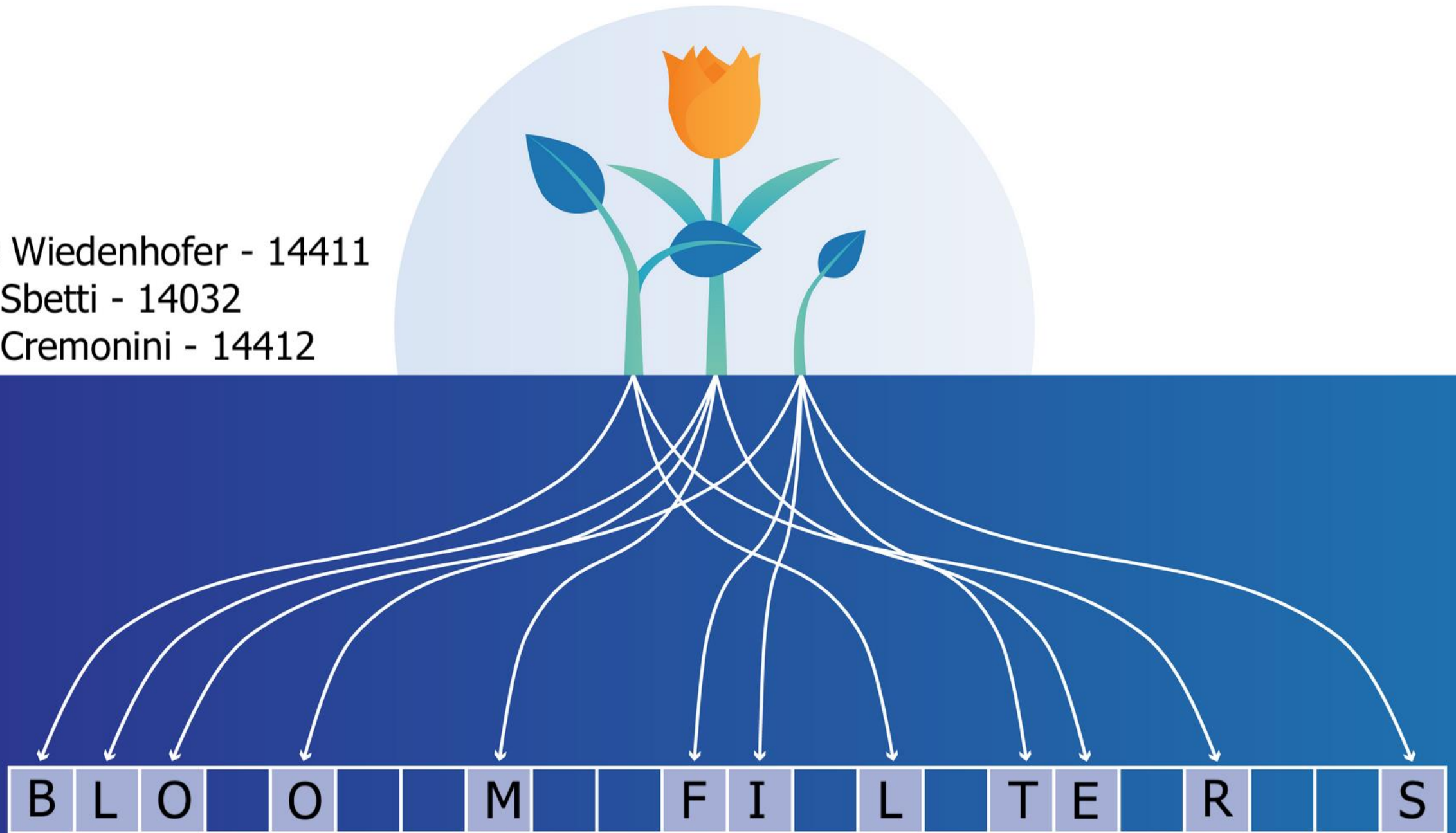


Hannes Wiedenhofer - 14411
Davide Sbeti - 14032
Davide Cremonini - 14412





The goal



- Comparing different approaches for parallelising Bloom filters
- Focusing on CPU parallelisation (OpenMP)
- Testing the implementations on different machines

