PARALLELIZATION TECHNIQUES IN CONCURRENT DATA STRUCTURES AND ALGORITHMS

Diploma Thesis

ΧΡΙΣΤΙΝΑ ΧΡ. ΓΙΑΝΝΟΥΛΑ

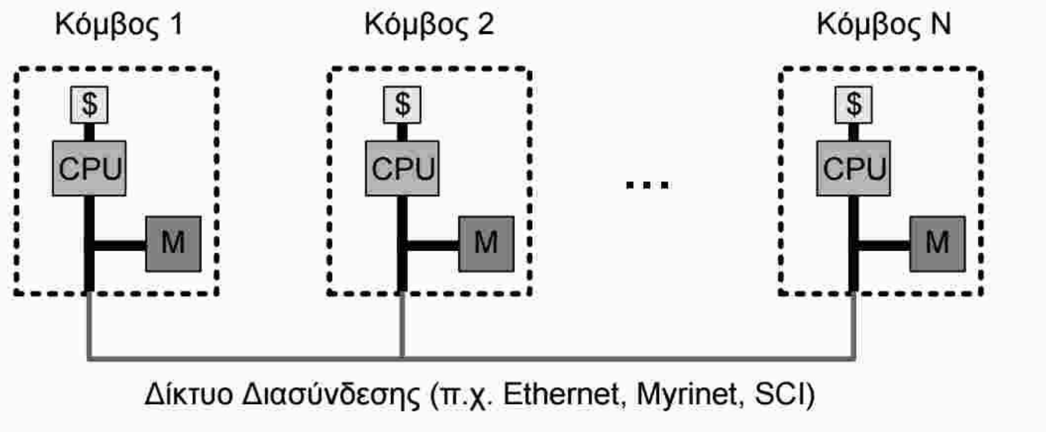
INTRODUCTION

Moore's law

- the number of transistors in an integrated circuit will double approximately every two years

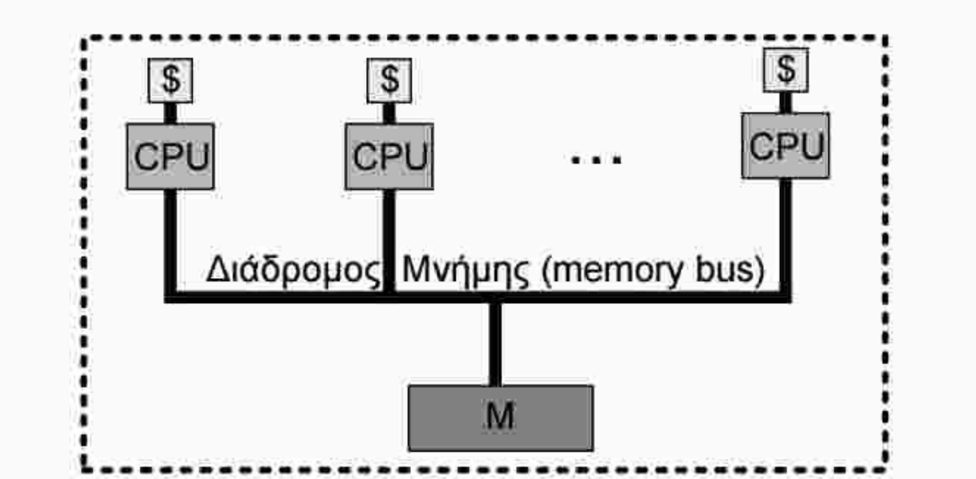
Parallel Architectures

- Shared memory architecture
- Distributed memory architecture
- Hybrid memory architecture



Shared memory architecture

- Uniform Memory Access (UMA)
- Non-Uniform Memory Access (NUMA)

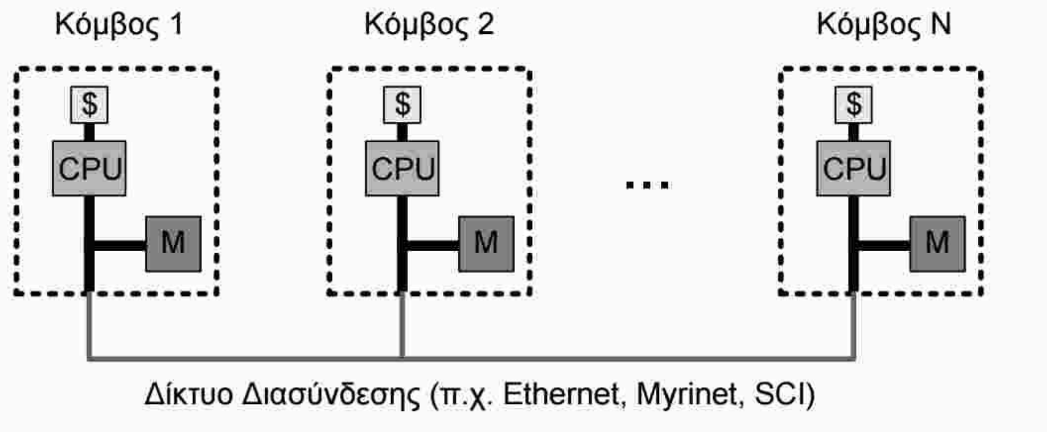


Moore's law

- the number of transistors in an integrated circuit will double approximately every two years

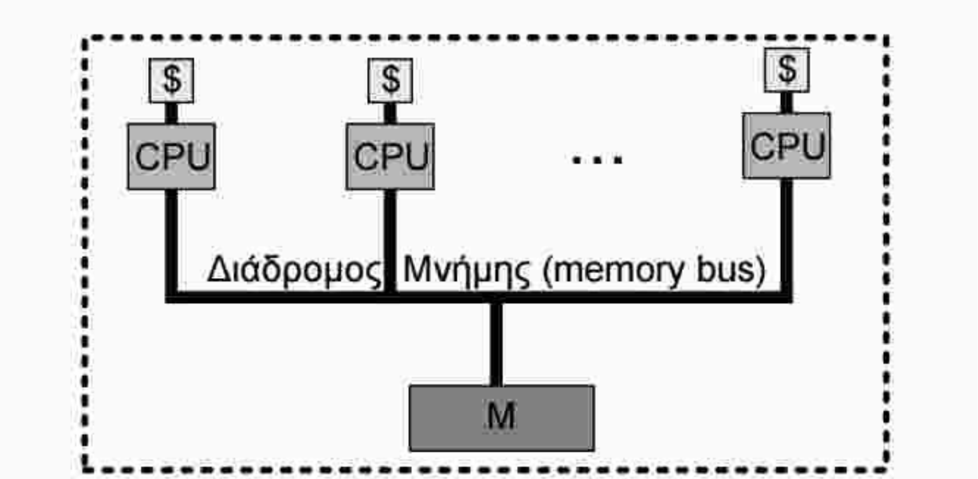
Parallel Architectures

- Shared memory architecture
- Distributed memory architecture
- Hybrid memory architecture



Shared memory architecture

- Uniform Memory Access (UMA)
- Non-Uniform Memory Access (NUMA)



Redesign:

- **Data structures**
- **Algorithms**

Synchronization

- Threads communicate with each other
- Threads execute operations in the same shared data
- Be executed as if running in isolation
- Particular segment of program: critical section

Synchronization techniques

- Mutual Exclusion (blocking scheme)
- Atomic operations (non-blocking scheme)
- Transactional memory (extracting instruction groups to atomic transactions)

Listing 1.1: Inconsistencies due to lack of synchronization

Thread 1	**Thread 2**
`var1 = counter;`	`var2 = counter;`
`var1 += 1;`	`var2 -= 1;`
`counter = var1;`	`counter = var2;`

CONCURRENT BINARY SEARCH TREES

Concurrent Data Structures

- Multiple threads access data simultaneously

Concurrent Search trees

- Locate specific values from within a set (key-value pairs)
- BST: no balance
- AVL, Red-Black trees: height balanced (rotations and recolorings for rebalance)

Techniques for constructing concurrent data structures

- Coarse-grained locking (mutual exclusion)
  - ✓ Lock granularity: the entire shared data are locked via a global lock
  - ✓ Easy to program
- Fine-grained locking
  - ✓ Multiple locks, small granularity are used to protect the smallest possible part of the data structure
  - ✓ Hard to program
  - ✓ Avoid deadlock: acquire locks in the same direction (global order)
- Lock-free programming (non-blocking scheme)
  - ✓ Programming without locks, using atomic operations: CAS, TAS TTAS

