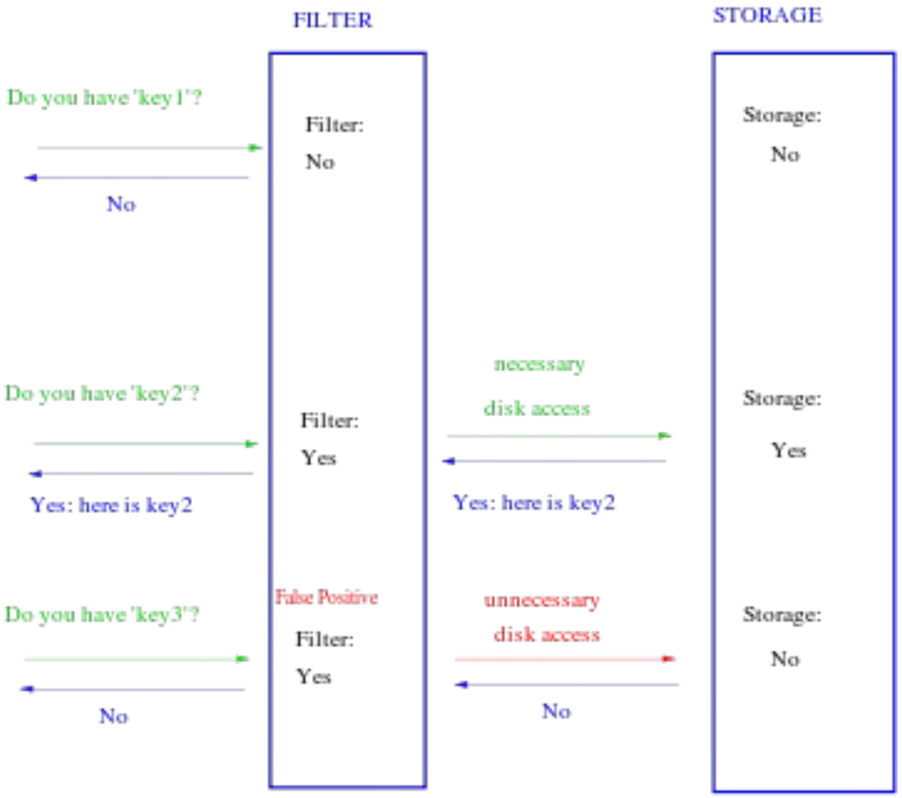



```
insert(entry) {
  // determine slot in hash table
  slot = entry->hash >> hashTableShift
  do {
    old = hashTable[slot]
    // set next to old entry without tag
    entry->next = removeTag(old)
    // add old and new tag
    new = entry | (old&tagMask) | tag(entry->hash)
    // try to set new value,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

Figure 7: Lock-free insertion into tagged hash table

CAS: atomic compare-and-swap operation

Bloom filters



Incur **multiple reads**

May **not fit into cache**

Hash tagging

Low overhead

Proposed hash table

Only store pointers

- 2x size of the input
- Reduce collisions
- Allow for tuples of variable size
- Probe misses fast

Large virtual memory pages

- Reduced number of TLB misses
- Avoid scalability problems

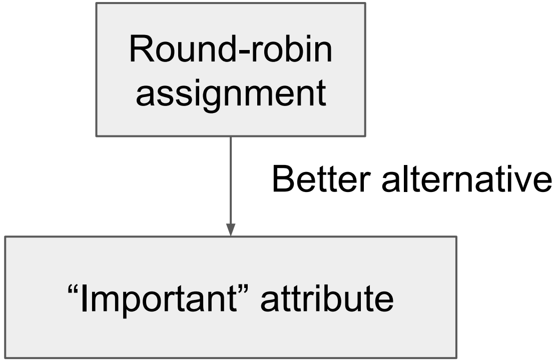
Allocate with Unix mmap

?

In order to implement NUMA-local table scans

Do relations need to be distributed over the memory nodes?