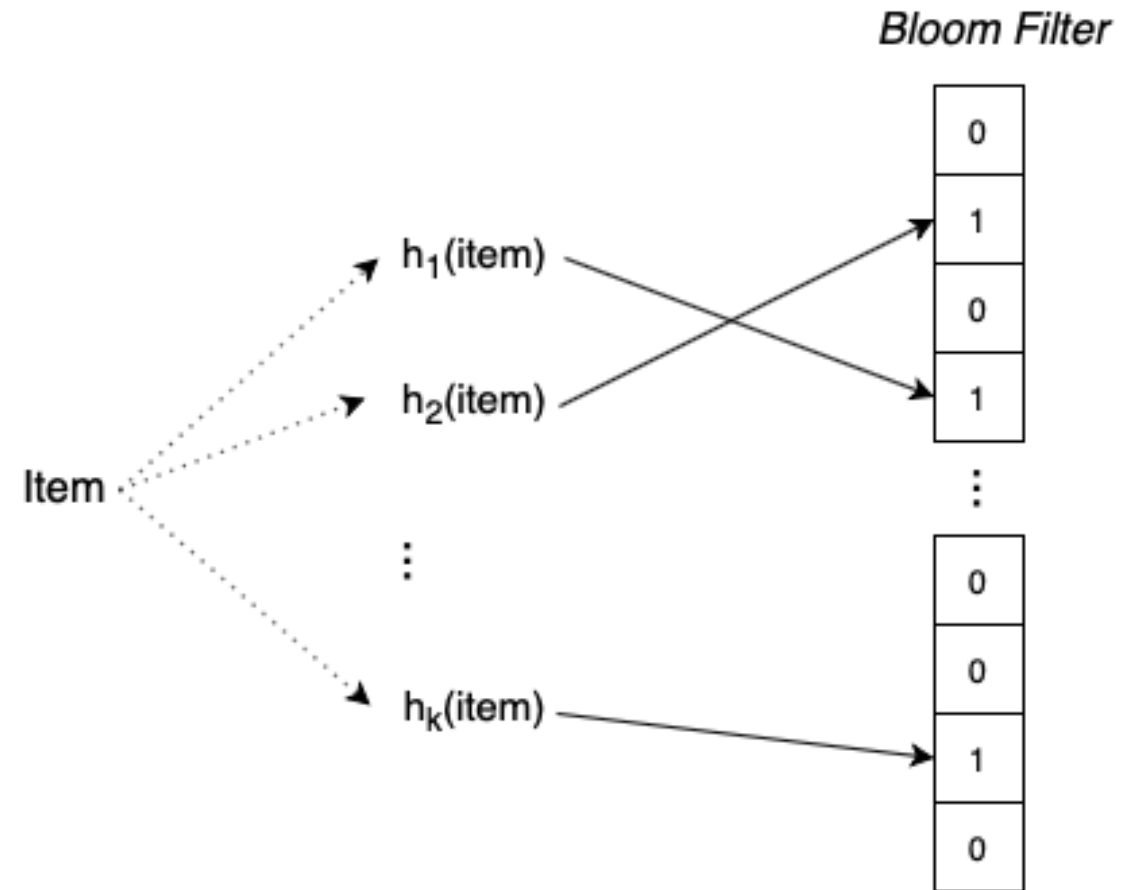


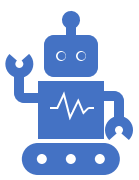
Bloom Filters

- space efficient probabilistic data structures
 - used to efficiently identify whether an element is (approximately) present in a set
 - components:
 - an array of n bits
 - k independent random hash functions, each with a range between 0 and $n-1$
-

Bloom Filters

- New element insertion:
 - k positions are obtained from the k hash functions
 - The corresponding bits are set to 1
- Querying the filter:
 - k positions are obtained from the k hash functions
 - If all k bits are set to 1  element could be present
 - If at least one bit is 0  element for sure not present



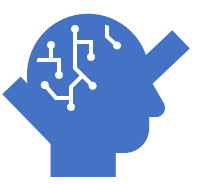


Implementations

- Focusing on CPU parallelization, we decided to use OpenMP
- OpenMP is a scalable model for parallel applications in C/C++
- Different strategies were implemented:
 - Sequential Baseline
 - OpenMP Baseline
 - Atomic
 - Reduction

Common implementation procedure

- Bloom filter uses n 32-bit words
- Seeds generated using Xoshiro pseudo-random generator
- Procedure:
 - Input elements inserted into the filter
 - Error check on insertions
 - False positive check using test set
 - Results (execution time) stored in CSV file



Sequential Baseline

**Implementation of standard
algorithm**

**No parallelization techniques
are used**

Single thread executes operations
sequentially



OpenMP Baseline

- Lock (critical region) on the whole insertion function
 - Only one thread can insert data at a time
 - Queries:
 - Also performed in parallel
 - No critical region is required (reading operation)
 - Approach used also in following implementations
-

