MADALGO

CENTER FOR MASSIVE DATA ALGORITHMICS

Lecture on Multicores

Darius Sidlauskas
Post-doc

Danmarks Grundforskningsfond
Danish National Research Foundation

Outline

- Part 1
  - Background
  - Current multicore CPUs
- Part 2
  - To share or not to share
- Part 3
  - Demo
  - War story

Outline

- **Part 1**
  - **Background**
  - **Current multicore CPUs**
- Part 2
  - To share or not to share
- Part 3
  - Demo
  - War story

Software crisis

"The major cause of the software crisis is that the machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem."

-- E. Dijkstra, 1972 Turing Award Lecture

Before..

- The 1st Software Crisis
  - When: around '60 and 70'
  - Problem: large programs written in assembly
  - Solution: abstraction and portability via high-level languages like C and FORTRAN
- The 2nd Software Crisis
  - When: around '80 and '90
  - Problem: building and maintaining large programs written by hundreds of programmers
  - Solution: software as a process (OOP, testing, code reviews, design patterns)
    - Also better tools: IDEs, version control, component libraries, etc.

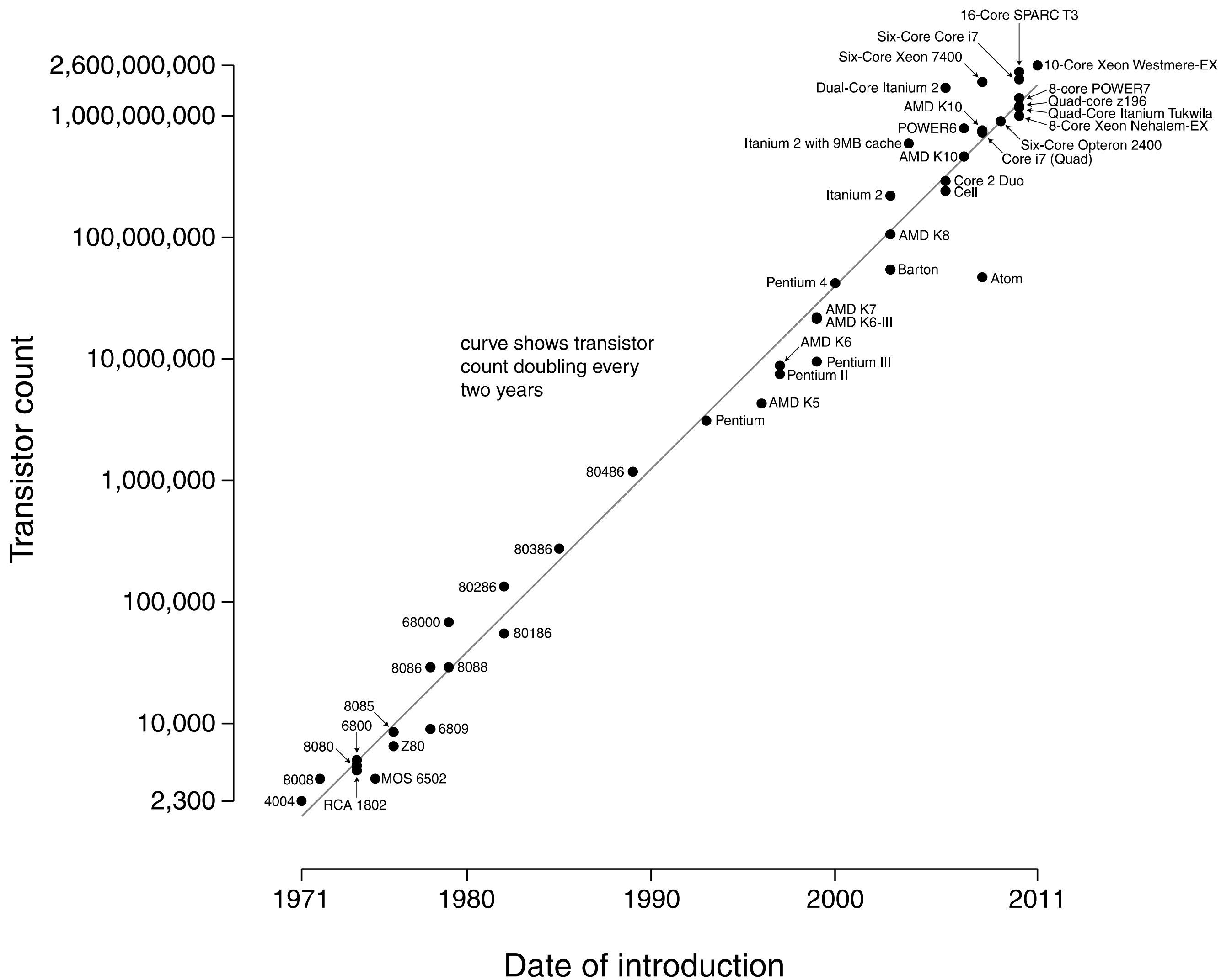
Recently..

- Processor-oblivious programmers
  - A Java program written on PC works on your phone
  - A C program written in '70 still works today and is faster
  - Moore’s law takes care of good speedups

Currently..

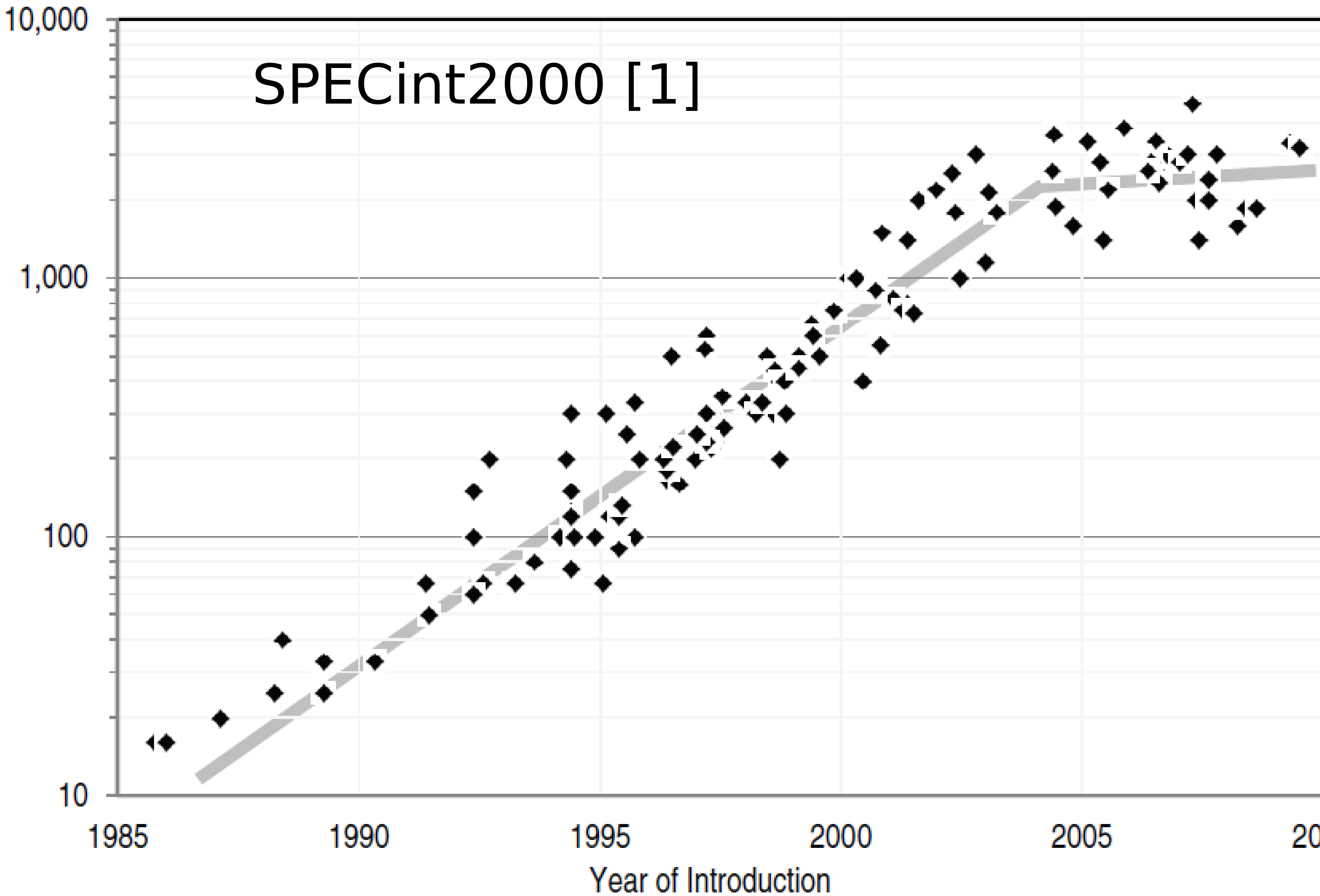
- Software crisis again?
  - When: 2005 and ...
  - Problem: sequential performance is stuck
  - Required solution: continuous and reasonable performance improvements
    - To process large datasets (BIG Data!)
    - To support new features
    - Without loosing portability and maintainability

Moore's law



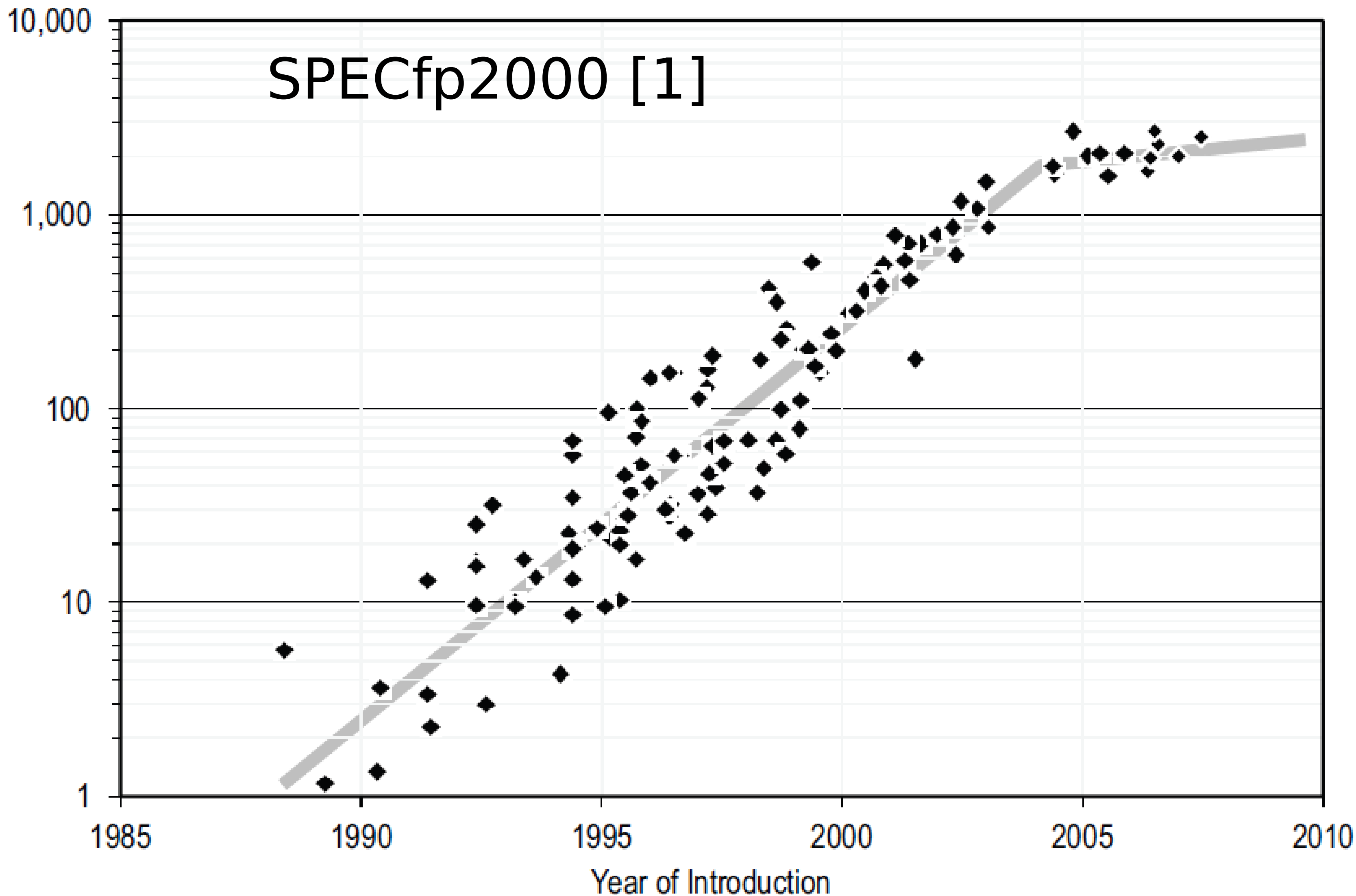
Danmarks Grundforskningsfond
Danish National Research Foundation