NUMA-Aware Table Partitioning

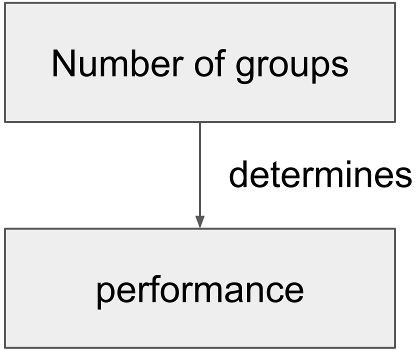


Reside on the **same** socket

Less cross-socket communication

Same hash function

Grouping/Aggregation



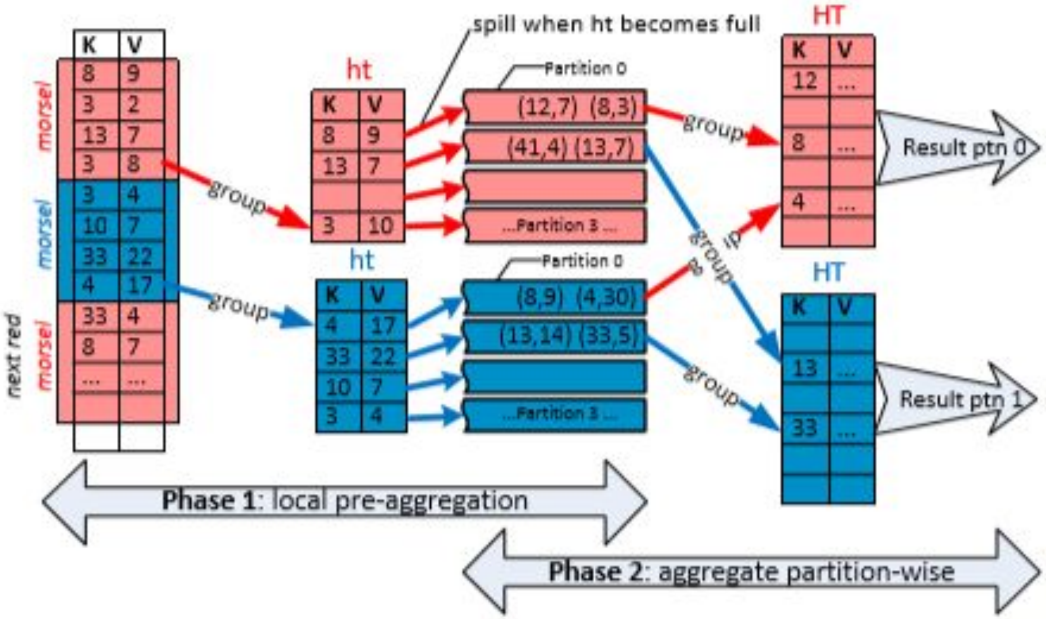


Figure 8: Parallel aggregation

In main memory

Are hash-based algorithms usually faster than sorting?

Sorting

- ✗ Sort-based join or aggregation
- ✓ Order by or top-k clause

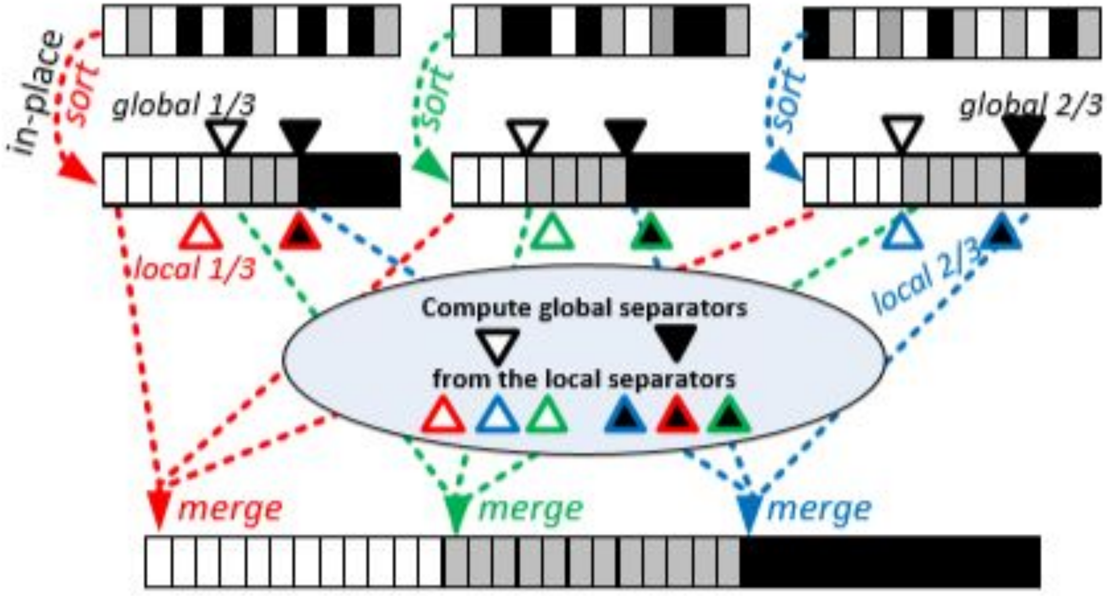


Figure 9: Parallel merge sort

Evaluation

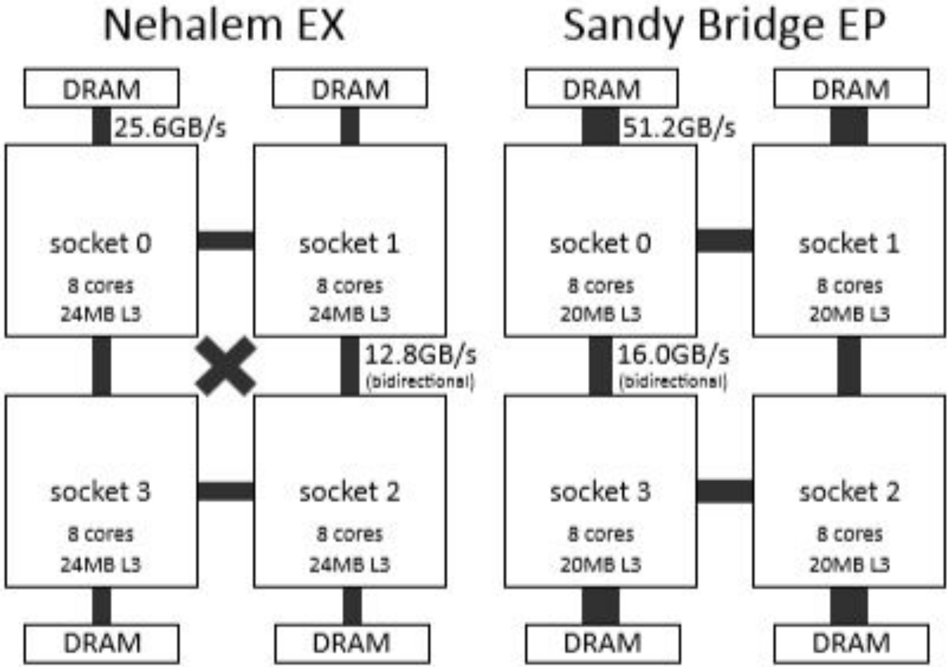


Figure 10: NUMA topologies, theoretical bandwidth

Linux

32 cores

64 threads

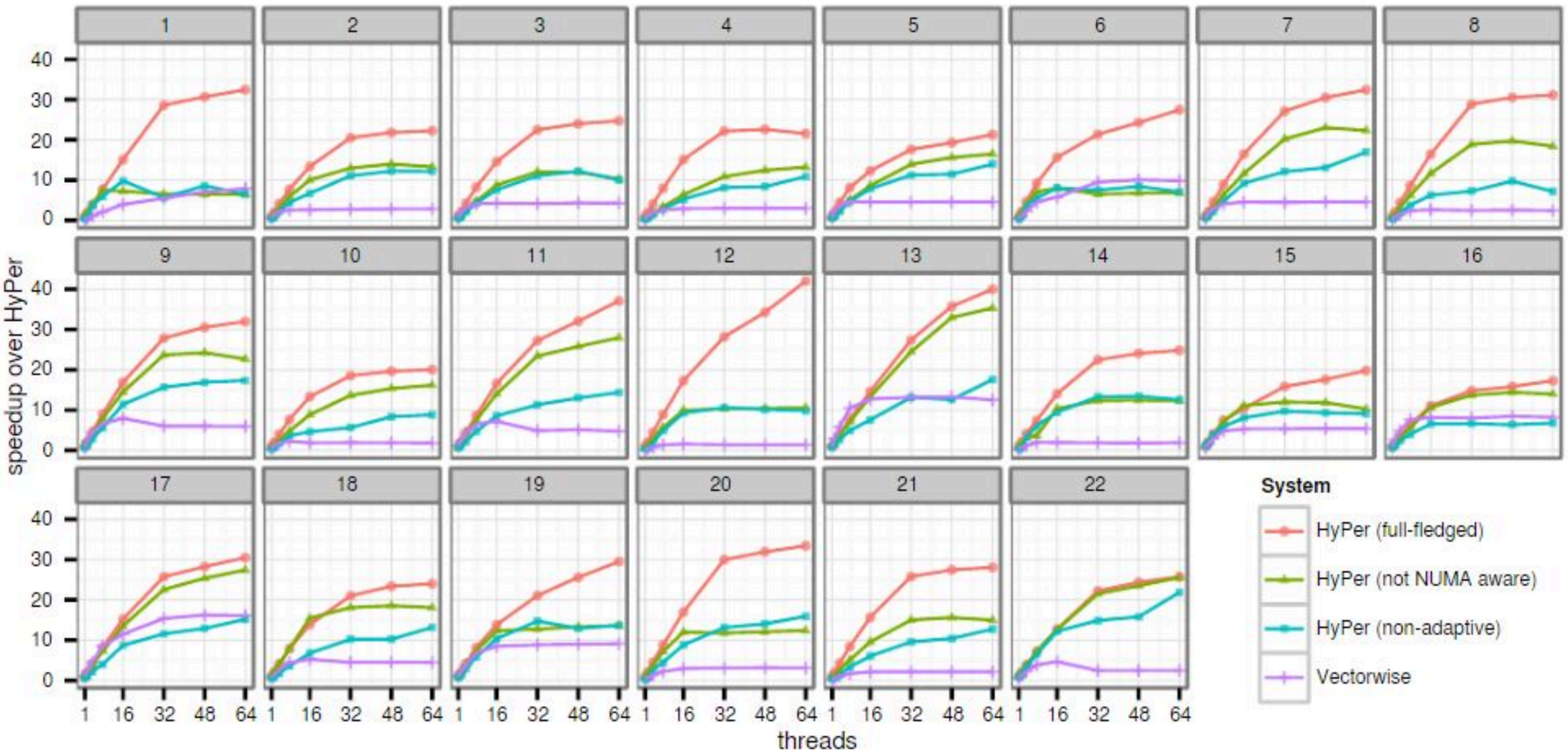