Internal trees

- Internal nodes (nodes with two children): store key-value pairs

External trees

- Internal nodes: used only for routing purposes
- Key-value pairs are stored only in the leaves

Bottom-up trees

- An update operation has two phases
- 1st: a traversal with a top-down manner until the desired node is reached
- 2nd: rebalancing the tree with a bottom up manner

Top-down trees

- There is only a top down traversal
- Performing rebalances in advance (generally more tree modifications are performed)
- Created to simplify fine-grained locking approaches

System configuration

Huawei platform (60-core, NUMA architecture)

- 4 sockets (Intel(R) Xeon(R) CPU E7-4880 v2 @ 2.50GHz)
- 15 cores per socket (30 threads with hyperthreading)
- 32KB L1 data cache per core
- 32KB L1 instruction cache per core
- 256KB L2 cache per core
- 38MB L3 cache per socket
- 1TB RAM

A naive approach

Coarse-grained locking

- AVL external bottom-up tree
- RBT external bottom-up tree
- RBT internal bottom-up tree
- RBT external top-down tree
- RBT internal top-down tree

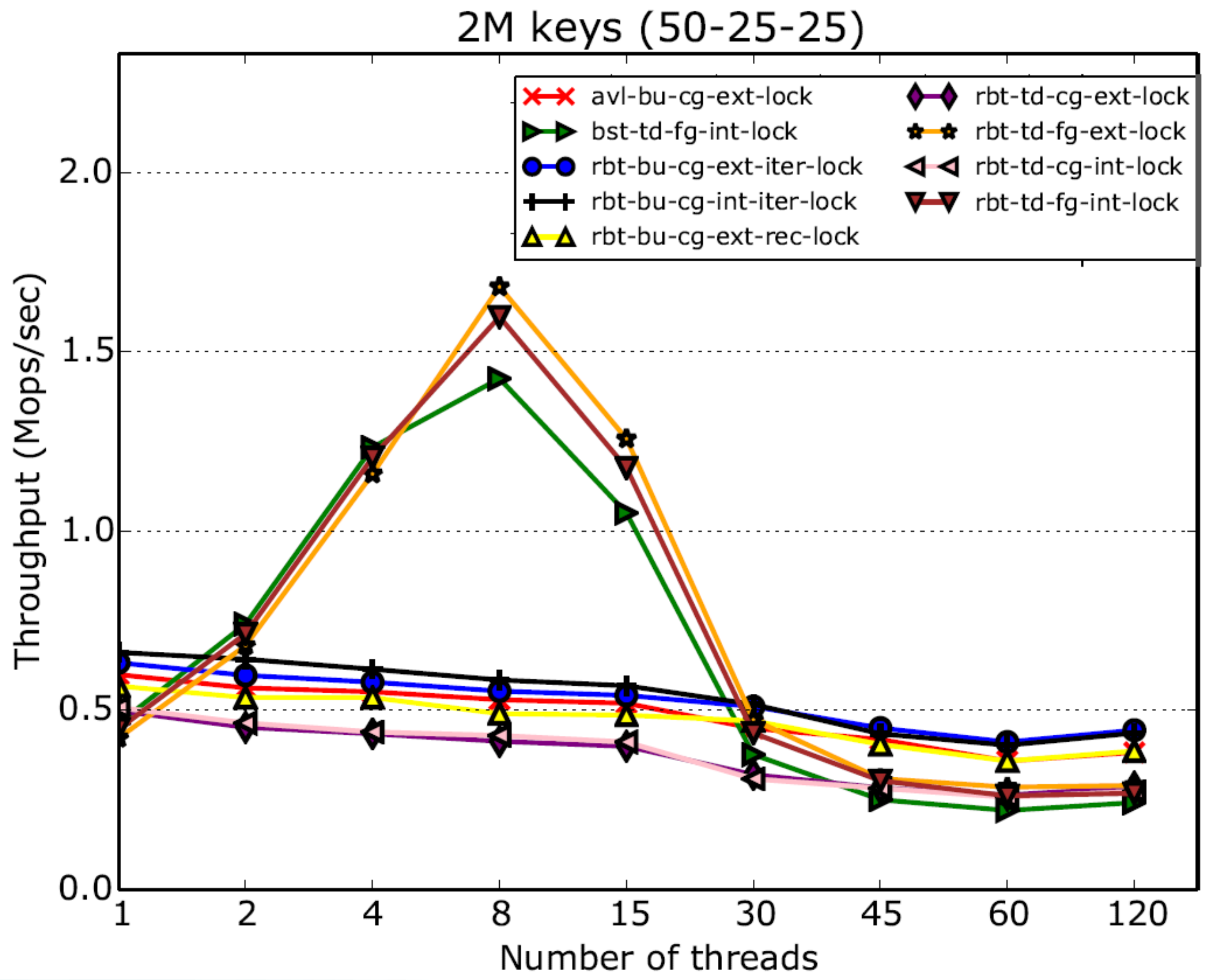
Fine-grained locking

- BST internal top-down tree
- RBT external top-down tree
- RBT internal top-down tree

Run configurations

- Key ranges: 2K, 32K, 2000000 keys
- Different workloads (lookup-insertion-deletion) : 80-10-10, 50-25-25, 20-40-40

Performance results



- Fine-grained locking scale up to 8 threads (poorly)
- 30 threads: NUMA effect more than coarse-grained implementations
- Coarse-grained locking provides no parallelism
- RBT external fine-grained: best throughput, since internal approach has to lock the whole subtree in deletion

A sophisticated approach

Bronson

- Lock-based
- Relaxed balanced AVL tree
  - ✓ Rebalances may be delayed
- Partial external tree
  - ✓ Leaving a deleted node when it has 2 children
- Version numbers to indicate if a write is in progress

Aravind

- Lock-free BST tree
- Atomic operations: CAS, bit-test-and-set (BST)
- Marking deleting edges instead of nodes
- Multiple keys can be removed in a single step

Draschler

- Lock-based
- BST internal tree
- Logical ordering
- Find successor in O(1) (succ pointer)