Uniprocessor performance

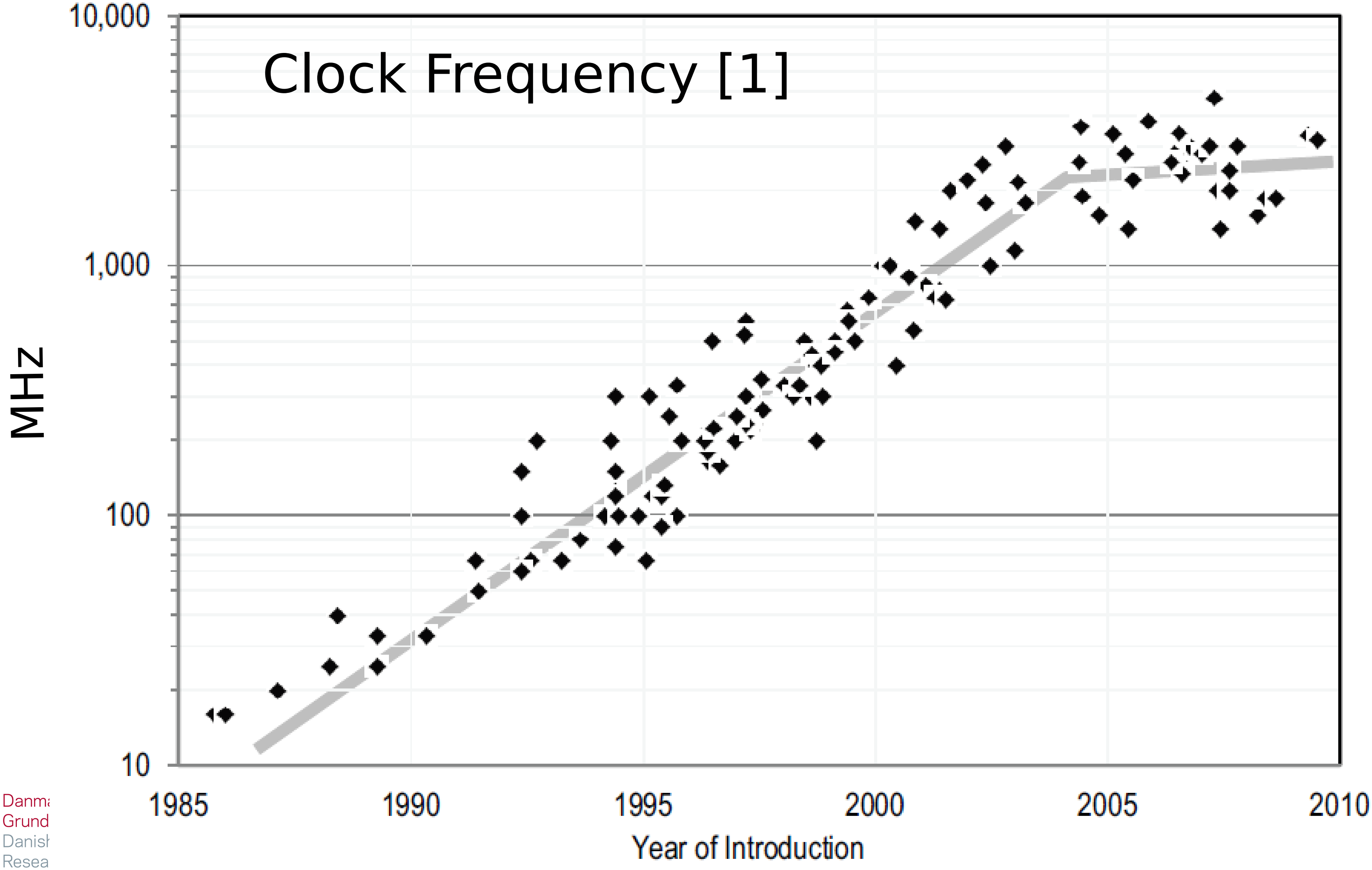


Danma
Grundf
Danish
Resear

Uniprocessor performance (cont.)



Uniprocessor performance (cont.)



Why

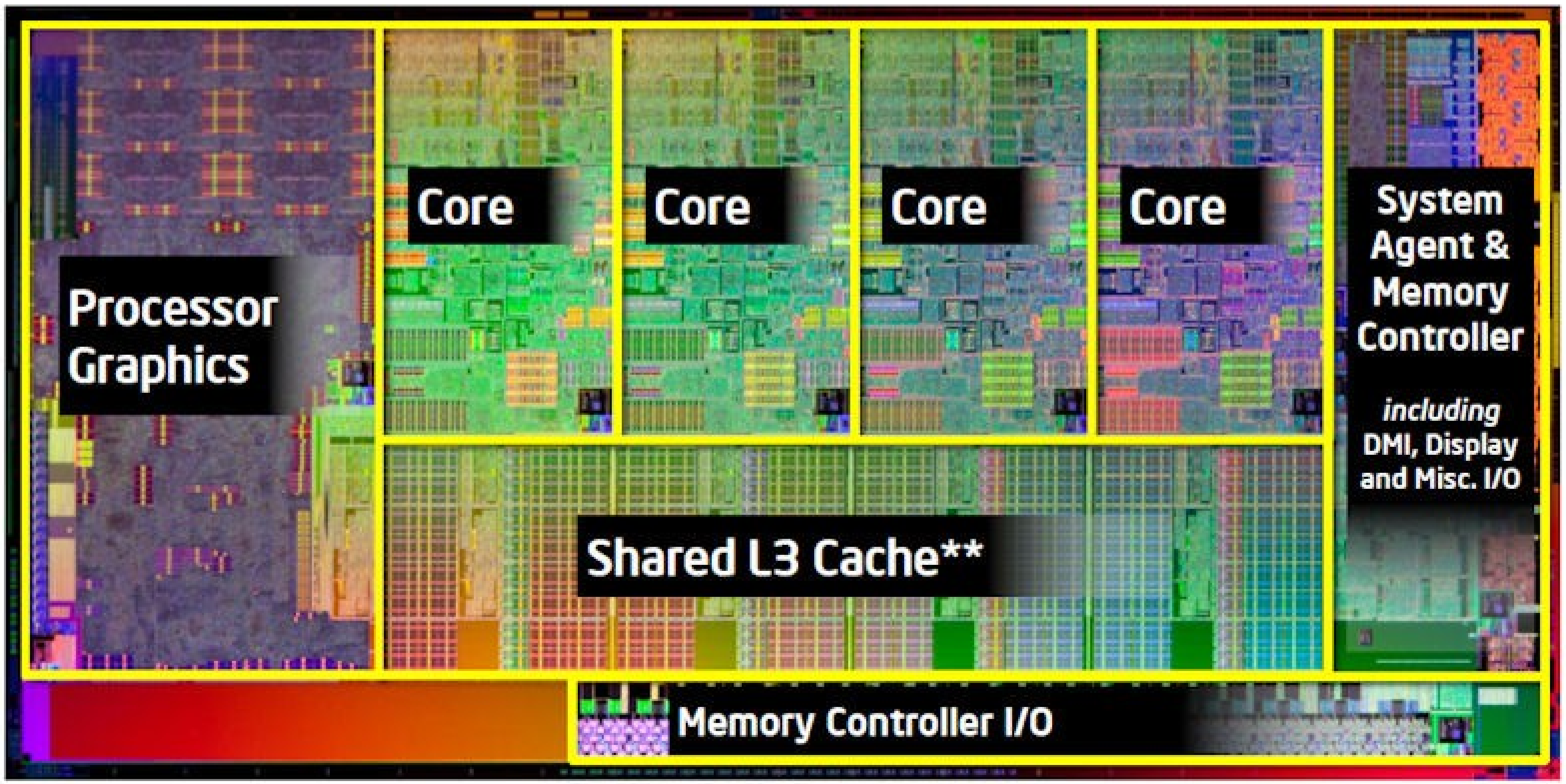
- Power considerations
  - Consumption
  - Cooling
  - Efficiency
- DRAM access latency
  - Memory wall
- Wire delays
  - Range of wire in one clock cycle
- Diminishing returns of more instruction-level parallelism
  - Out-of-order execution, branch prediction, etc.