Figure 11: TPC-H scalability on Nehalem EX (cores 1-32 are "real", cores 33-64 are "virtual")

NUMA Awareness

TPC-H #	HyPer time [s]	HyPer scal. [×]	HyPer rd. [GB/s]	HyPer wr. [GB/s]	[%] remote	[%] QPI	Vectorwise time [s]	Vectorwise scal. [×]	Vectorwise rd. [GB/s]	Vectorwise wr. [GB/s]	[%] remote	[%] QPI
1	0.28	32.4	82.6	0.2	1	40	1.13	30.2	12.5	0.5	74	7
2	0.08	22.3	25.1	0.5	15	17	0.63	4.6	8.7	3.6	55	6
3	0.66	24.7	48.1	4.4	25	34	3.83	7.3	13.5	4.6	76	9
4	0.38	21.6	45.8	2.5	15	32	2.73	9.1	17.5	6.5	68	11
5	0.97	21.3	36.8	5.0	29	30	4.52	7.0	27.8	13.1	80	24
6	0.17	27.5	80.0	0.1	4	43	0.48	17.8	21.5	0.5	75	10
7	0.53	32.4	43.2	4.2	39	38	3.75	8.1	19.5	7.9	70	14
8	0.35	31.2	34.9	2.4	15	24	4.46	7.7	10.9	6.7	39	7
9	2.14	32.0	34.3	5.5	48	32	11.42	7.9	18.4	7.7	63	10