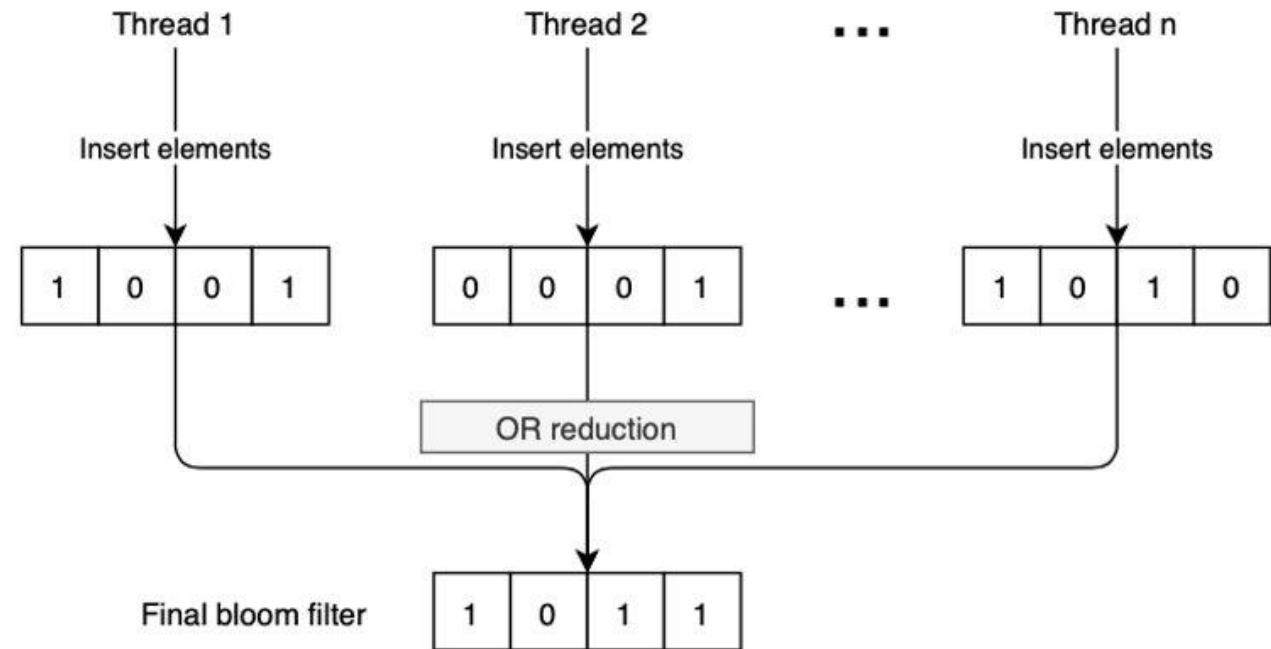


Atomic

- Critical region replaced by an atomic or-operation
 - Filter is locked only when setting the bits
 - Concurrency is still allowed when calculating the hash
-

Reduction

- Each thread has its own entire filter
- All elements assigned to it are inserted in local filter
- No lock is required
- Reduction at the end of insertion process (OpenMP reduction)
 - Result: a single merged filter



Datasets

String length	Number of entries	Notes
10	100	
	1000	
	10.000	
	100.000	
	1.000.000	
	10.000.000	
20	100	
	1000	
	10.000	
	100.000	
	1.000.000	
	10.000.000	
Variable (power law)	100	Datasets generated using the power law, so with variable length (max 60 character) and distribution parameter equal to 2.
	1000	
	10.000	
	100.000	
	1.000.000	
	10.000.000	