Overclocking [2]

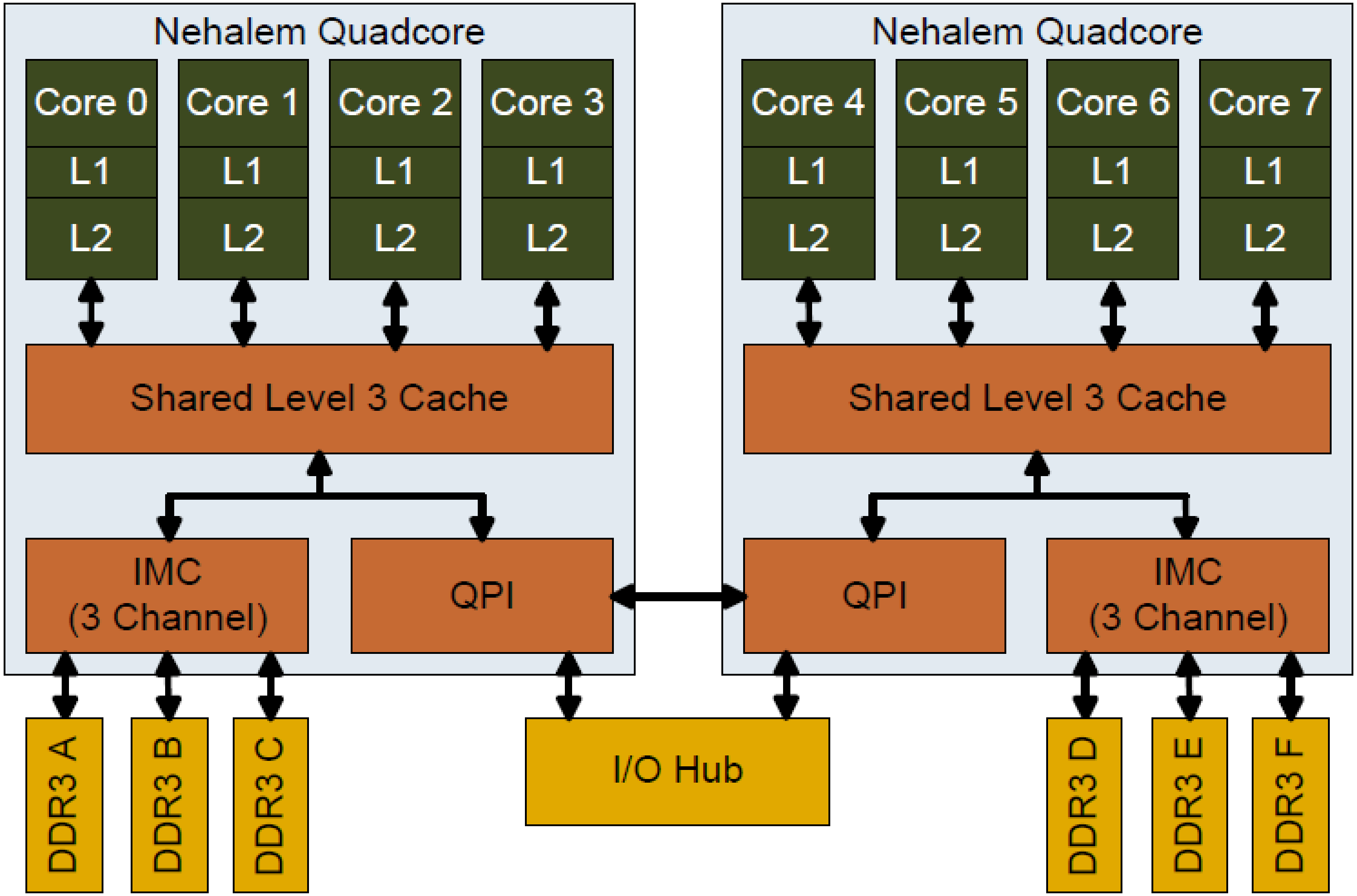
- Air-water: ~5.0 GHz
  - Possible at home
- Phase change: ~6.0 GHz
- Liquid helium: 8.794 GHz
  - Current world record
  - Reached with AMD FX-8350

Shift to multicores

- Instead of going faster --> go more parallel!
  - Transistors are used now for multiple cores

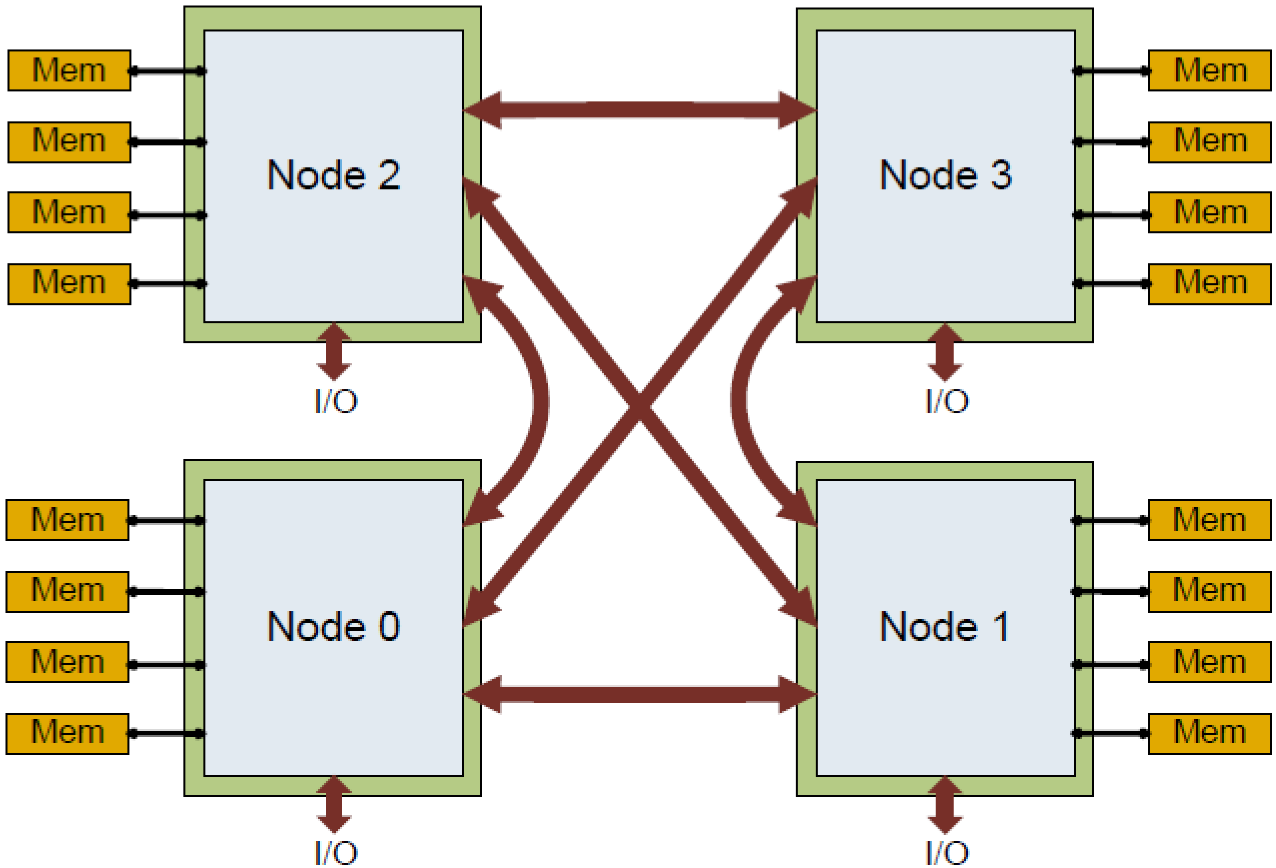


Multi-socket configuration

- 

Four-socket configuration

▪



Current commercial multicore CPUs

- Intel
  - i7-4960X: 6-core (12 threads), 15 MB Cache, max 4.0 GHz
  - Xeon E7-8890 v2: 15-core (30 threads), 37.5 MB Cache, max 3.4 GHz (x 8-socket configuration)
  - Phi 7120P: 61 cores (244 threads), 30.5 MB Cache, max 1.33 GHz, max memory BW 352 GB/s
- AMD
  - FX-9590: 8-core, 8 MB Cache, 4.7 GHz
  - A10-7850K: 12-core (4 CPU 4 GHz + 8 GPU 0.72 GHz), 4 MB C
  - Opteron 6386 SE: 16-core, 16 MB Cache, 3.5 GHz (x 4-socket conf.)
- Oracle
  - SPARC M6: 12-core (96 threads), 48 MB Cache, 3.6 GHz (x 32-socket configuration)

Concurrency vs. Parallelism

- Parallelism
  - A condition that arises when at least two threads are executing simultaneously
  - A specific case of concurrency
- Concurrency:
  - A condition that exists when at least two threads are making progress.
  - A more generalized form of parallelism
  - E.g., concurrent execution via time-slicing in uniprocessors (virtual parallelism)
- Distribution:
  - As above but running simultaneously on different machines (e.g., cloud computing)

Amdhal's law

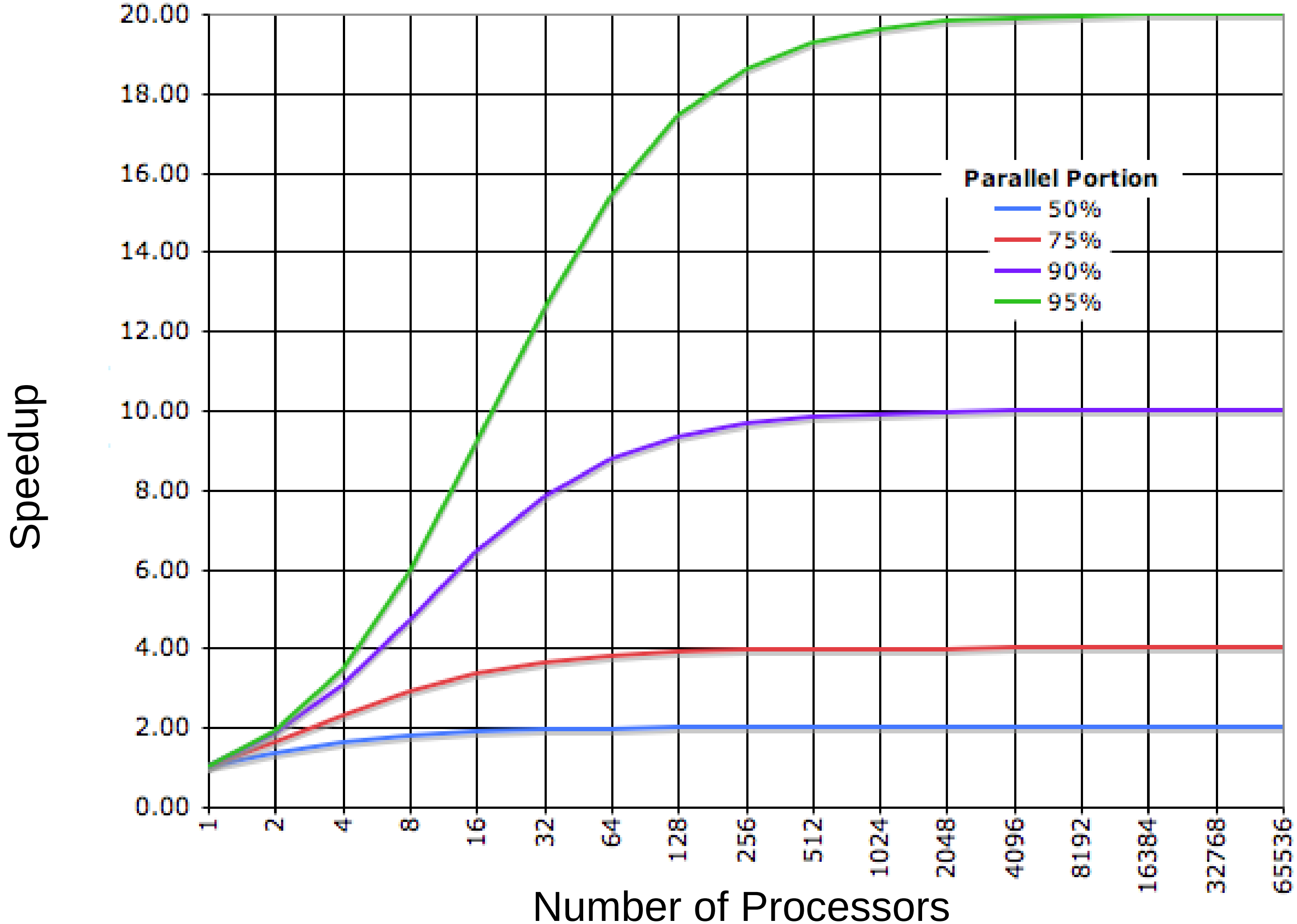
- Potential program speedup is defined by the fraction of code that can be parallelized
- Serial components rapidly become performance limiters as thread count increases
  - *p* – fraction of work that can parallelized
  - *n* – the number of processors