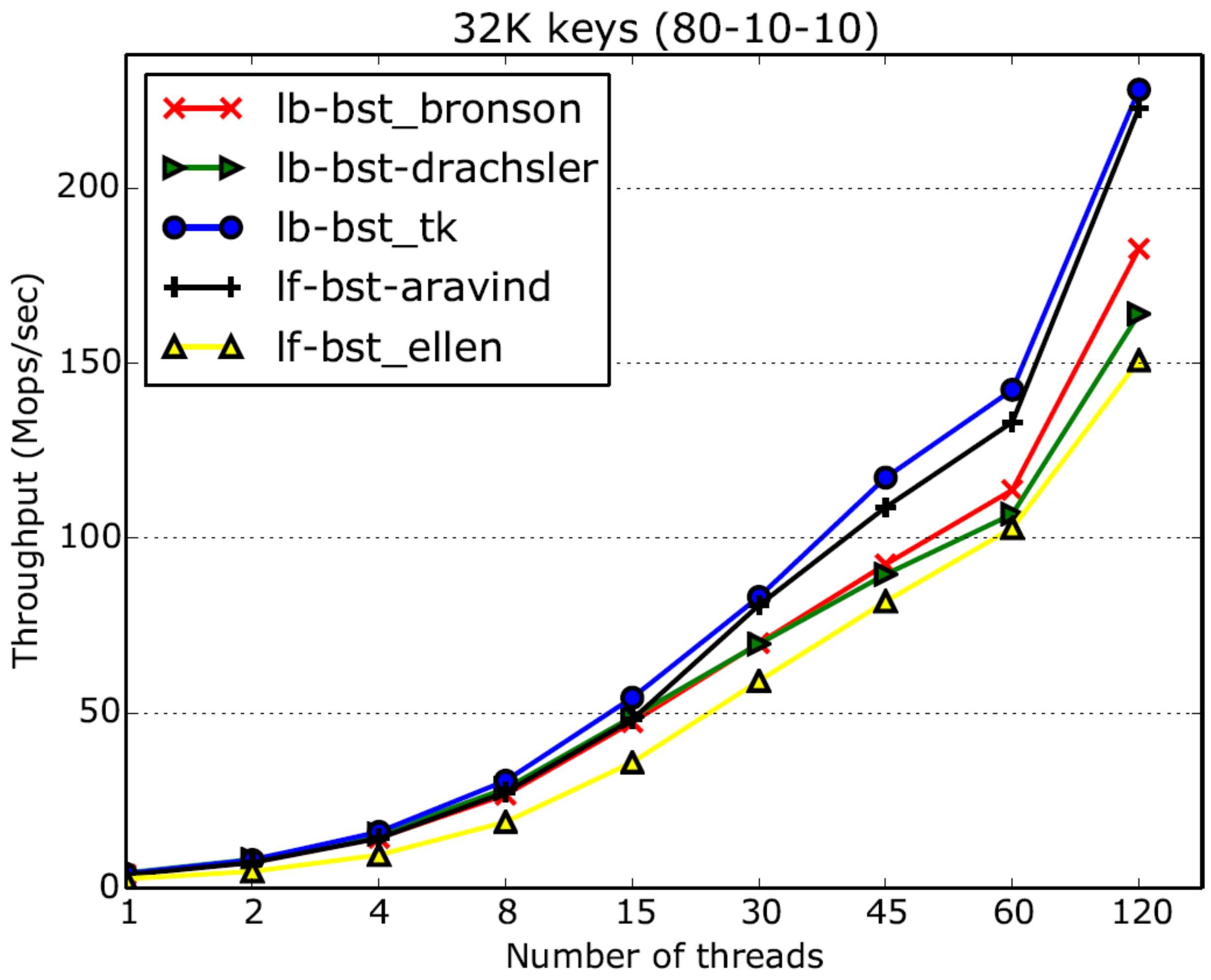
BST Ticket

- Lock-based external BST
- Version numbers to detect concurrent updates

Ellen

- Lock-free external BST tree
- Atomic operation: CAS
- Use explicit objects

Performance results



- BST Ticket, Aravind :better, Ellen, Draschler, Bronson: more complex synchronization
- Cache coherence: limiting factor, attempt to minimize the amount of cache traffic
- Lock-based and lock-free trees are close in terms of performance
- Lock-free: robustness, better applicable to oversubscriptions configurations
- Number of stores in a successful update should be close to an asynchronous, sequential algorithm

Conclusions

Naive lock-based implementations

- Very low throughput
- Easy to implement

More sophisticated implementations

- Really hard to implement
- Lock-based:
  - ✓ High performance
  - ✓ No robustness
- Lock-free:
  - ✓ High performance
  - ✓ Robustness

TRANSACTIONAL MEMORY

Transactional Memory

- a group of instructions is executed as an atomic transaction
- easy programming
- the underlying TM system executes the transaction in an atomic way
- the programmer defines the block of code to be executed as transaction
- If no conflicts are detected, the TM system persists the results (**transactional commit**)
- If conflicts are detected, the TM system rolls back the transaction. Modifications are discarded (**transactional abort**)

Three categories

- Software Transactional Memory (STM)
- Hardware Transactional Memory (HTM)
- Hybrid Transactional Memory (Hybrid TM)

Basic TM characteristics

- Data versioning
  - ✓ Eager versioning (undo-log to roll back the transaction)
  - ✓ Lazy versioning (write-buffer)
- Conflict detection
  - ✓ Pessimistic detection (detect conflicts early)
  - ✓ Optimistic detection (detect conflicts at commit time)
- Conflict resolution (resolution policy e.g. stall or abort the transaction)
- Isolation
  - ✓ Strong isolation (conflicts detected even if in a non-transactional code)
  - ✓ Weak isolation (transaction isolated only from other transactions
- Granularity (unit of storage over which TM system detects conflicts)
  - ✓ Object granularity
  - ✓ Word granularity
  - ✓ Cache line granularity
- Conflicts
  - ✓ Data conflicts
  - ✓ Capacity aborts
  - ✓ Explicit abort
  - ✓ Other (e.g system calls)
- Best effort
  - ✓ No forward progress is guaranteed
  - ✓ A non-transactional fallback path is necessary

Intel’s Haswell HTM

- HTM implementation (2013)

TSX Intel Interfaces

- Hardware Lock Elision (HLE)
  - ✓ two new instruction prefixes, used to denote the bounds of the critical section
  - ✓ run to hardware without TSX
- Restricted Transactional Memory (RTM)
  - ✓ flexibility to specify a fallback code path
  - ✓ does not run to hardware without TSX
  - ✓ 4 new instructions
    - ❑ XBEGIN
    - ❑ XEND
    - ❑ XTEST
    - ❑ XABORT

PARALLELIZING DIJKSTRA’S ALGORITHM

Dijkstra's algorithm

SSSP Problem

- Directed graph G=(V,E)
- Non-negative edges
- Source vertex s
- Observation: any subpath of any shortest path is itself a shortest path

Shortest path estimate d(v)

- Gradually converges to $\delta(v)$ through relaxations
- Relax(v,w): $d(w) = \min\{d(w), d(v) + w(v,w)\}$

  can we find a better path ?

Three partitions of vertices

- Settled: $d(v) = \delta(v)$
- Queued: $d(v) > \delta(v)$ and $d(v) \neq \infty$
- Unreached: $d(v) = \infty$

Dijkstra's algorithm

Serial algorithm

Input : $G = (V, E)$, $w : E \to \mathbf{R}^+$,
source vertex s, min Q
Output : shortest distance array d,
predecessor array π

foreach $v \in V$ **do**
    $d[v] \leftarrow \text{INF}$;
    $\pi[v] \leftarrow \text{NIL}$;
    `Insert`(Q, v);
end
$d[s] \leftarrow 0$;
while $Q \neq \emptyset$ **do**
    $u \leftarrow$ `ExtractMin`(Q);
    foreach *v adjacent to u* **do**
        $sum \leftarrow d[u] + w(u, v)$;
        if $d[v] > sum$ **then**
            `DecreaseKey`(Q, v, sum);
            $d[v] \leftarrow sum$;
            $\pi[v] \leftarrow u$;
    end
end

- Min d(v): min-priority queue
- Binary heap: $\Theta(m * \log n)$
- Fibonacci heap: $\Theta(m + n * \log n)$

Using binary heap

- Array implementation
- Maintain all but the settled vertices
- Min-heap property:
  $\forall\, i : d(\text{parent}(i)) \leq d(i)$

The four phases of the algorithm

Main thread

while $Q \neq \emptyset$ **do**

  $u \leftarrow \texttt{ExtractMin}(Q);$ timer 1

  $done \leftarrow 0;$

  foreach v *adjacent to* u **do**

    $sum \leftarrow d[u] + w(u, v);$ timer 2

    Begin-Xact

    if $d[v] > sum$ **then** timer 3

      $\texttt{DecreaseKey}(Q, v, sum);$

      $d[v] \leftarrow sum;$

      $\pi[v] \leftarrow u;$ timer 4

    End-Xact

  end

  Begin-Xact

  $done \leftarrow 1;$

  End-Xact

end

Timers:

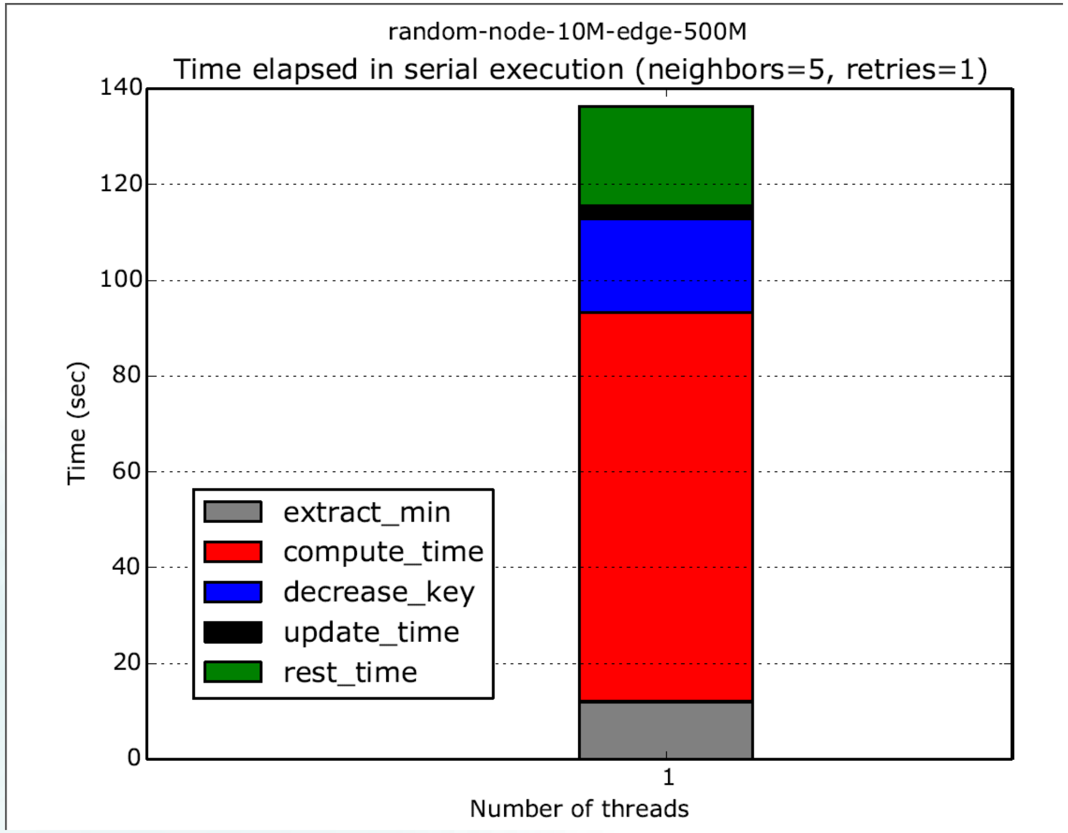
- Timer 1: ExtractMin()
- Timer 2: Compute an estimate for distance
- Timer 3: DecreaseKey()
- Timer 4: Update new distance (relax)