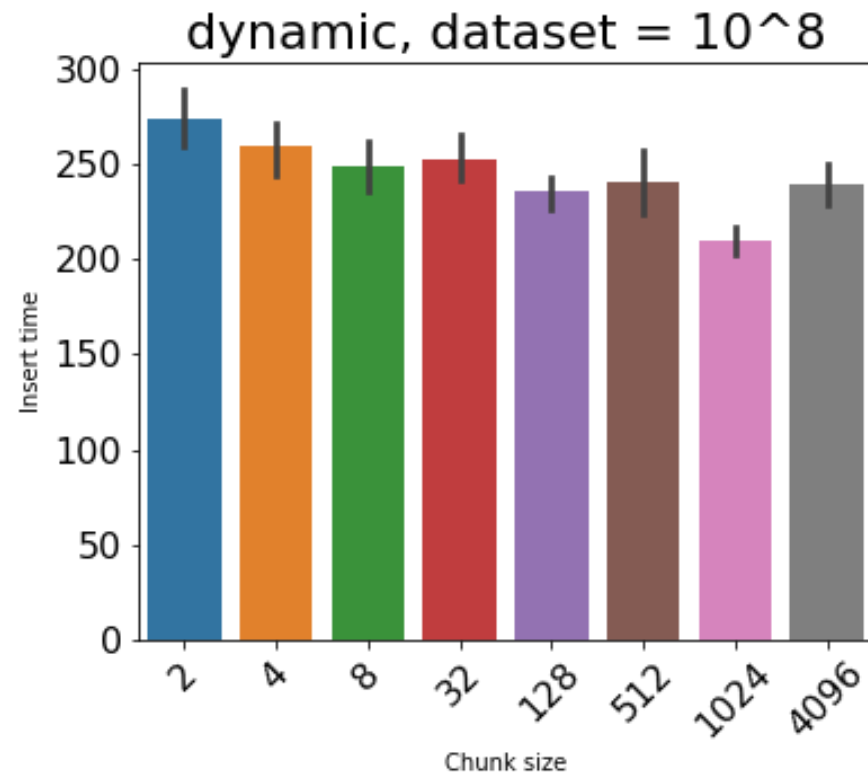
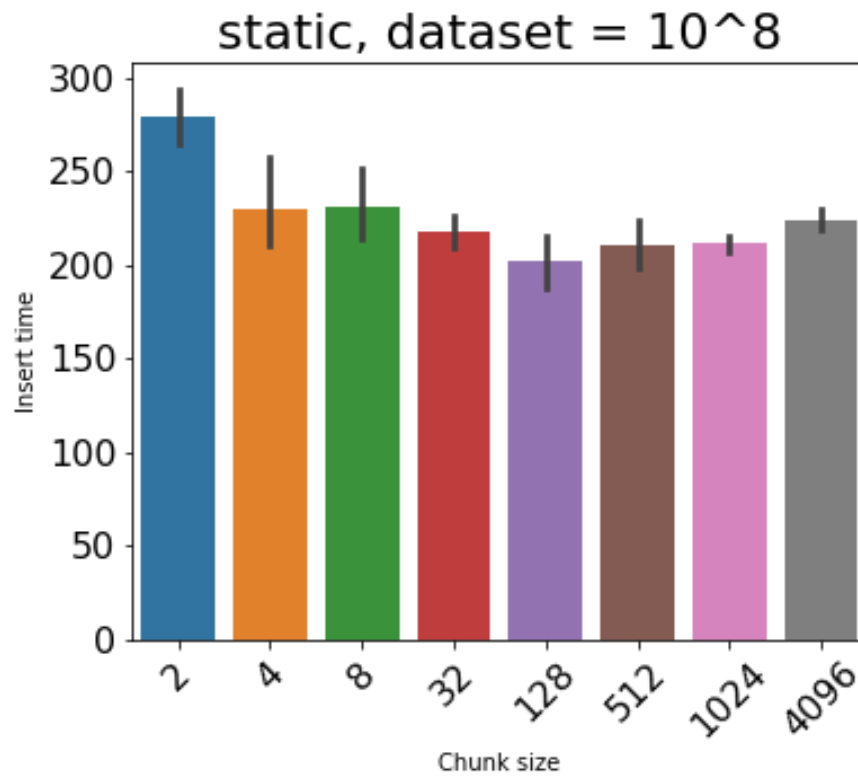
Environment

	Lenovo Legion 5 17IMH05H	Ironmaiden UNIBZ machine
Number of CPUs	1	1
CPU	Intel® Core™ i7 – 10750H	Intel(R) Xeon(R) CPU E5- 2667
CPU Base Frequency	2.60 GHz	2.90 GHz
Number of cores	6	32
Hyperthreading	enabled	enabled
OS	Windows 10 20H2	Ubuntu 16.04.7 LTS



Experiments results

- Different results are reported:
 - Chunk Size
 - Insert Time
 - Query Time
 - Speedup