$$\text{Speedup} = \frac{1}{(1-p) + \frac{p}{n}}$$

fraction of time to complete sequential work

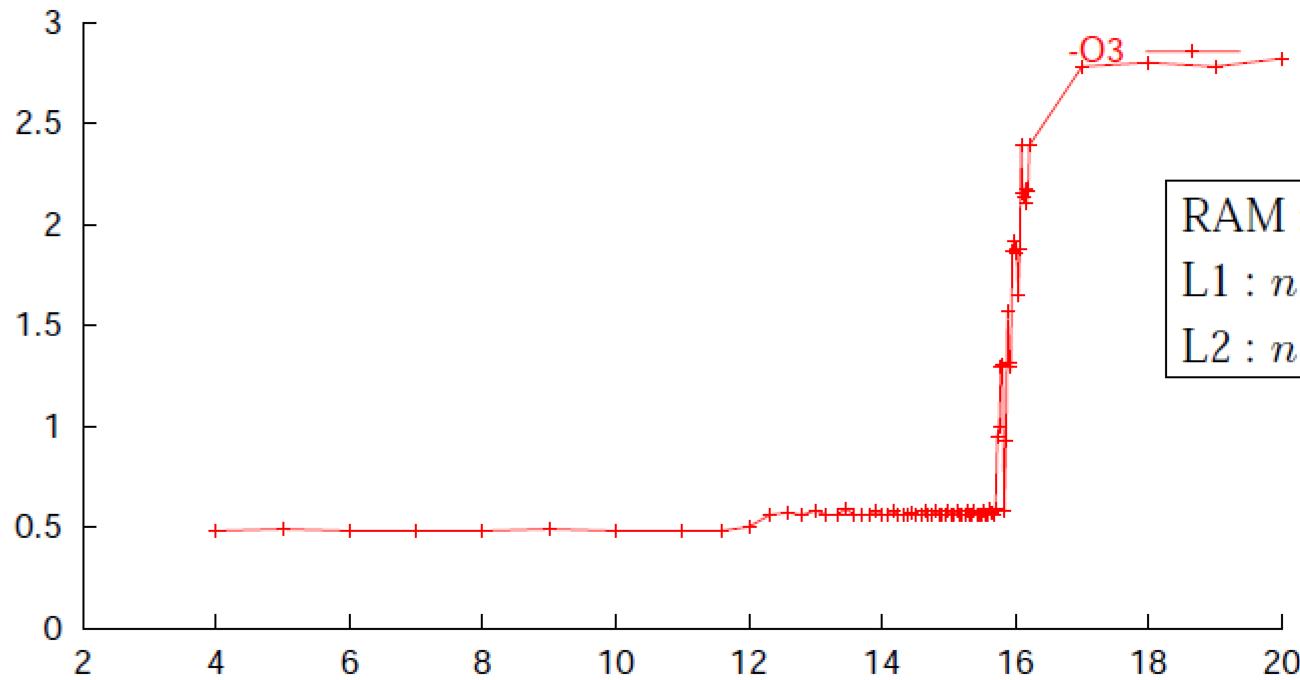
fraction of time to complete parallel work

Amdhal's law


20.00
18.00
16.00
14.00
12.00
10.00
8.00
6.00
4.00
2.00
0.00
Speedup
Parallel Portion
50%
75%
90%
95%
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
65536
Number of Processors

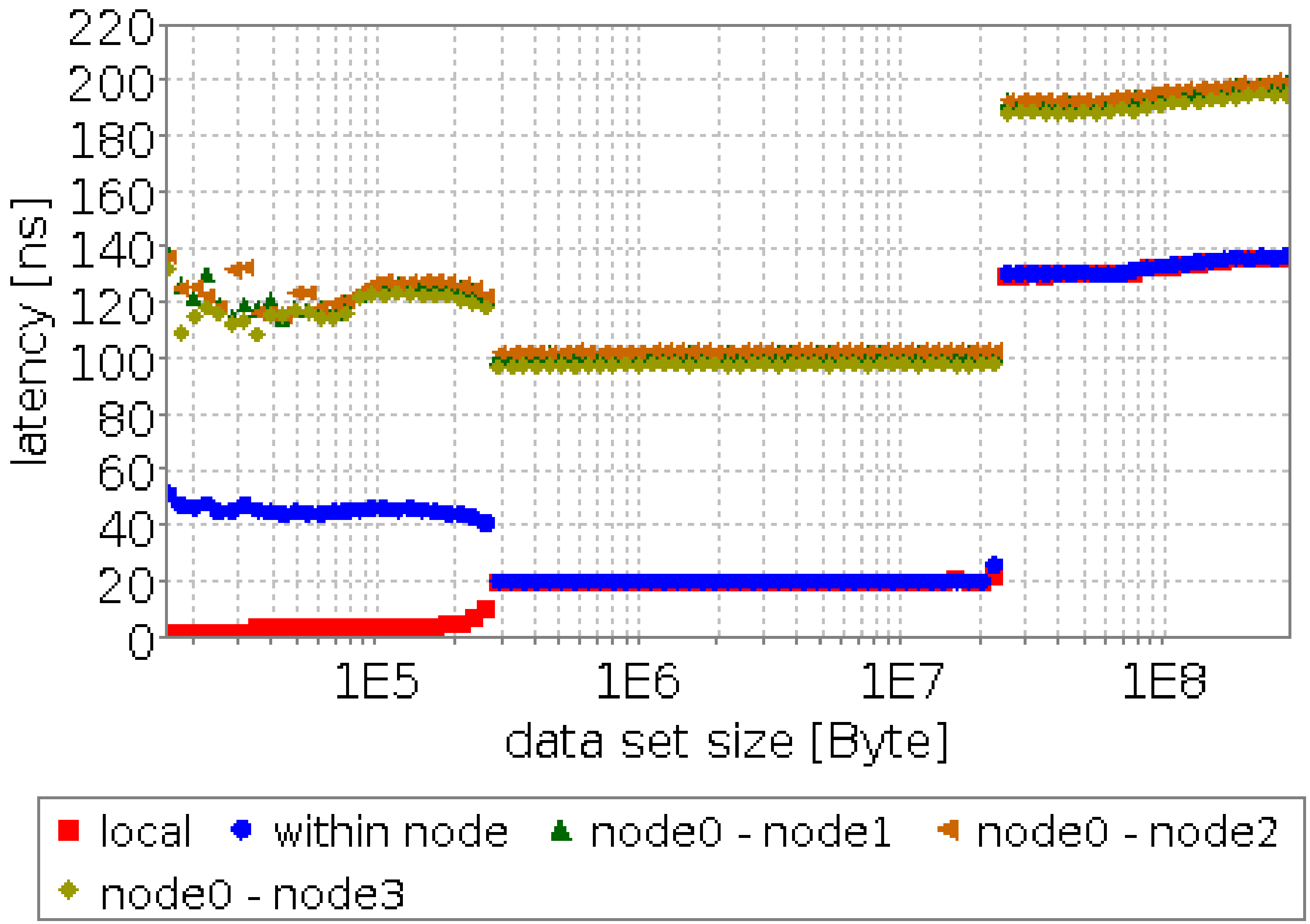
You've seen this..

- L1 and L2 Cache Sizes



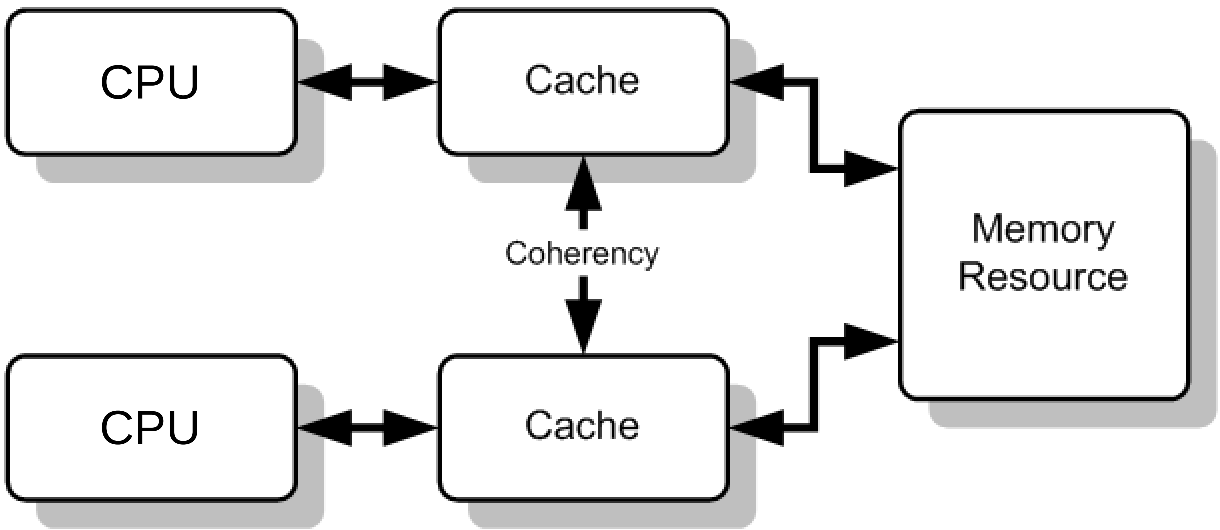
$$\begin{aligned} \text{RAM} &: n \approx 2^{25} \equiv 128\,MB \\ \text{L1} &: n \approx 2^{12} \equiv 16\,KB \\ \text{L2} &: n \approx 2^{16} \equiv 256\,KB \end{aligned}$$

NUMA effects [3]



Cache coherence

- Ensures the consistency between all the caches.