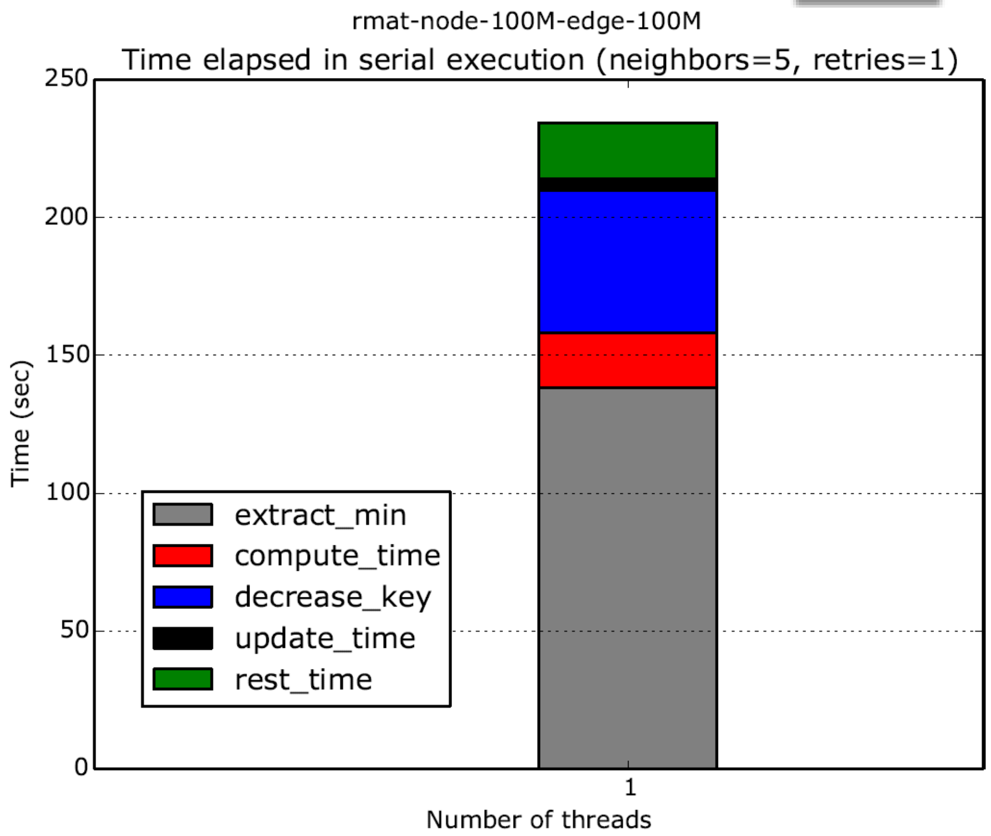
System configuration

Haci3 platform (28-core, NUMA architecture)

- 2 sockets (Intel(R) Xeon(R) CPU E5-2697 v3 @ 2.60GHz)
- 14 cores per socket (28 threads with hyperthreading)
- 32KB L1 data cache per core
- 32KB L1 instruction cache per core
- 256KB L2 cache per core
- 35MB L3 cache per socket
- 128GB RAM
- Hardware Transactional Memory:
  - ✓ lazy data versioning
  - ✓ eager conflict resolution
  - ✓ best effort HTM
  - ✓ strong isolation
  - ✓ cache line granularity
  - ✓ 4MB read set
  - ✓ 22KB write set

Experimentation in the serial algorihtm





- Four phases: ExtractMin(), compute distance, DecreaseKey(), update distance
- Parallel part: very small percentage of the total runtime
- Can not extract much parallelism
- Dijkstra's algorithm: inherently serial algorithm

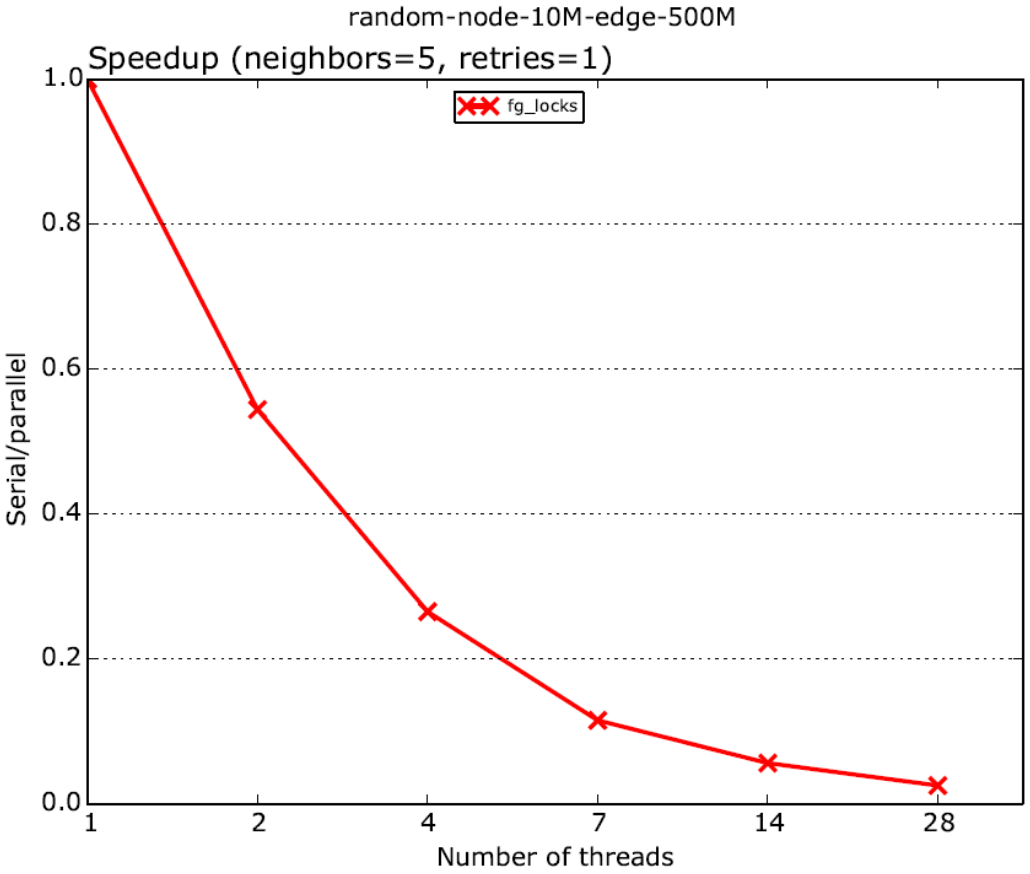
A naïve parallelization

algorithm

Input : $G = (V, E)$, $w : E \to \mathbf{R}^+$, source vertex s, min Q

Output : shortest distance array d, predecessor array π

foreach $v \in V$ **do**
  $d[v] \leftarrow$ INF;
  $\pi[v] \leftarrow$ NIL;
  Insert(Q, v);
end
$d[s] \leftarrow 0$;
while $Q \neq \emptyset$ **do**
  if (id == 0)
    $u \leftarrow$ ExtractMin(Q);
  BARRIER
  Parallel For
  foreach *v adjacent to u* **do**
    $sum \leftarrow d[u] + w(u, v)$;
    if $d[v] > sum$ **then**
/* Fine-grained*/ DecreaseKey(Q, v, sum);
      $d[v] \leftarrow sum$;
      $\pi[v] \leftarrow u$;
  end
end