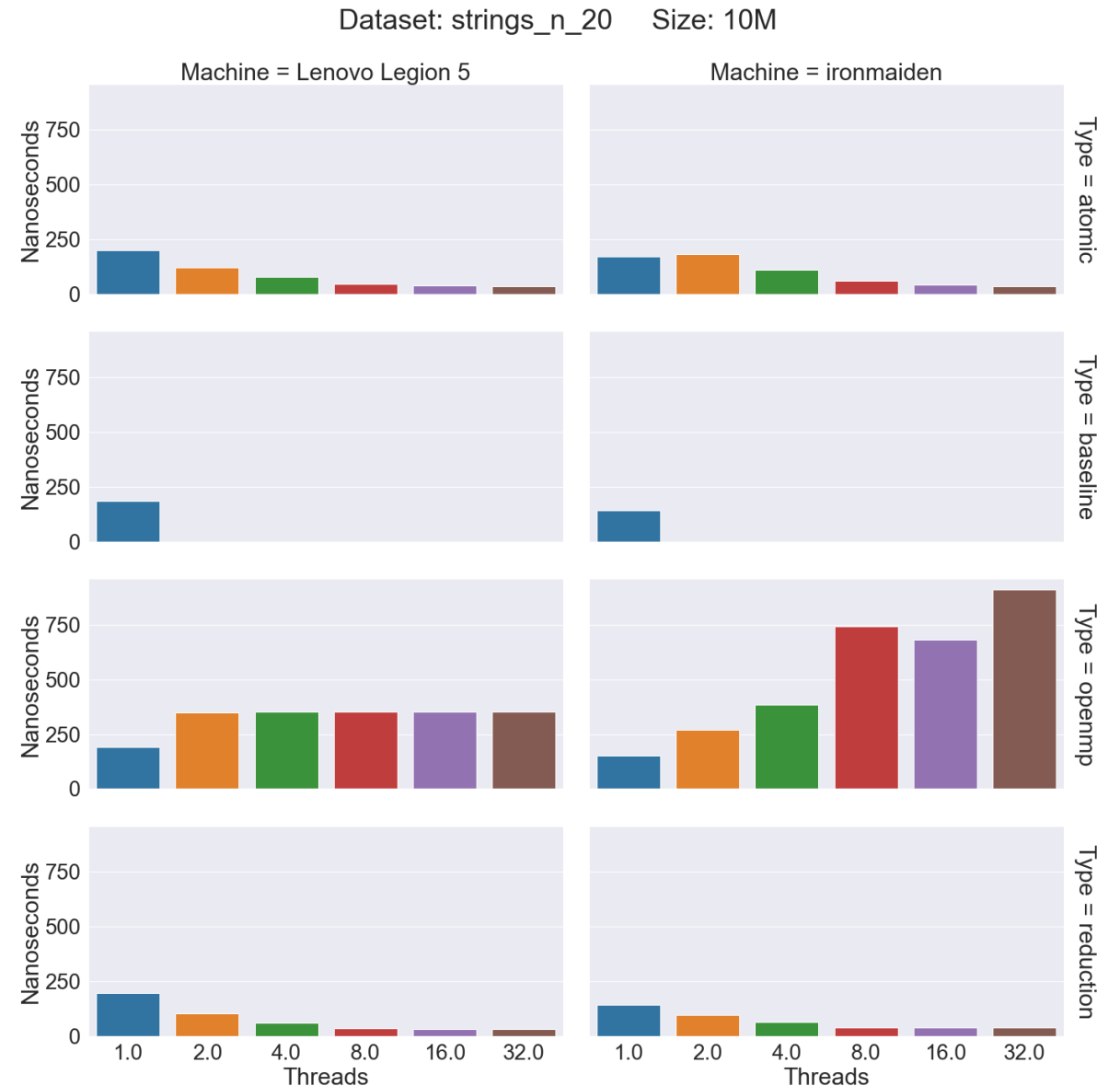


Chunk Size

- Baseline OpenMP with different chunk sizes and schedulers
- We choose 128 elements as chunk size, used in the following experiments
- Intervals show difference is not statistically significant