10	0.60	20.0	26.7	5.2	37	24	6.46	5.7	12.1	5.7	55	10
11	0.09	37.1	21.8	2.5	25	16	0.67	3.9	6.0	2.1	57	3
12	0.22	42.0	64.5	1.7	5	34	6.65	6.9	12.3	4.7	61	9
13	1.95	40.0	21.8	10.3	54	25	6.23	11.4	46.6	13.3	74	37
14	0.19	24.8	43.0	6.6	29	34	2.42	7.3	13.7	4.7	60	8
15	0.44	19.8	23.5	3.5	34	21	1.63	7.2	16.8	6.0	62	10
16	0.78	17.3	14.3	2.7	62	16	1.64	8.8	24.9	8.4	53	12
17	0.44	30.5	19.1	0.5	13	13	0.84	15.0	16.2	2.9	69	7
18	2.78	24.0	24.5	12.5	40	25	14.94	6.5	26.3	8.7	66	13
19	0.88	29.5	42.5	3.9	17	27	2.87	8.8	7.4	1.4	79	5
20	0.18	33.4	45.1	0.9	5	23	1.94	9.2	12.6	1.2	74	6
21	0.91	28.0	40.7	4.1	16	29	12.00	9.1	18.2	6.1	67	9
22	0.30	25.7	35.5	1.3	75	38	3.14	4.3	7.0	2.4	66	4