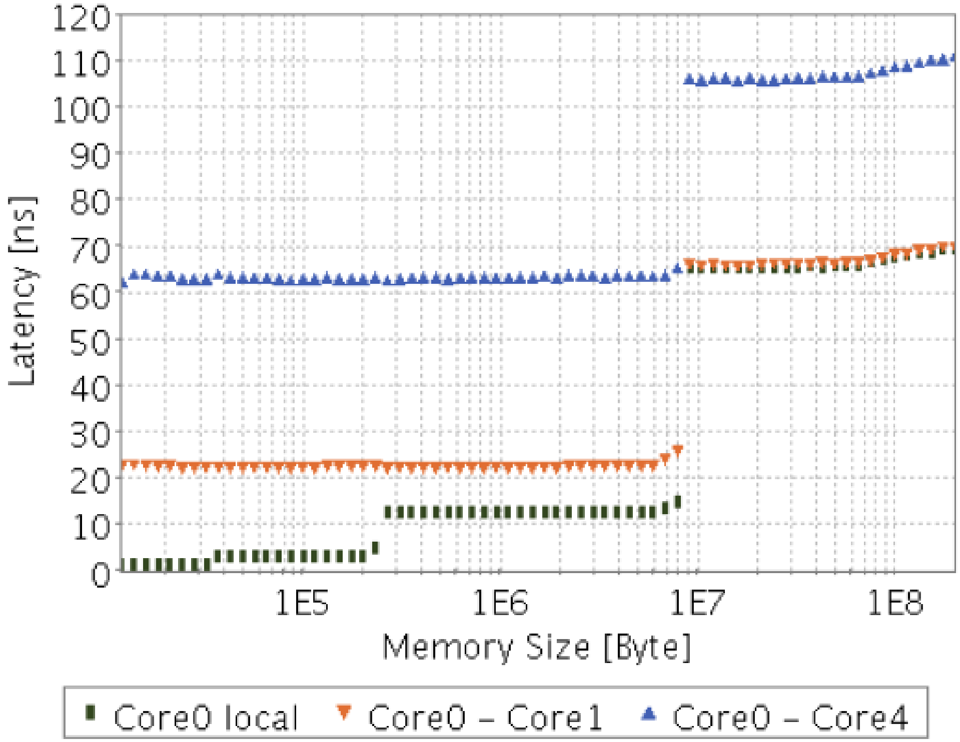


MESIF protocol

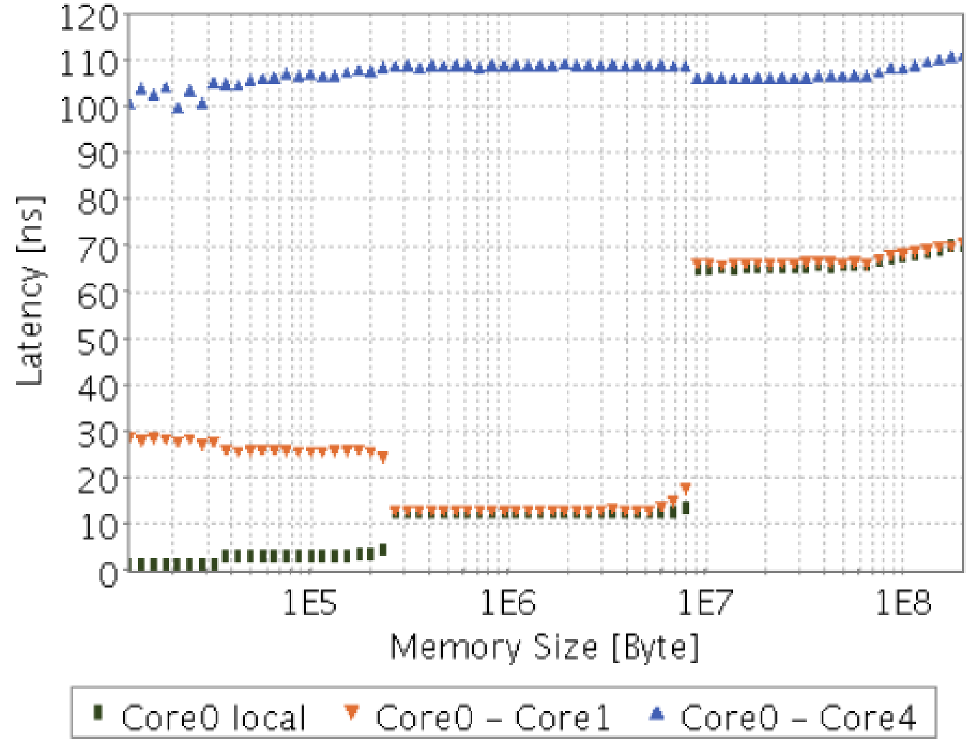
- Modified (M): present only in the current cache and *dirty*. A write-back to main memory will make it (E).
- Exclusive (E): present only in the current cache and *clean*. A read request will make it (S), a write-request will make it (M).
- Shared (S): maybe stored in other caches and *clean*. Maybe changed to (I) at any time.
- Invalid (I): unusable
- Forward (F): a specialized form of the S state

Cache coherency effects [4]

Exclusive cache lines



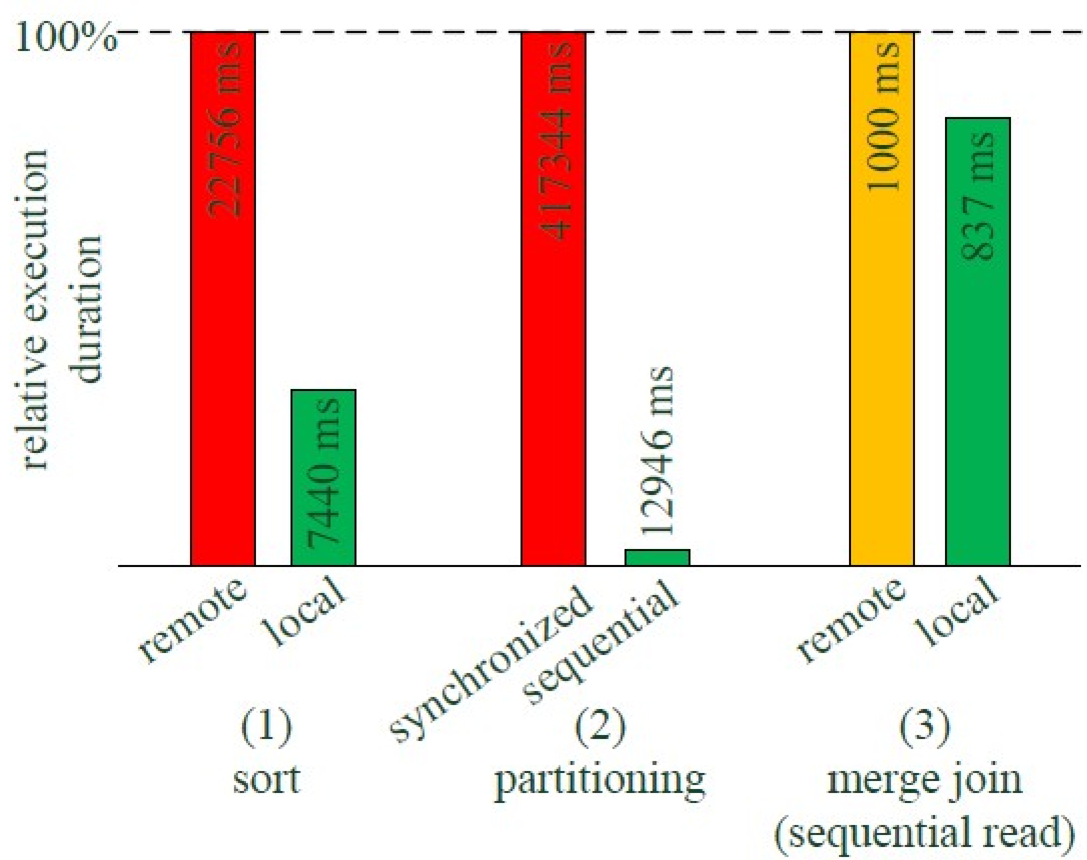
Modified cache lines



Latency in nsec on 2-socket Intel Nehalem [3]

Does it have effect in practice?

- Processing 1600M tuples on 32-core machine [5]



Commandments [5]

- C1: *Thou shalt not write thy neighbor's memory randomly* – chunk the data, redistribute, and then sort/work on your data locally.
- 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.

Outline

- Part 1
  - Background
  - Current multicore CPUs
- **Part 2**
  - **To share or not to share?**
- Part 3
  - Demo
  - War story

Automatic contention detection and amelioration for data-intensive operations

- A generic framework (similar to Google's MapReduce) that
  - Efficiently parallelizes generic tasks
  - Automatically detects contention
  - Scales on multi-core CPUs
  - Makes programmer's life easier :-)
- Based on
  - J. Cieslewicz, K. A. Ross, K. Satsumi, and Y. Ye. “Automatic contention detection and amelioration for data-intensive operations.” In *SIGMOD* 2010.
  - Y. Ye, K. A. Ross, and N. Vesdapunt. Scalable aggregation on multicore processors. In *DaMoN* 2011

To Share or not to share

- Independent computation
  - Shared-nothing (disjoint processing)
  - No coordination (synchronization) overhead
  - No contention
  - Each thread use only 1/N of CPU resources
  - Merge step required
- Shared computation
  - Common data structures
  - Coordination (synchronization) overhead
  - Potential contention
  - All threads enjoy all CPU resources
  - No merge step required

Thread level parallelism

- On-chip coherency enables fine-grain parallelism
  - that was previously unprofitable (e.g., on SMPs)
- However, *beware*:
  - Correct parallel code does not mean no contention bottlenecks (hotspots)
  - Naive implementation can lead to huge performance pitfalls
  - Serialization due to shared access
  - E.g., many threads attempt to modify the same hash cell

Aggregate computation

- Parallelizing simple DB operation:

```
SELECT R.G, count(*), sum(R.V)
FROM R
GROUP BY R.G
```

- What happens when values in R.G are highly skew?
- What happens when number of cores is much higher than |G|?
- Recall the key question: to share or not to share?

Atomic CAS instruction

- Notation: `CAS( &L, A, B )`
- The meaning:
  - Compare the old value in location L with the expected old value A. If they are the same, then exchange the new value B with the value in location L.
  - Otherwise do not modify the value at location L because some other thread has changed the value at location L (since last time A was read). Return the current value of location L in B.
- After a CAS operation, one can determine whether the location L was successfully updated by comparing the contents of A and B.

Atomic operations via CAS

```
atomic_inc_64( &target ) {
    do {
        cur_val = Load(&target);
        new_val = cur_val + 1;
        CAS(&target, cur_val, new_val);
    } while (cur_val != new_val);
}
atomic_dec_64( &target );
atomic_add_64( &target, value);
atomic_mul_64( &target, value);
...
```

What is contention then?

- Number of CAS retries

Measuring contention (pseudo-code)

```
my_atomic_inc_64( &target, &cas_counter ) {
    do {
        cur_val = Load(&target);
        new_val = cur_val + 1;
        CAS(&target, cur_val, new_val);
        cas_counter++;
    } while (cur_val != new_val);
}
my_atomic_dec_64( &target, &cas_counter );
my_atomic_add_64( &target, value, &cas_counter);
my_atomic_mul_64( &target, value, &cas_counter);
...
```

Measuring contention (assembly code)

```
.inline my_atomic_add_64,0 ! %o1 contains update value
ldx [%o0], %o4             ! load current sum into %o4;
ld [%o2], %o5              ! load update-counter into %o5
1:
inc 1, %o5                 ! increment update-counter
add %o4, %o1, %o3          ! add value to current sum; put in %o3
casx [%o0], %o4, %o3       ! compare-and-swap %o3 into memory
                           ! location of sum;
                           ! %o4 contains the value seen
cmp %o4, %o3               ! check if compare-and-swap succeeded
                           ! i.e., if %o4 is equal to %o3
bne,a,pn %xcc, 1b          ! if not, retry loop starting at 1:
    mov %o3, %o4           ! statement executed even when branch
                           ! taken; %o4 now has a more recent value
                           ! of the current sum and we have to add
                           ! %o1 over again
st %o5, [%o2]              ! store the update-counter
.end
```

Contention management

- Applies only to commutative operations
  - I.e., changing the order of the operands does not change the result
  - E.g., aggregation and partitioning
- General idea:
  - Perform operation on X and measure contention
  - Create extra version of X when contented
  - Spread the subsequent accesses among the two copies of X
  - Combine the results at the end

Framework

- Requires 4 user-defined template functions
  - `create-clone`: how a new version is created (x = 0)
  - `combine`: how multiple versions are merged (x + x1)
  - `simple-update`: how the new value of a data item is obtained from the current value and an update (x += v)
  - atomic-update: user defined function (next slide)
- Framework takes care
  - When to clone
  - Which clone is accessed by which thread

Example of atomic-update

- ```
  bool AggregatorAtomicUpdate(Aggregator *agg,
                              const uint64_t value) {
      int32_t cas_counter = 0;

      my_atomic_inc_64(&agg->count, &cas_counter);
      my_atomic_add_64(&agg->sum, value, &cas_counter);
      return (3 < cas_counter);
  }
  ```