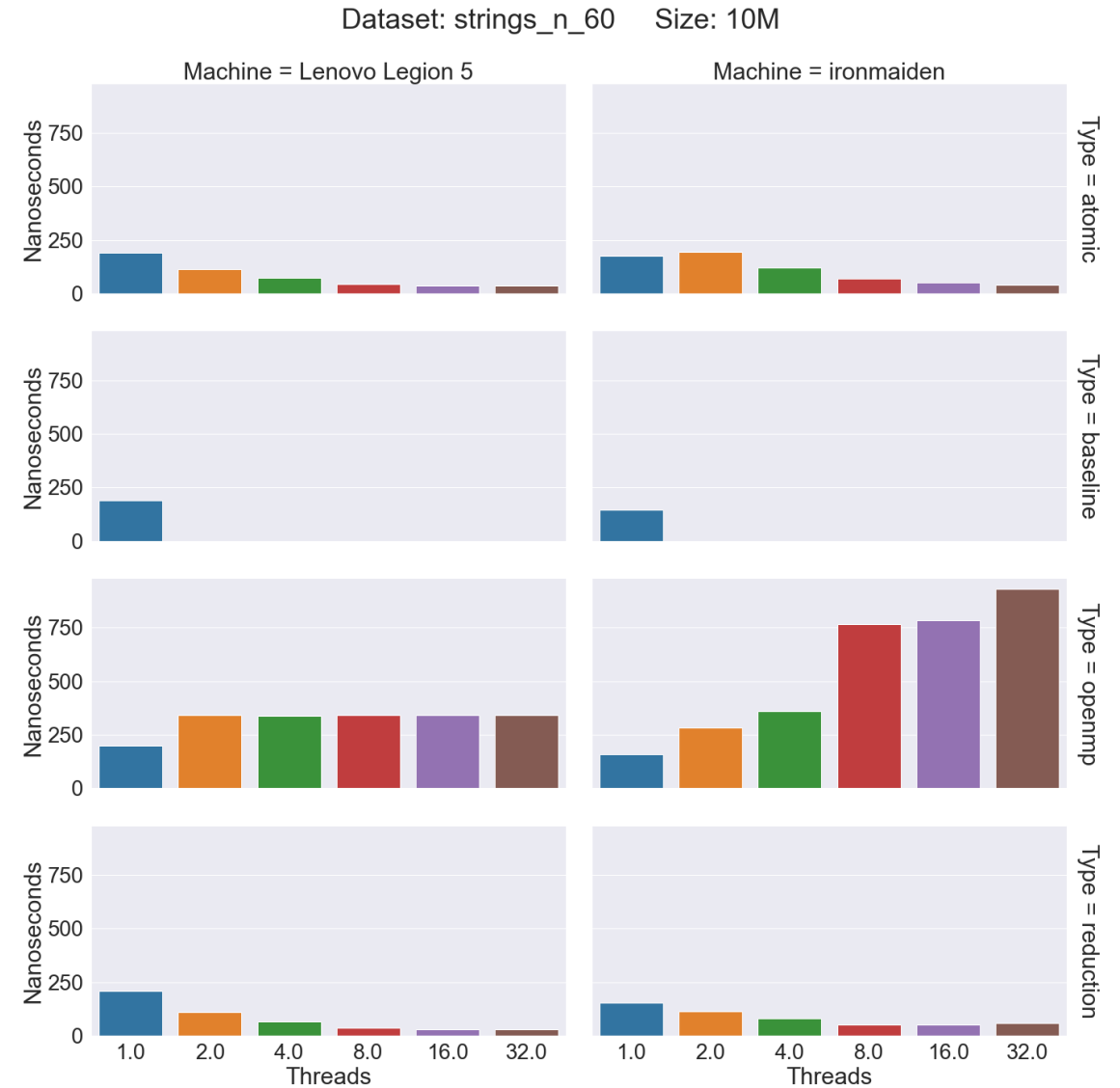
Insert Time

- Atomic and reduction appear to be the fastest
 - This can be explained by the short-time lock / absence of it
- Most surprising result: OpenMP
 - Deterioration of performance with more threads
 - This occurs because it translates to a sequential algorithm with overhead (creating and managing threads)