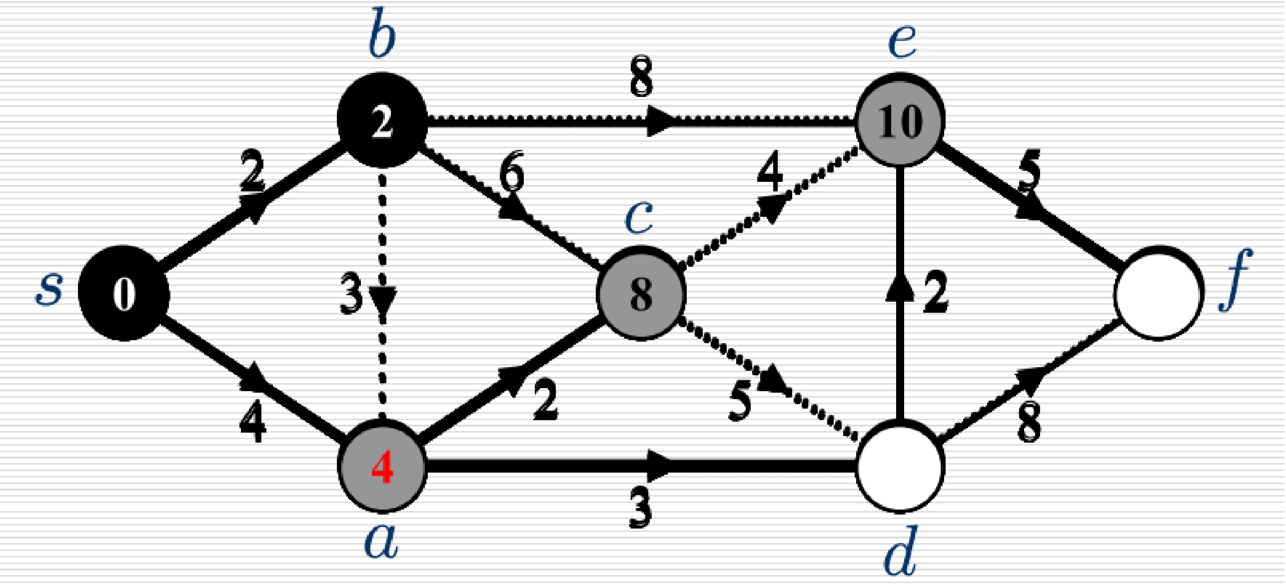


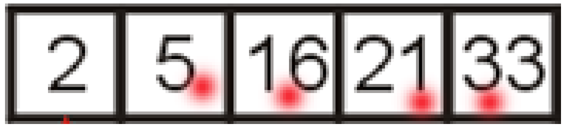
Thread	Time (sec)
1	78
2	145
4	298
7	687
14	1422
28	3184

Parallelizing Dijkstra’s algorithm

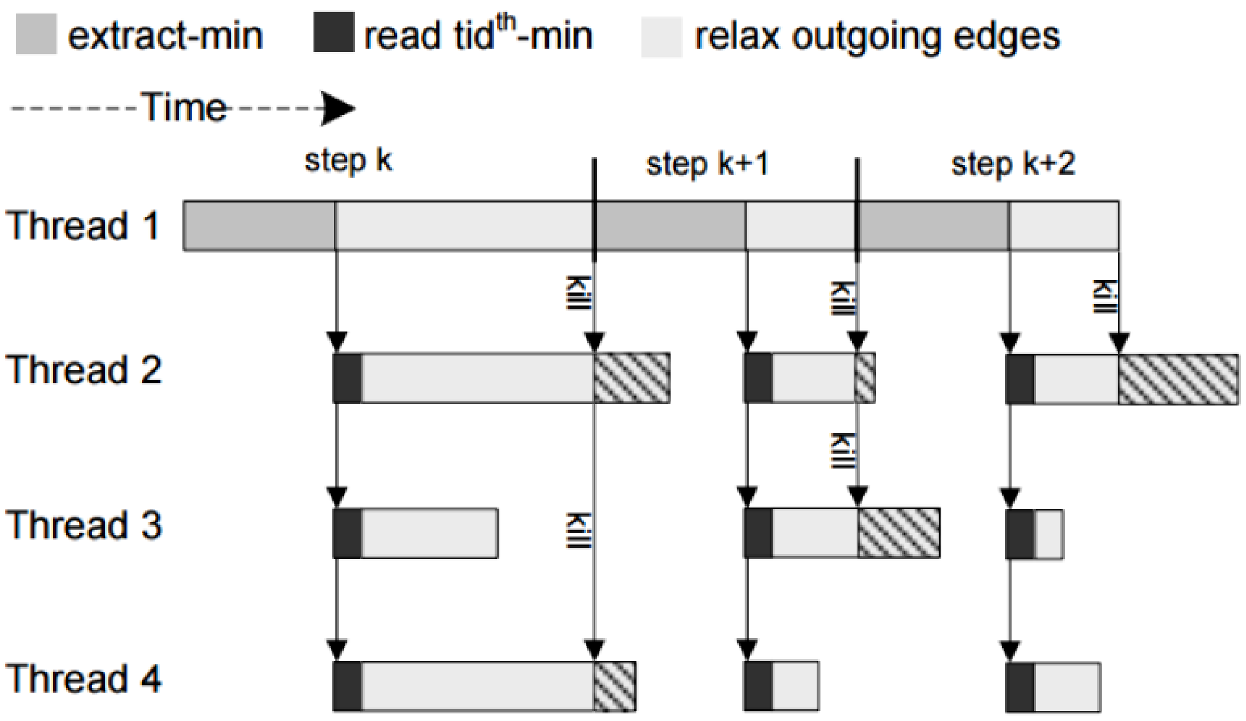
The idea

- In serial: relaxations are performed only from the extracted (settled) vertex
- Allow relaxations for the queued vertices (some of them might have obtained the optimal value)
  - ✓ If the vertex is settled, helper thread will offload work from the main thread
  - ✓ If the vertex has not obtained its optimal value, the updated (suboptimal) distances will be corrected by the main thread





Execution pattern



- main thread extracts the min vertex and signals helper threads
- n helper threads read the next n vertices in the queue
- main thread stops all helper threads at the end of each iteration
- unfinished will be corrected in next steps

Parallelizing Dijkstra's algorithm

Main thread

while $Q \neq \emptyset$ **do**
  $u \leftarrow \texttt{ExtractMin}(Q)$;
  $done \leftarrow 0$;
  foreach *v adjacent to u* **do**
    $sum \leftarrow d[u] + w(u, v)$;
    Begin-Xact
    if $d[v] > sum$ **then**
      $\texttt{DecreaseKey}(Q, v, sum)$;
      $d[v] \leftarrow sum$;
      $\pi[v] \leftarrow u$;
    End-Xact
  end
  Begin-Xact
  $done \leftarrow 1$;
  End-Xact
end

Helper thread

while $Q \neq \emptyset$ **do**
  while $done = 1$ **do** ;
  $x \leftarrow \texttt{ReadMin}(Q, tid)$
  $stop \leftarrow 0$
  foreach *y adjacent to x* **and while** $stop = 0$ **do**
    Begin-Xact
    if $done = 0$ then
      $sum \leftarrow d[x] + w(x, y)$
      if $d[y] > sum$ **then**
        $\texttt{DecreaseKey}(Q, y, sum)$
        $d[y] \leftarrow sum$
        $\pi[y] \leftarrow x$
    else
      $stop \leftarrow 1$
    End-Xact
  end
end

- Enclose relaxation within a transaction

Parallelizing Dijkstra's algorithm

Main thread

```
while Q ≠ ∅ do
    u ← ExtractMin(Q);
    done ← 0;
    foreach v adjacent to u do
        sum ← d[u] + w(u, v);
        Begin-Xact
        if d[v] > sum then
            DecreaseKey(Q, v, sum);
            d[v] ← sum;
            π[v] ← u;
        End-Xact
    end
    Begin-Xact
    done ← 1;
    End-Xact
end
```

Optimizations :

- Strong isolation HTM: "done" variable not in transactional mode
- Cache line granularity in HTM: structure padding in binary heap's vertices
- Coarse-grained transactions: more relaxations within a single transaction (less overhead)
- Best effort HTM: in fall back path helper threads can never obtain the global lock such that no to interfere in main thread's progress.