- Recall:

  ```
  SELECT R.G, count(*), sum(R.V)
  FROM R
  GROUP BY R.G
  ```
  ```

Techniques for managing contention

- Main concerns:
  - What information to maintain about the current number of clones?
  - How to map threads to clones in a balanced fashion?
- Two broad approaches for managing clones:
  - Global
  - Local

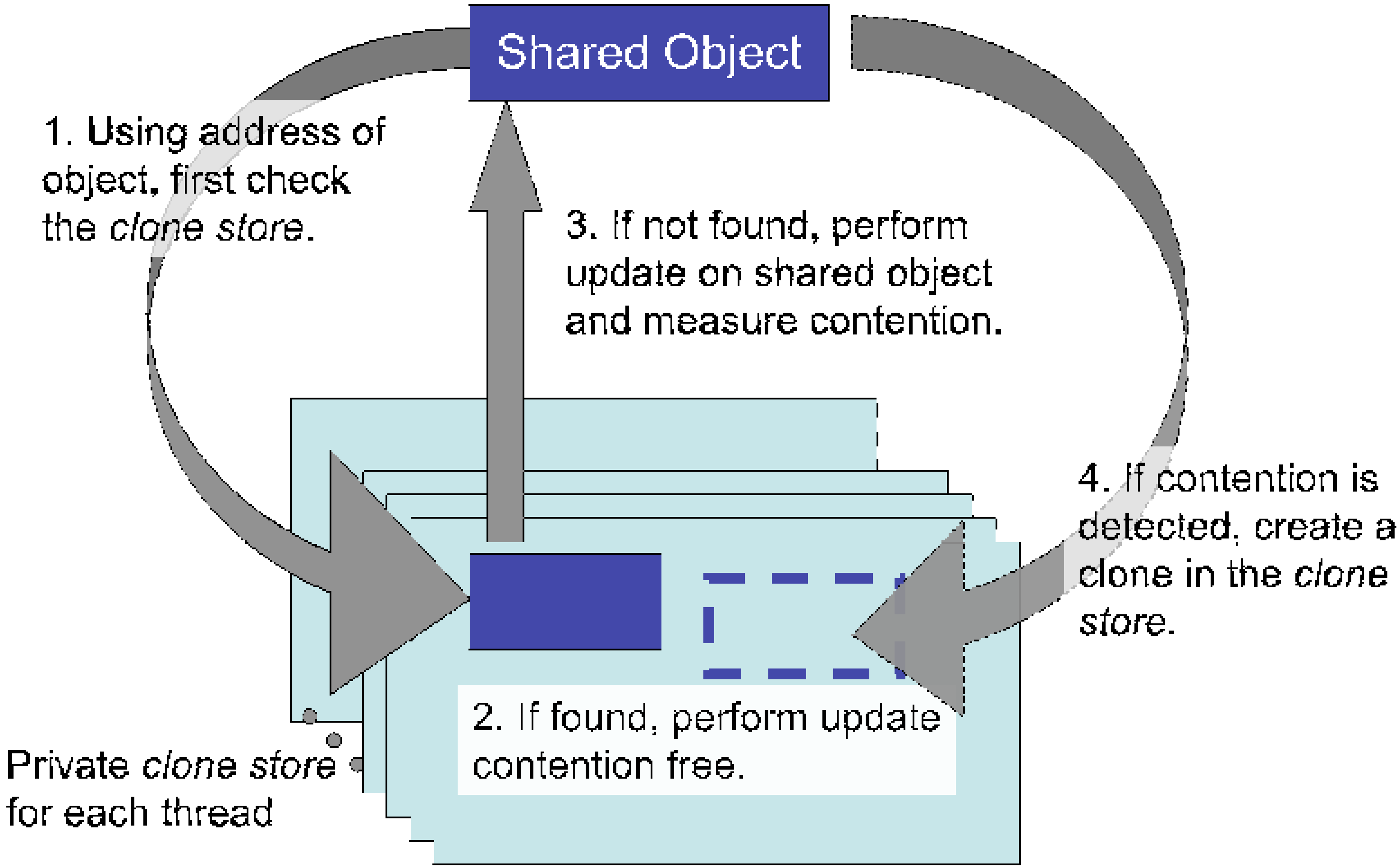
Managing clones globally

- New clones are created in *shared* address space
- Clone allocation happens in response to a *single* contention event (no threshold counters)
- The number of clones is always *doubled*
  - E.g., we can get to 64 clones of a heavy-hitter element after 6 contention steps
  - With few very popular items, each thread might end up having its own clone (no atomic operations needed afterwards!)

Managing clones locally

- Each thread creates clones in a local table used by that thread alone
- Table size is kept small
  - e.g., smaller than the thread's share of the L1 data cache
- When the table is full, new insertions are accomplished by spilling an existing value into the global data element

Managing clones locally (cont.)



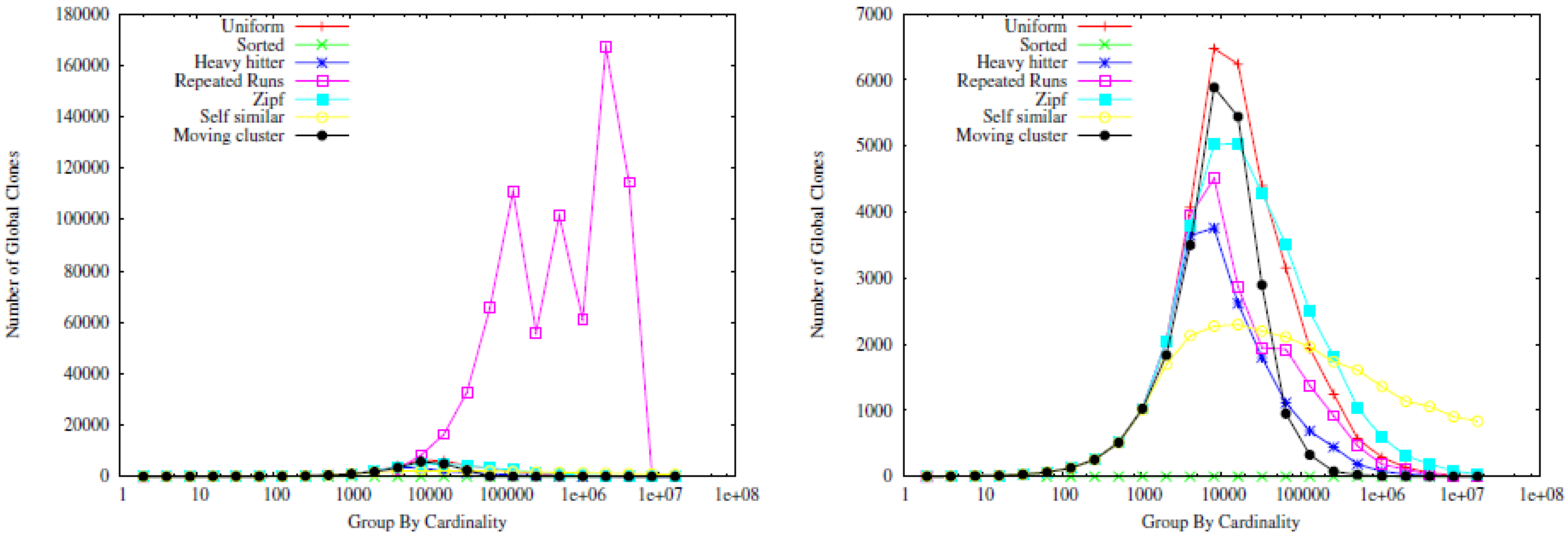
Experimental platforms

Platform	Sun T2	Intel Nehalem Xeon E5620
Operating System	Solaris 10	Ubuntu Linux 2.6.32.25-server
Processors	1	2
Cores/processor (Threads/core)	8 (8)	4 (2)
RAM	32GB	48 GB
L1 Data Cache	8KB per core	32 KB per core
L1 Inst. Cache	16KB per core	32 KB per core
L2 Cache	4MB, 12-way Shared by 8 cores	256 KB per core
L3 Cache	N.A.	12MB, 16-way Shared by 4 cores

Input data

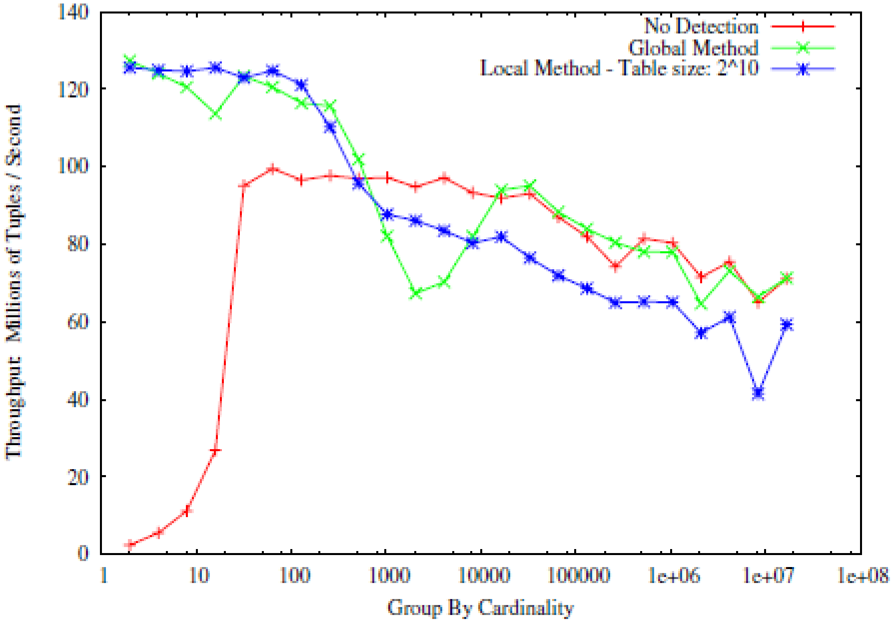