Insert time – power law

- The usage of variable length strings does not affect the results

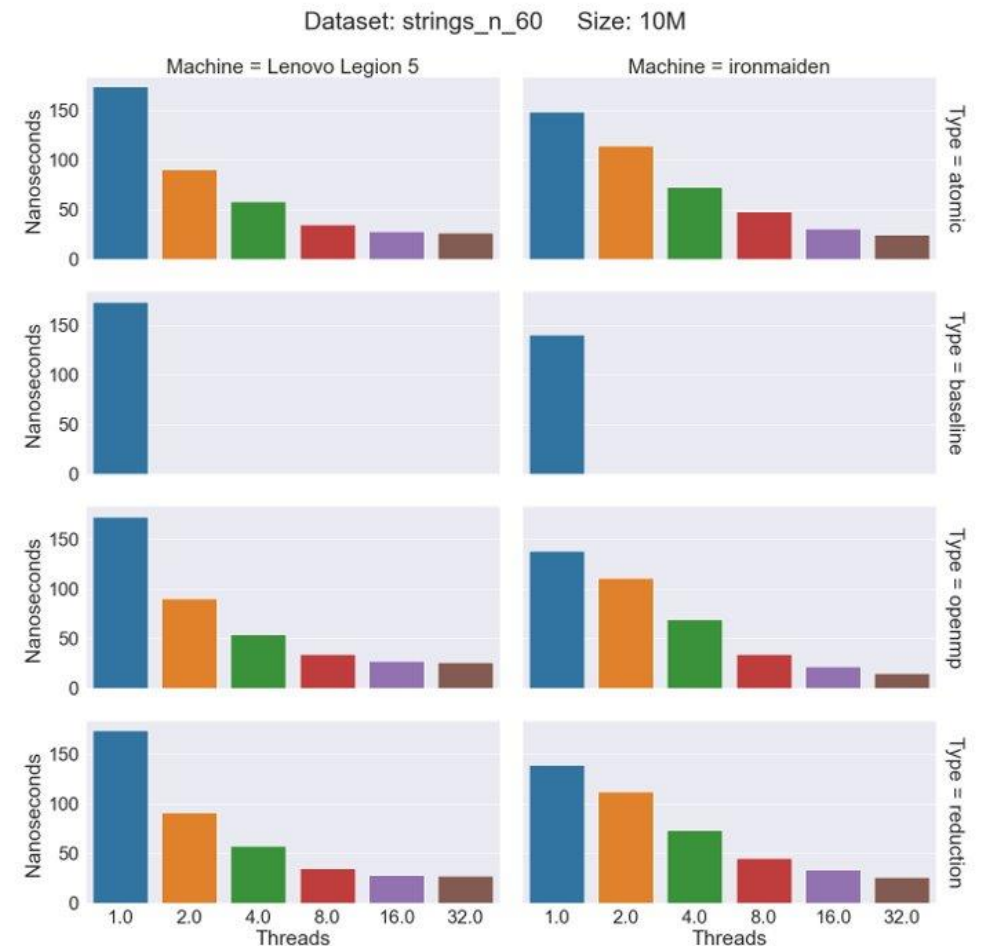
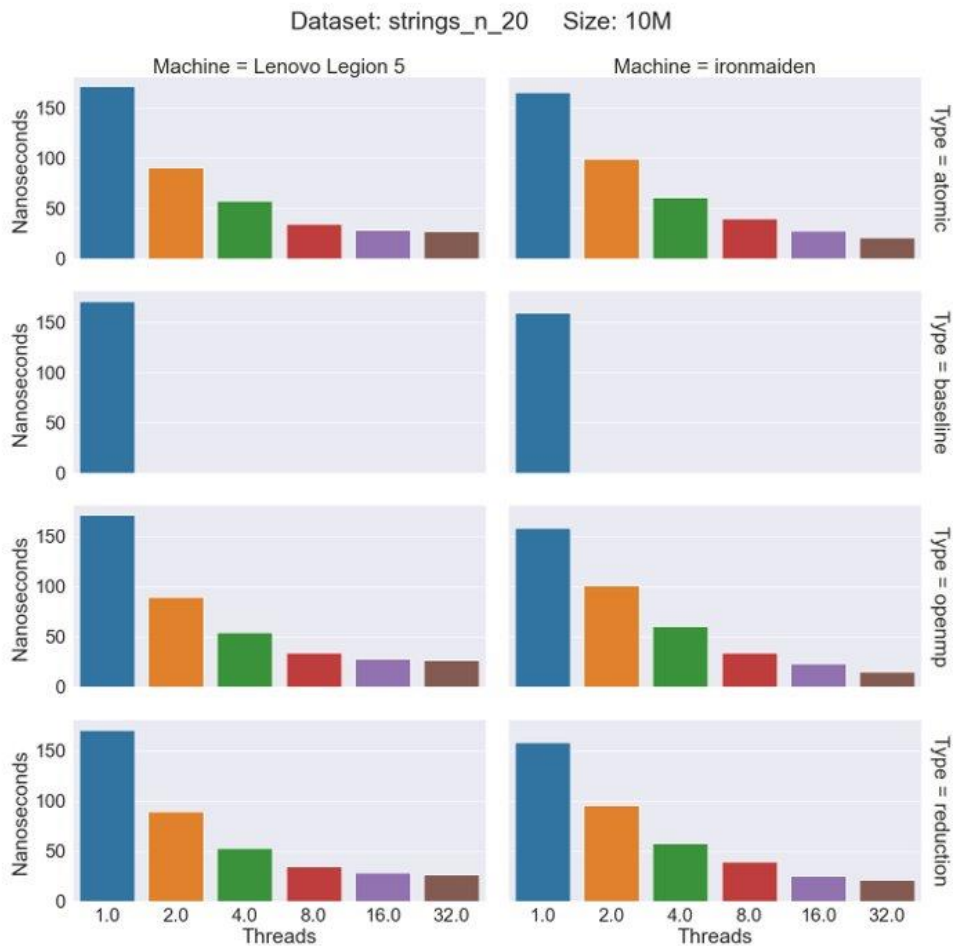


Query Time

- No locks necessary because read-only operation
- Different query times based on the element
 - Elements are not present in the set
 - As soon as a 0 bit is encountered the algorithm returns false