Experimentation in the parallel algorihtm

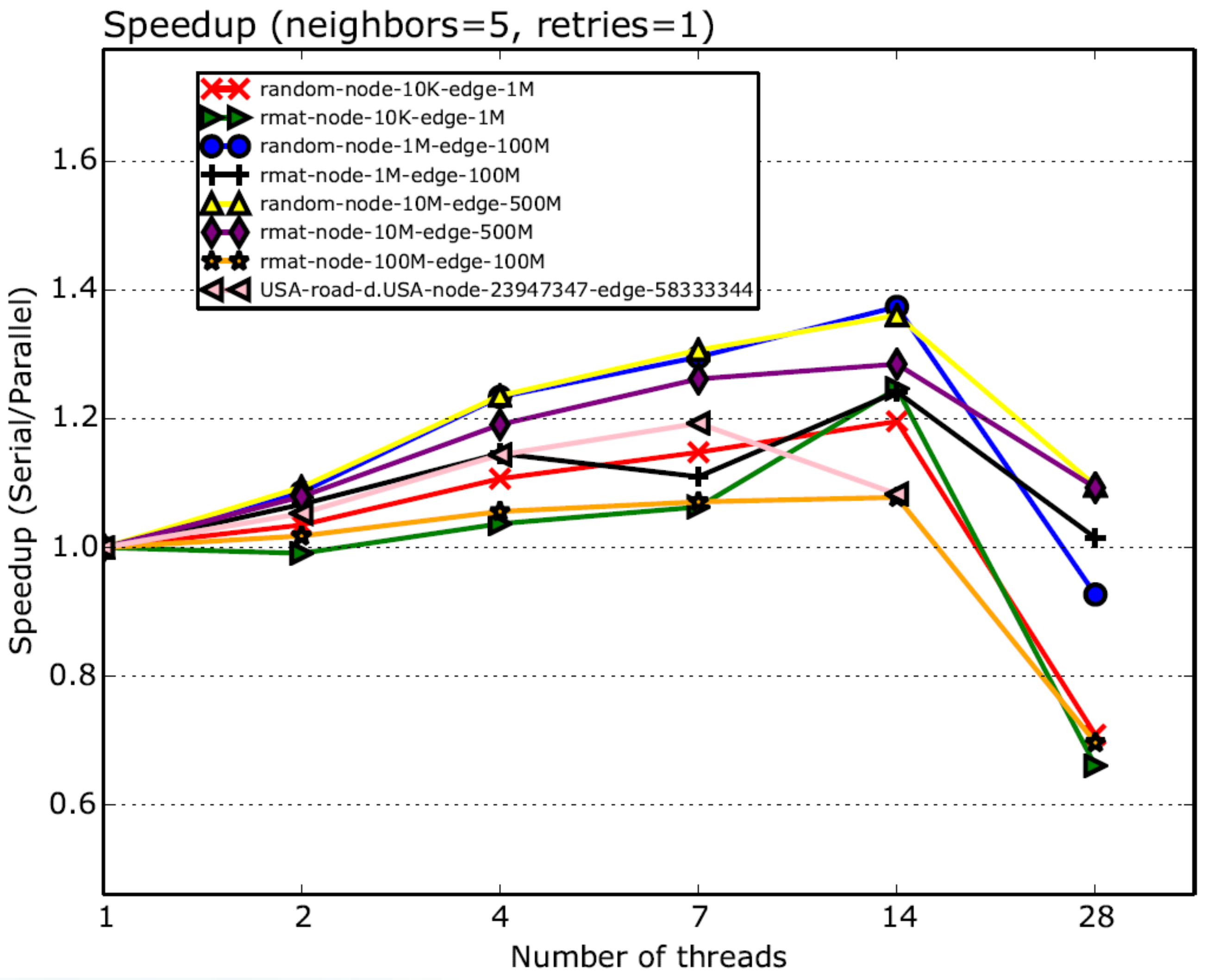
Number of retries in transactional mode before acquiring the global lock

- Main thread: 1 retry (goal: run as in the serial execution)
- Helper threads: can never acquire the global lock (always executing in transactional mode)

Number of edges examined within a single transaction (more coarse-grained)

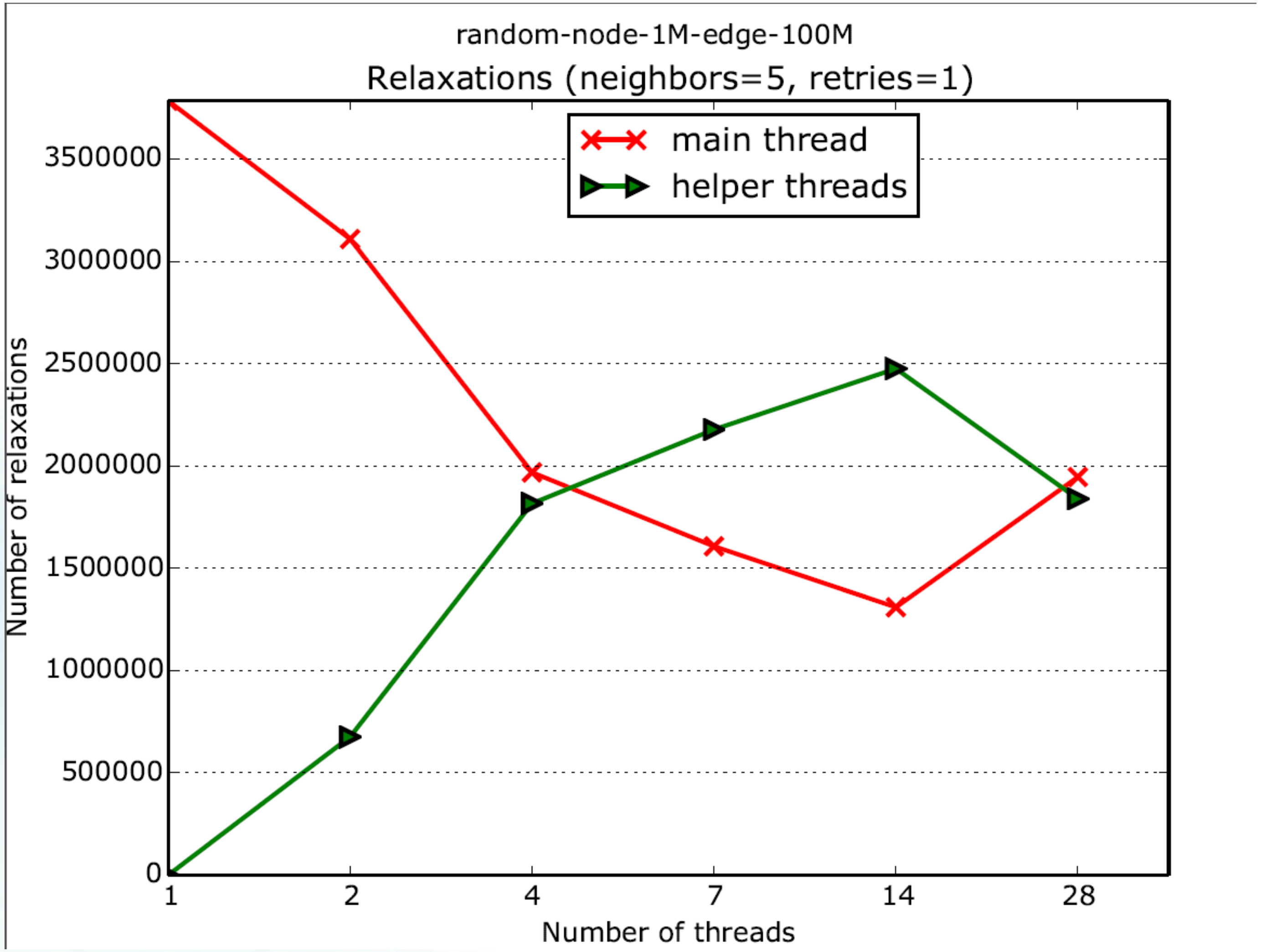
- Main thread: 5, 10 edges examined within a single transaction. A very coarse-grained transaction leads to capacity aborts, especially in large graphs.
- Helper threads: 1 edge per transaction (Helper threads have to perform many, small relaxations before they are stopped by the main thread).

Performance results



- Performance is strongly related to the density of the graph
- ✓ in dense graphs => more parallelism can be exposed
- ✓ sparse graphs => limited parallelism
- 28 threads: NUMA effect
- ✓ expensive cache line transfers from one socket to another => negatively influence scalability
- ✓ Future Work: redesign algorithm such that to scale even in the presence of non-uniformity.