- Refers to the characteristics of the group-by key in the input relation
- Synthetically generated distributions ($N = 2^{24}$):
  - Uniform
  - Sorted (1 1 1 2 3 3 4 5 … N)
  - Heavy hitter (50%)
  - Repeated-run (1 2 3 … N 1 2 3 … N 1 2 …)
  - Zipf (exponent of 0.5)
  - Self-similar (80-20 proportion)
  - Moving-cluster (locality window)
- During input generation a targeted group-by cardinality is specified

Cache and memory issues

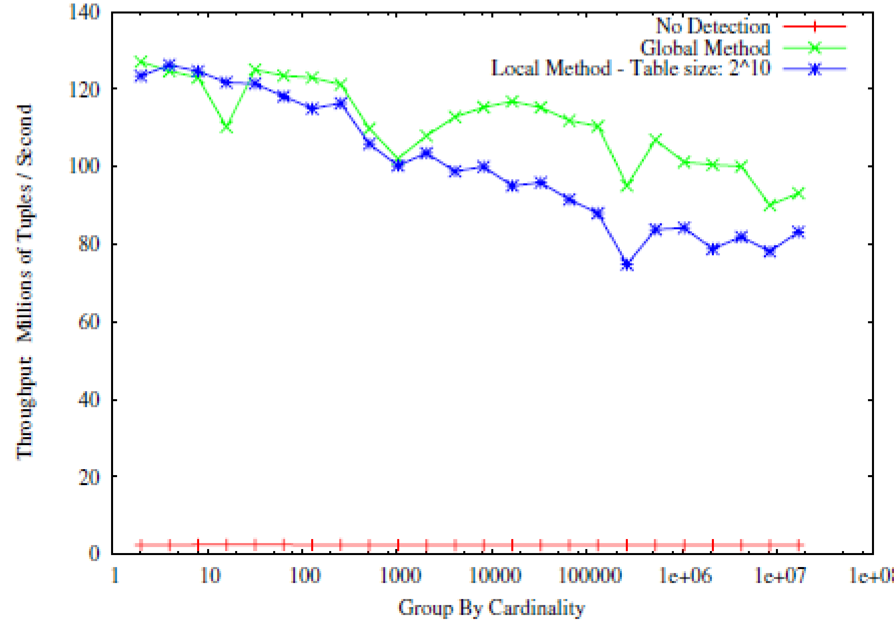


(a) Sequential reading (b) Prime increment reading

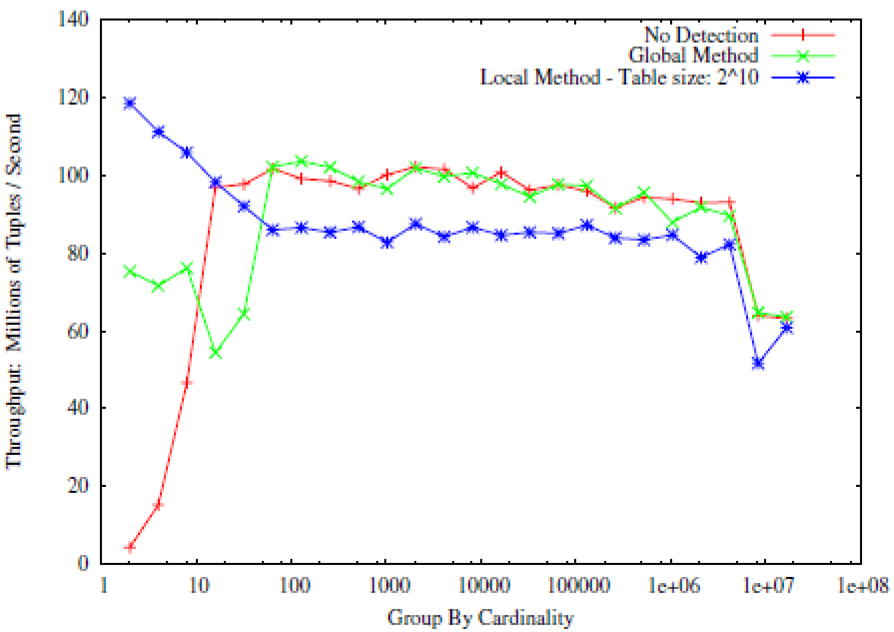
Number of group by values where contention has been detected and at least one clone constructed



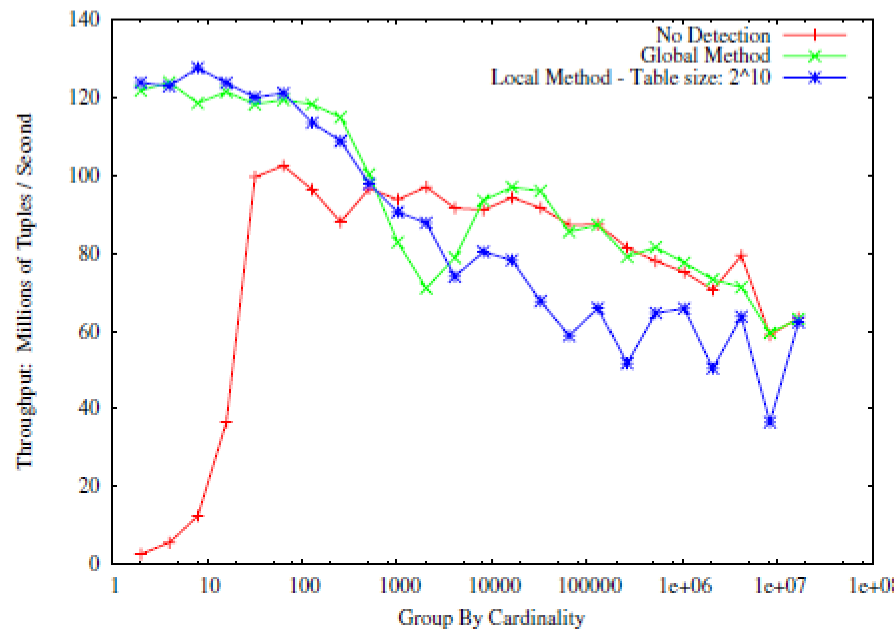
(a) Uniform



(b) Heavy Hitter

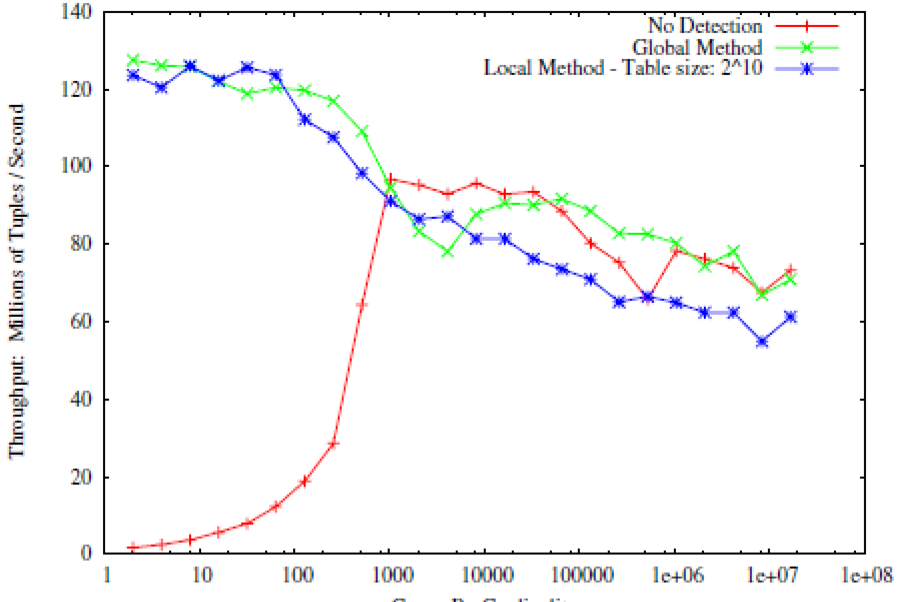


(c) Sorted

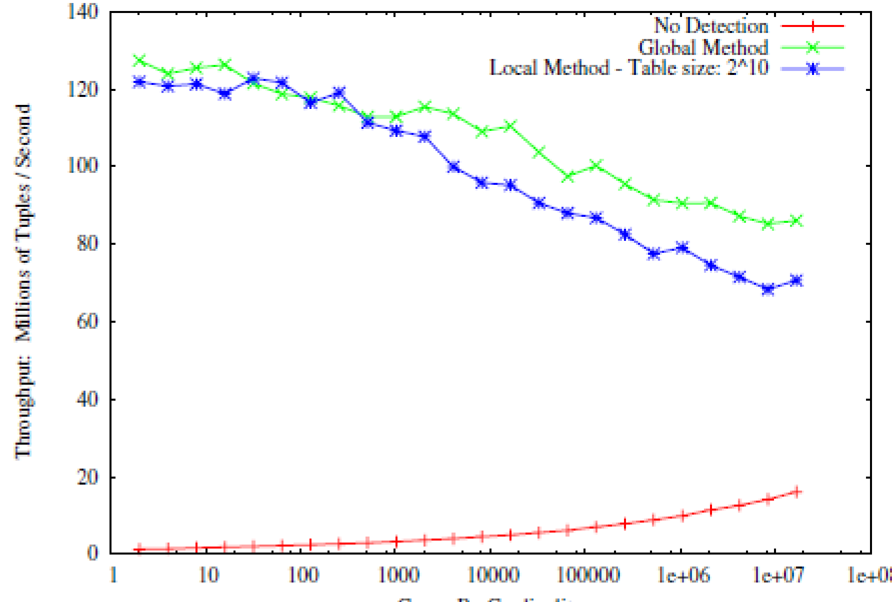


(d) Repeated Runs

Results

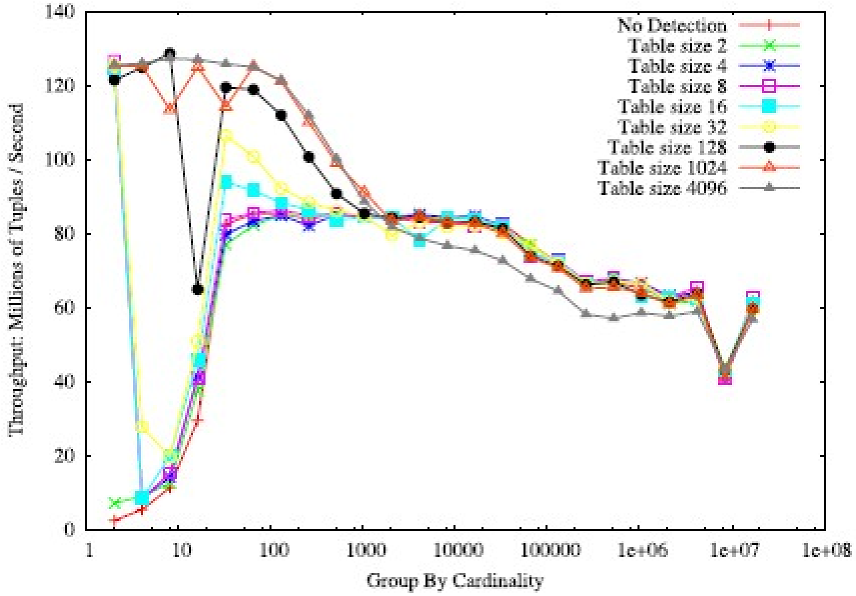


(e) Zipf



(f) Self Similar

Effects of the local table size



Conclusions

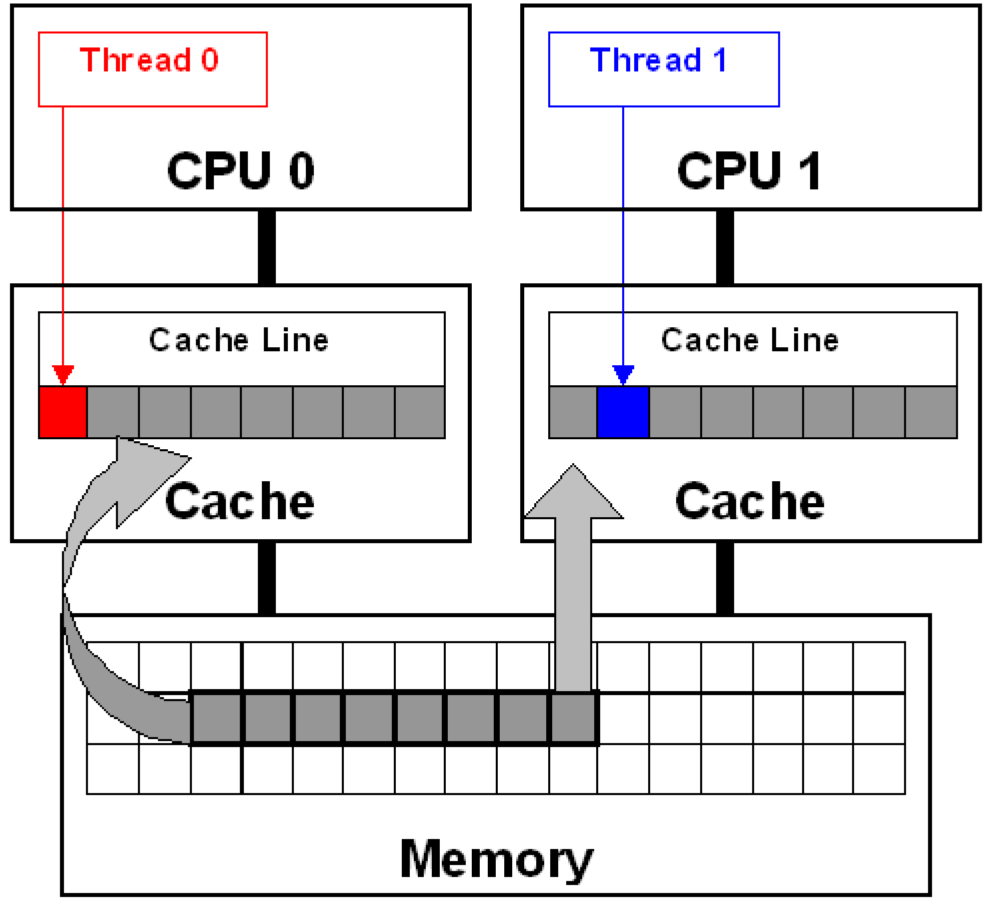
- Automatic contention detection
- Effective contention amelioration
- Both proposed schemes (global and local) mitigate contention
  - Global slightly faster
  - Local uses less memory
- However
  - Works just for commutative operations
  - Different architectures favor different approaches

Outline

- Part 1
  - Background
  - Current multicore CPUs
- Part 2
  - To share or not to share
- **Part 3**
  - **Demo**
  - **War story**

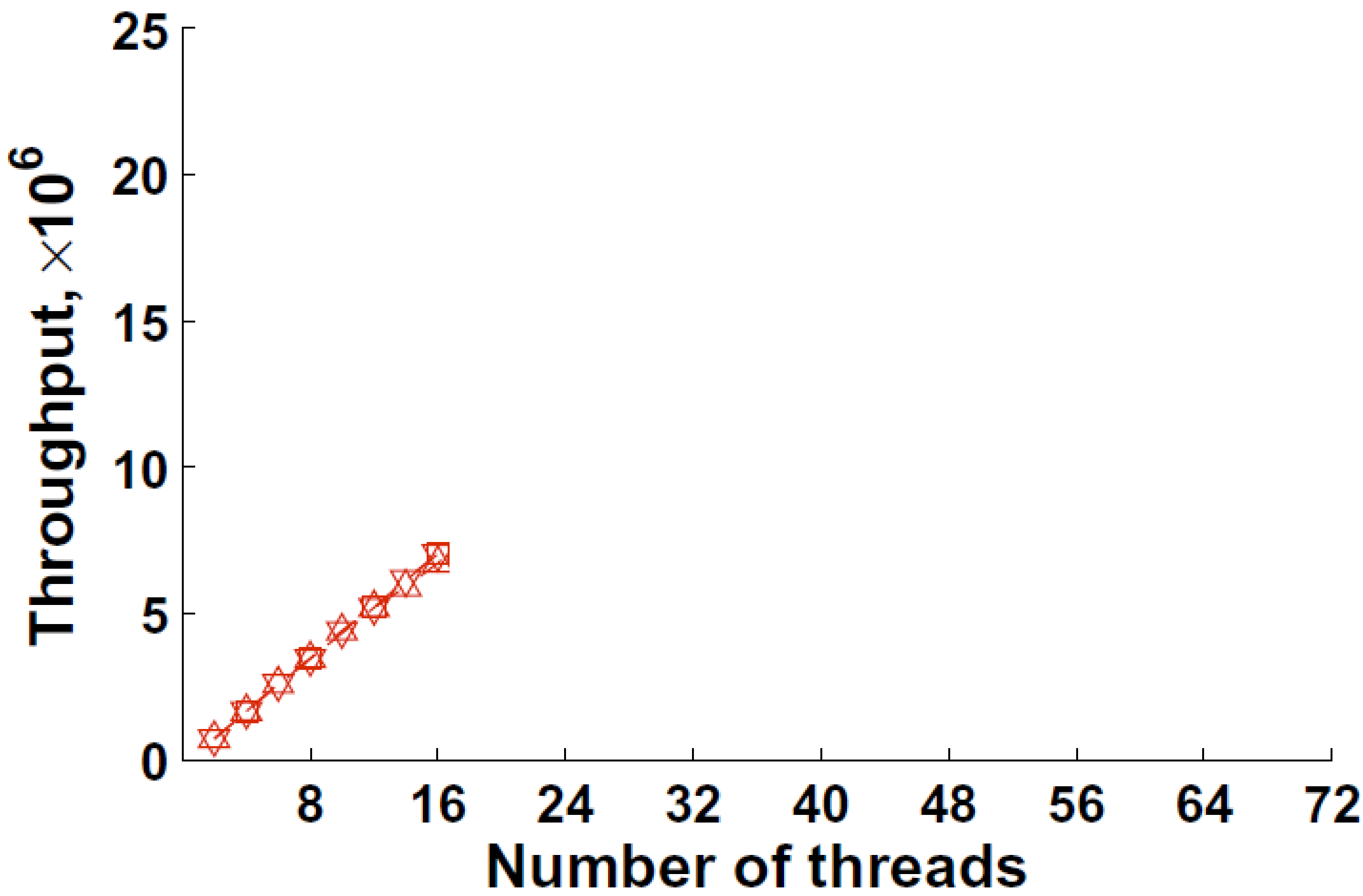
Demo: false sharing

- Threads operate on different variables
- But variables reside on the same cache line