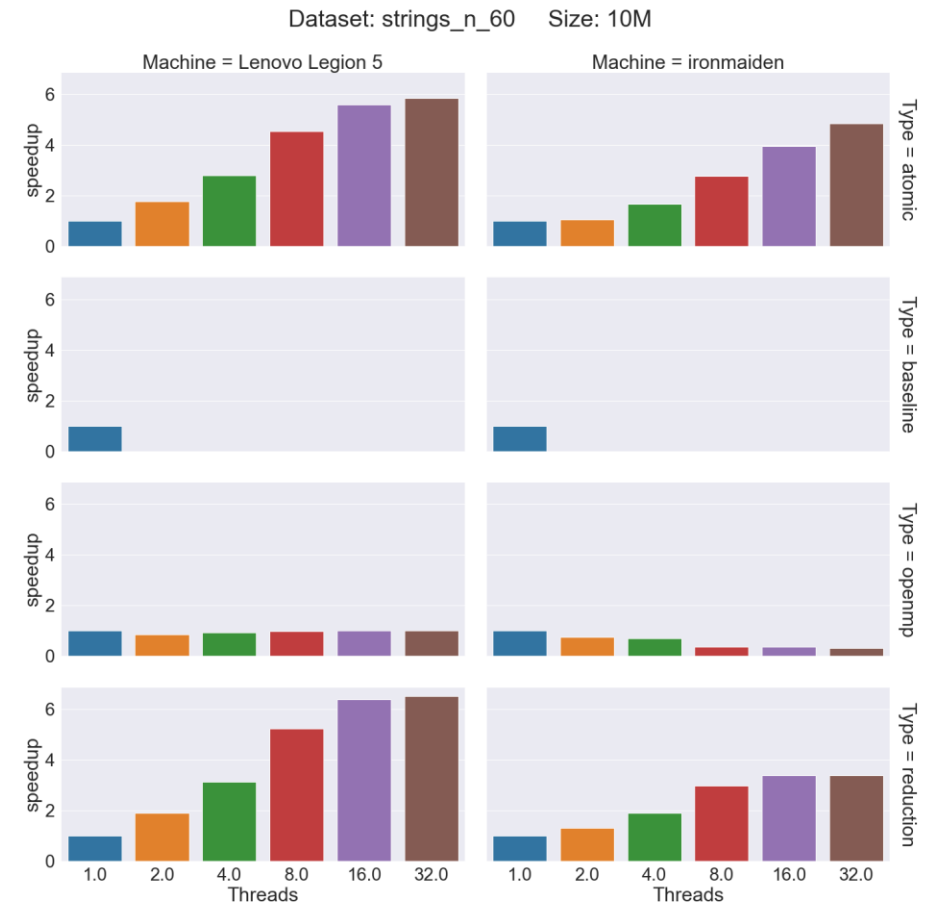
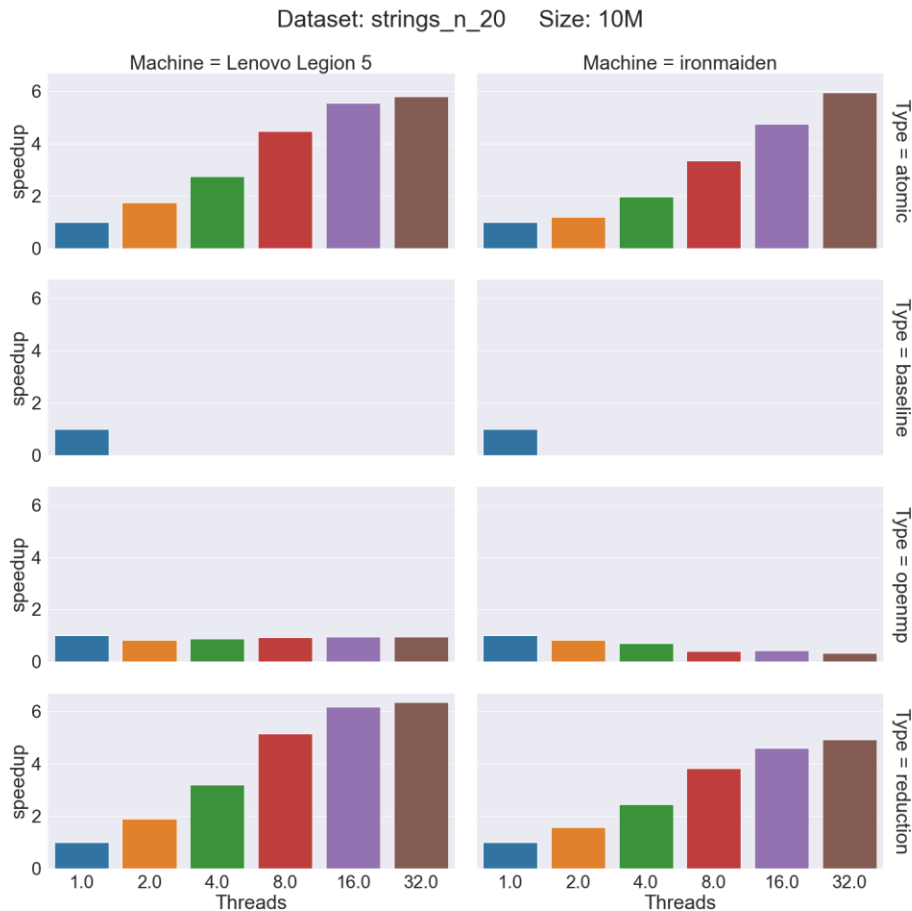
Query time - Results

- Query time scales well on all implementations (including OpenMP)
- This enforces the hypothesis of OpenMP being affected by the general lock



Speedup

- Implementations scale relatively well (exception: OpenMP)
- Reduction scales better on Lenovo PC
- Work load could be too low → large overhead → imperfect speedup



Conclusions

A thick yellow horizontal bar spans the width of the slide, with a vertical yellow bar extending downwards from its right end.

- It is possible to effectively parallelize Bloom filters
- Fastest algorithms: atomic and reduction
- OpenMP with lock on entire filter not effective (transforms to sequential)
- The experiments could be repeated using GPUs

Thank you for your attention