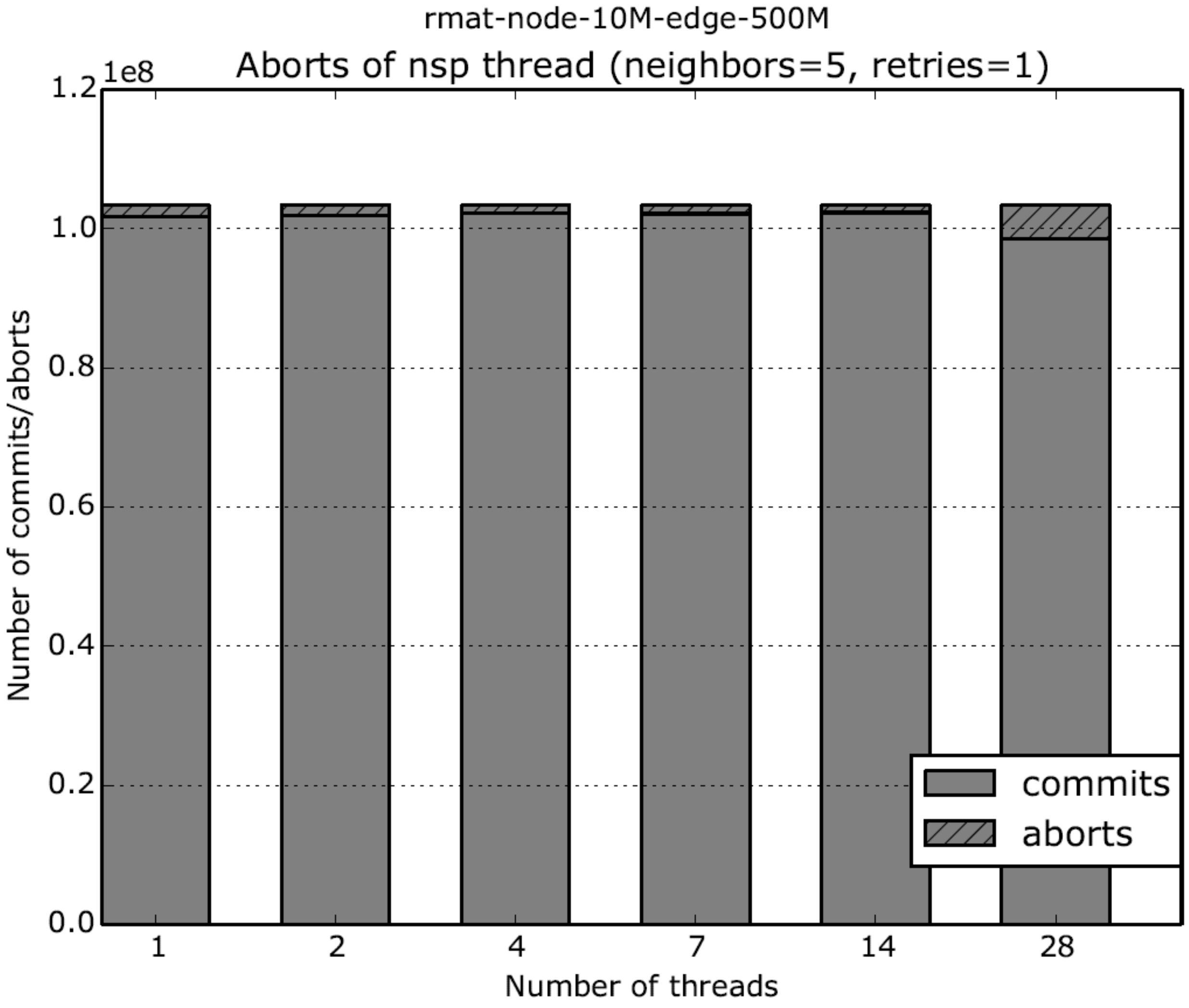
A closer look at the results



Relaxations

- Number of threads increases => fewer relaxations (main thread)
  => scalability
- Gain in relaxations => we believe that the algorithm can give much better performance
  - ✓ time spent in thread synchronization
  - ✓ delays due to conflicting transactions
- 28 threads: NUMA effect

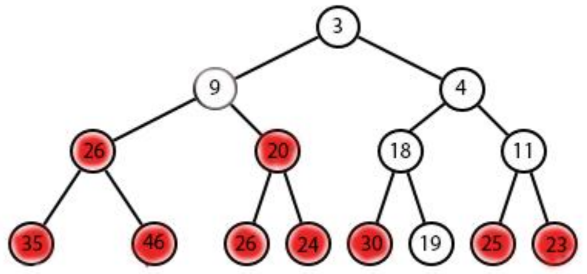
A closer look at the results

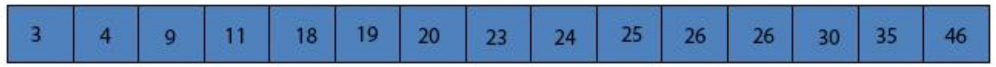


Aborts

- Main thread: suffers a really low number of aborts
- ✓ Helper threads do not obstruct its progress even not contributing to useful work
- Addition of more threads => do not increase the number of aborts
- ✓ If NUMA effect did not exist => better scalability for more helper threads

Employing skip list

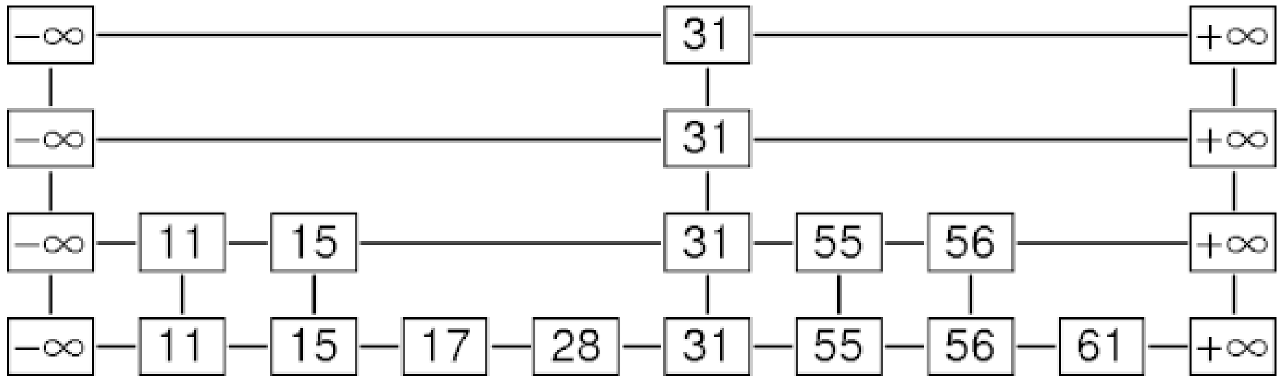




Characteristics

- Keys in sorted order
- It is build in O(logn) layers
- Sentinel elements (head, tail): in all layers
- Internal elements: random height between 1 and levelmax

Skip list in Dijkstra’s algorithm



- Serial: ExtractMin() in O(1)
- DecreaseKey() in O(logn)
- Parallel: Threads act more locally, write only the previous and the next element, while using binary heap has bigger contention window, especially when proceeding higher in the heap.
- A more concurrently friendly structure