Table 1: TPC-H (scale factor 100) statistics on Nehalem EX

Vectorwise

Not NUMA optimized

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
time [s]	0.21	0.10	0.63	0.30	0.84	0.14	0.56	0.29	2.44	0.61	0.10	0.33	2.32	0.33	0.33	0.81	0.40	1.66	0.68	0.18	0.74	0.47
scal. [×]	39.4	17.8	18.6	26.9	28.0	42.8	25.3	33.3	21.5	21.0	27.4	41.8	16.5	15.6	20.5	11.0	34.0	29.1	29.6	33.7	26.4	8.4

Table 2: TPC-H (scale factor 100) performance on Sandy Bridge EP

ses:

	bandwidth [GB/s]		latency [ns]	
	local	mix	local	mix
Nehalem EX	93	60	161	186
Sandy Bridge EP	121	41	101	257

NUMA-awareness

Elasticity

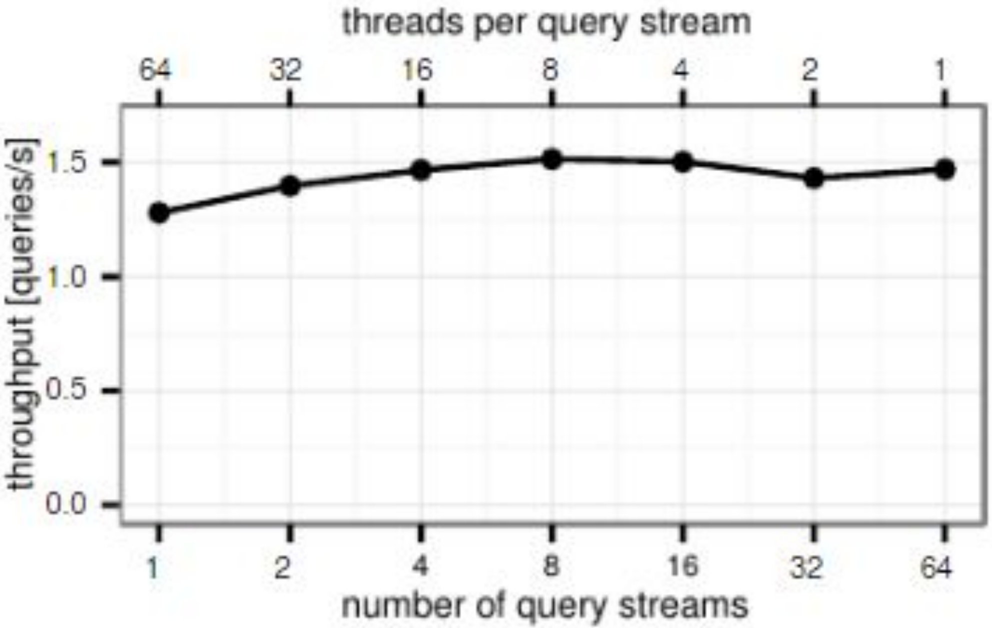


Figure 12: Intra- vs. inter-query parallelism with 64 threads

Elasticity cont.