Demo: NUMA effects

War story



Throughput, ×10^6
25
20
15
10
5
0
8
16
24
32
40
48
56
64
72
Number of threads

Looking for a master thesis topic?

- ACM SIGMOD 2014 Programming Contest
- ACM SIGSPATIAL GIS CUP 2014

References

[1] Samuel H. Fuller and Lynette I. Millett, "The Future of Computing Performance: Game Over or Next Level?" The National Academies Press, 2010. [link]

[2] CPU Overclocking World Records [link]

[3] D. Molka, R. Schöne, D. Hackenberg, & M. S. Müller. "Memory performance and SPEC OpenMP scalability on quad-socket x86_64 systems." In *ICA3PP*, 2011.

[4] D. Molka, D. Hackenberg, R. Schone, and M. S. Muller. "Memory performance and cache coherency effects on an intel nehalem multiprocessor system." In *PACT* 2009.

[5] Albutiu, M. C., Kemper, A., & Neumann, T. "Massively parallel sort-merge joins in main memory multi-core database systems." In *VLDB* 2012.

[6] J. Cieslewicz, K. A. Ross, K. Satsumi, and Y. Ye. "Automatic contention detection and amelioration for data-intensive operations." In *SIGMOD* 2010.

[7] Y. Ye, K. A. Ross, and N. Vesdapunt. "Scalable aggregation on multicore processors." In *DaMoN* 2011.

MADALGO
CENTER FOR MASSIVE DATA ALGORITHMICS

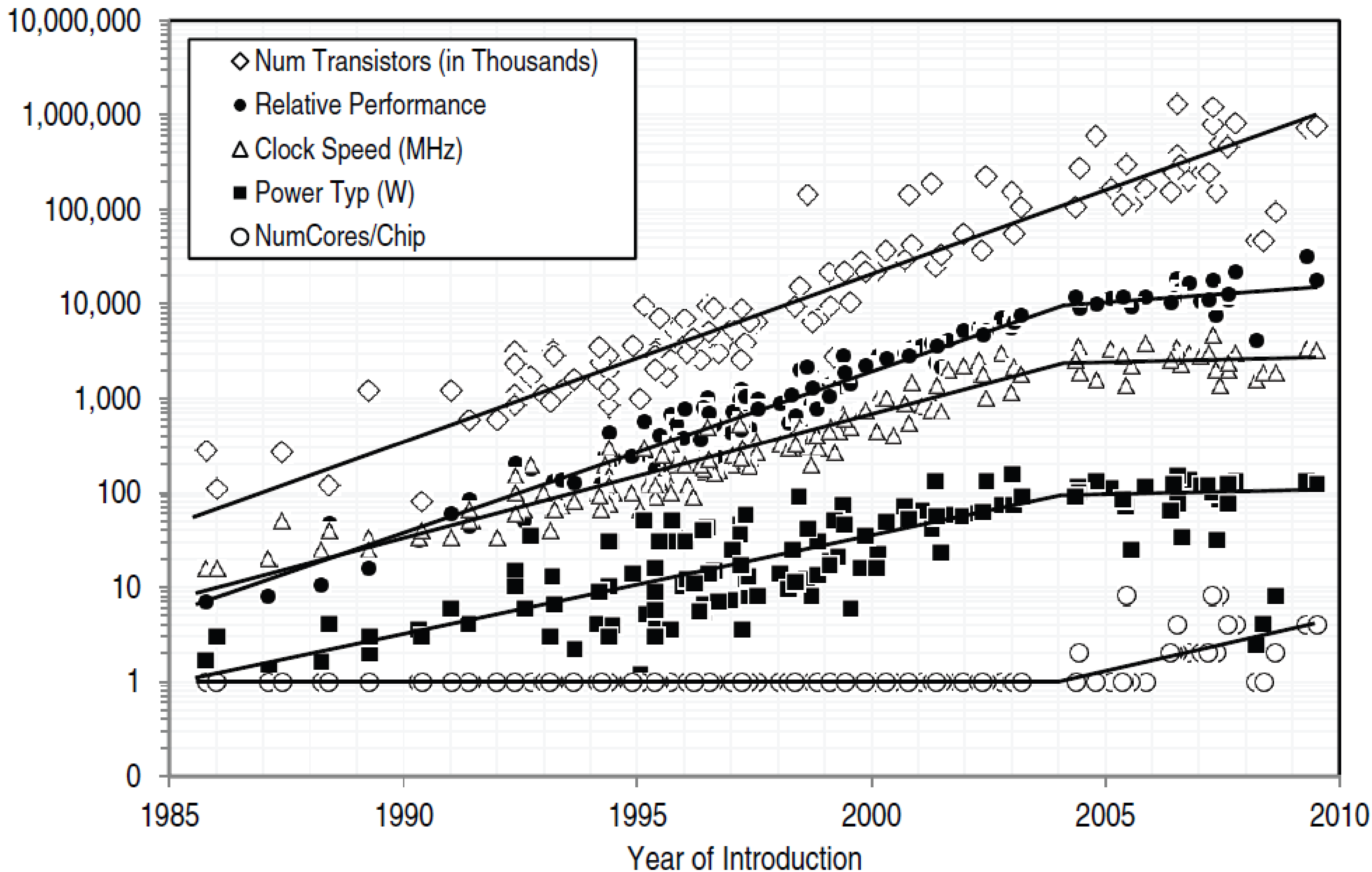
Thank you

Darius Sidlauskas
Post-doc

Contact: dariuss@madalgo.au.dk

Danmarks Grundforskningsfond
Danish National Research Foundation

All in one [1]


Num Transistors (in Thousands)
Relative Performance
Clock Speed (MHz)
Power Typ (W)
NumCores/Chip
10,000,000
1,000,000
100,000
10,000
1,000
100
10
1
0
1985
1990
1995
2000
2005
2010
Year of Introduction