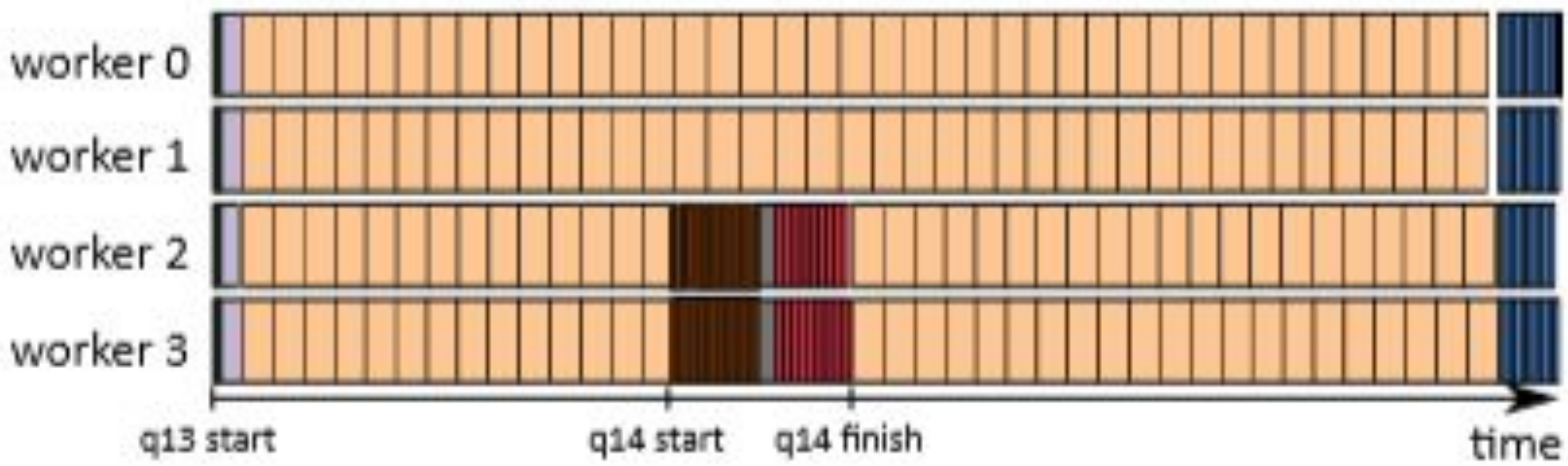


Figure 13: Illustration of morsel-wise processing and elasticity

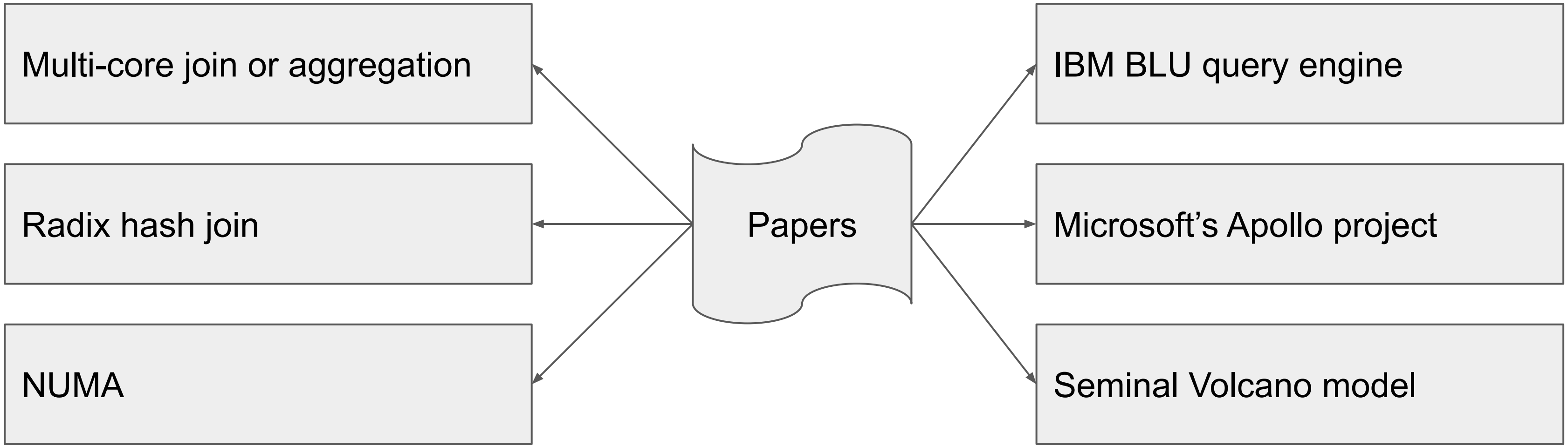
elastic

Star Schema Benchmark

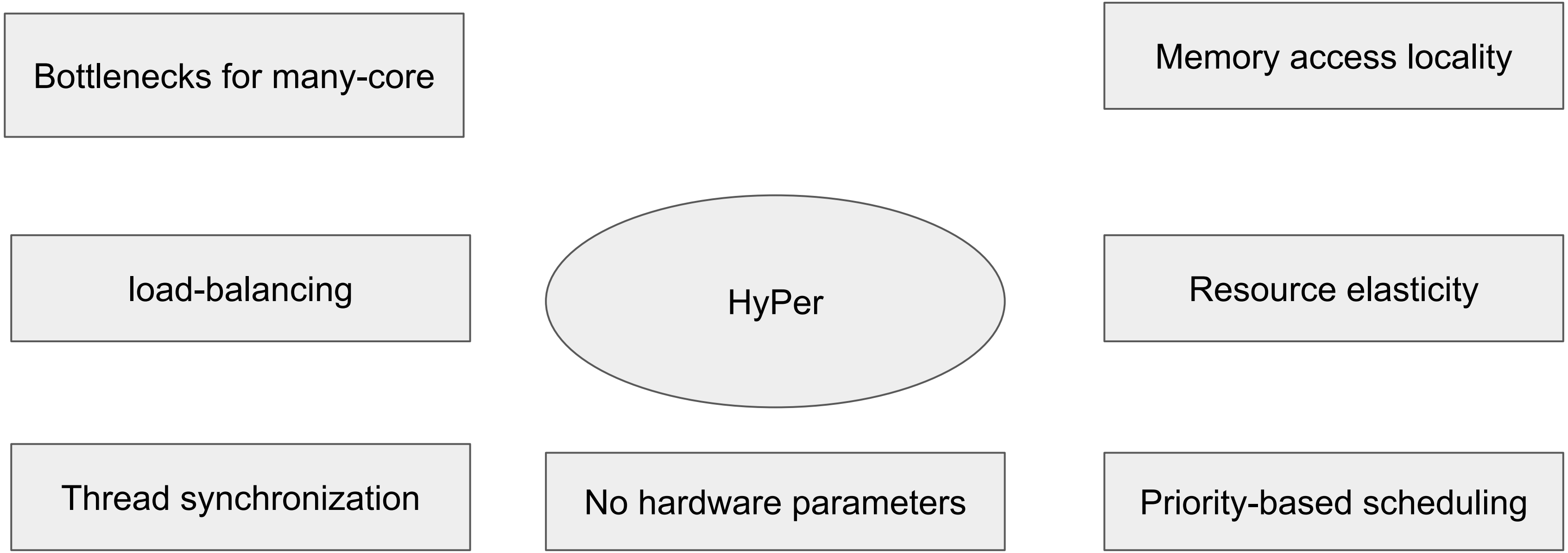
SSB #	time [s]	scal. [×]	read [GB/s]	write [GB/s]	remote [%]	QPI [%]
1.1	0.10	33.0	35.8	0.4	18	29
1.2	0.04	41.7	85.6	0.1	1	44
1.3	0.04	42.6	85.6	0.1	1	44
2.1	0.11	44.2	25.6	0.7	13	17
2.2	0.15	45.1	37.2	0.1	2	19
2.3	0.06	36.3	43.8	0.1	3	25
3.1	0.29	30.7	24.8	1.0	37	21
3.2	0.09	38.3	37.3	0.4	7	22
3.3	0.06	40.7	51.0	0.1	2	27
3.4	0.06	40.5	51.9	0.1	2	28
4.1	0.26	36.5	43.4	0.3	34	34
4.2	0.23	35.1	43.3	0.3	28	33
4.3	0.12	44.2	39.1	0.3	5	22

Table 3: Star Schema Benchmark (scale 50) on Nehalem EX

Related Work



Conclusion


Bottlenecks for many-core
Memory access locality
load-balancing
HyPer
Resource elasticity
Thread synchronization
No hardware parameters
Priority-based scheduling

Future Work

Underlying hardware

Further optimizations

Remote NUMA access