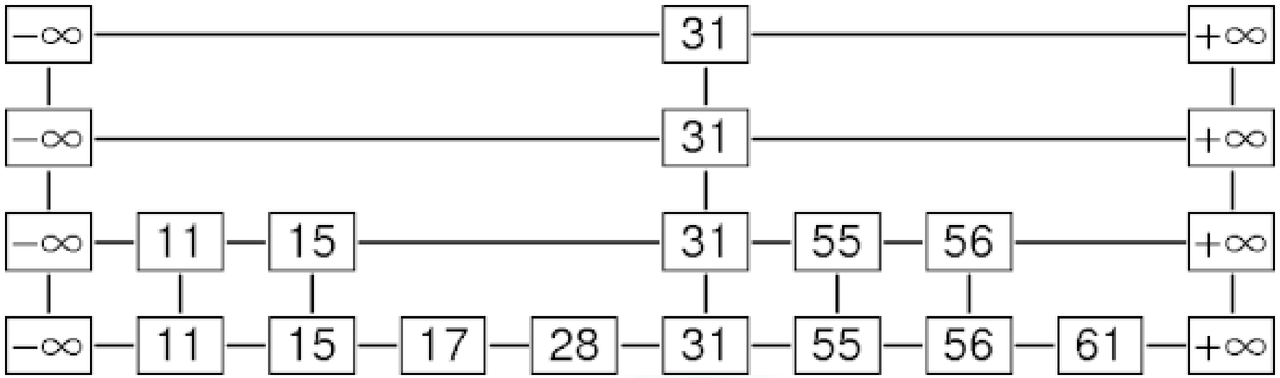
Employing skip list

During execution

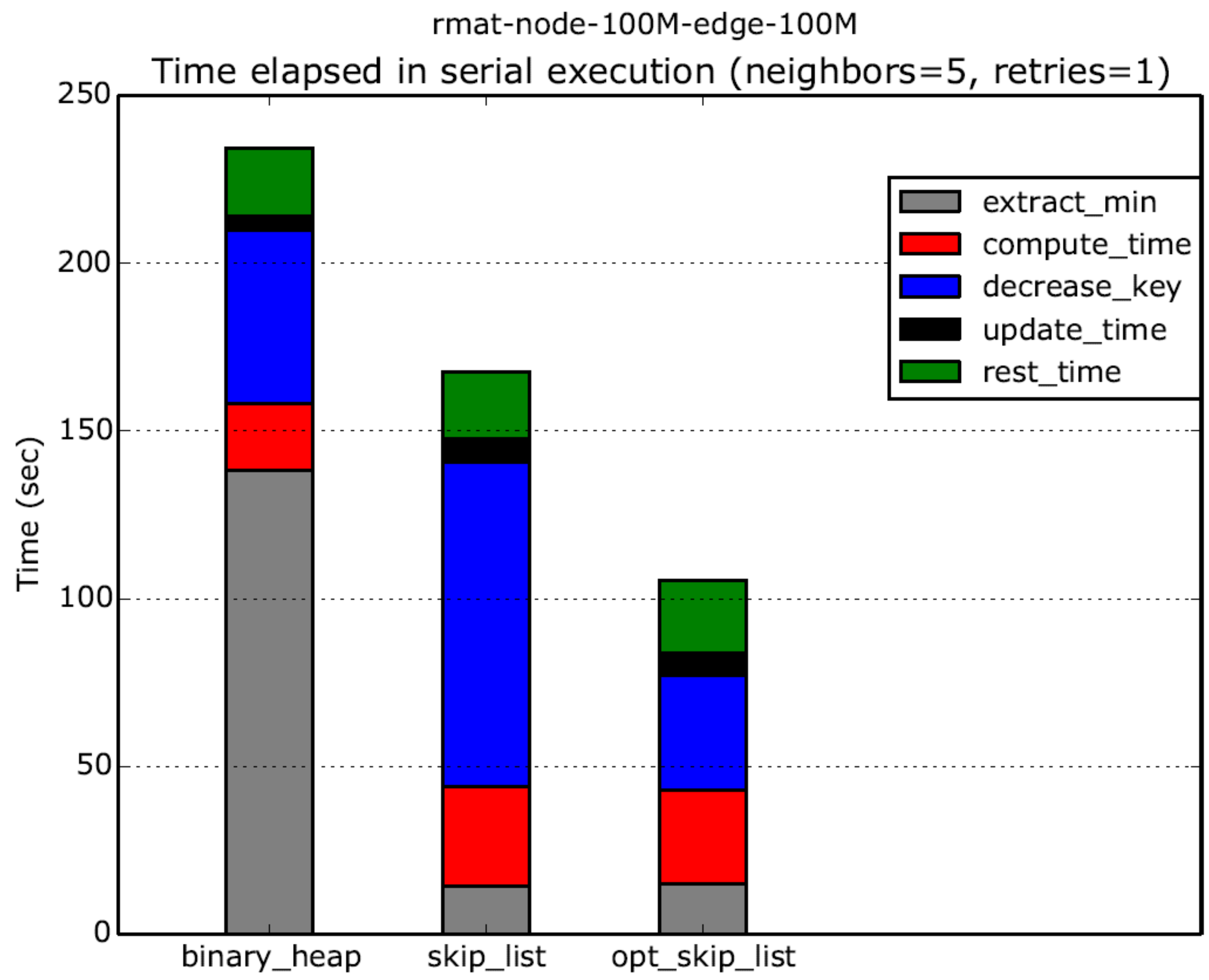
- Simple skip list: an element for each vertex of the graph (much memory)
- Time consuming traversal (DecreseKey() => too many steps)
- Too many elements with the same key (same distance from source)
- Execution time: extremely large
- Low performance

Solution => an optimized skip list

- Contains only elements with discrete keys
- Internal nested list: stores the ids of the vertices that have the same distance (key) from the source
- Less memory
- Faster traversal



Comparing the serial execution

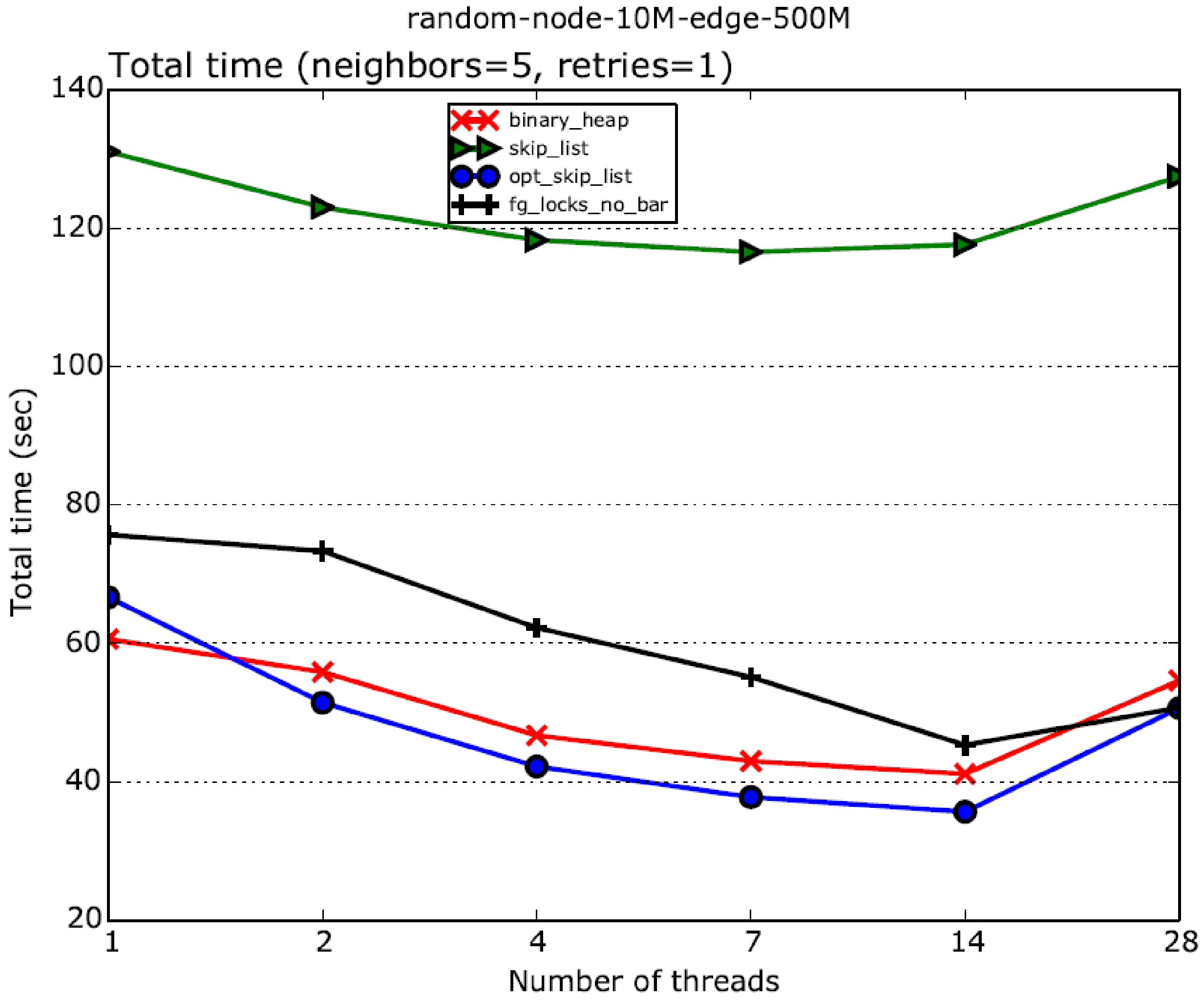


Total time

- Skip list: less time consuming ExtractMin() (O(1))
- DecraseKey(): too many steps in the simple skip list

✓ Many elements with the same key to be overtaken during traversal

Comparing the parallel execution



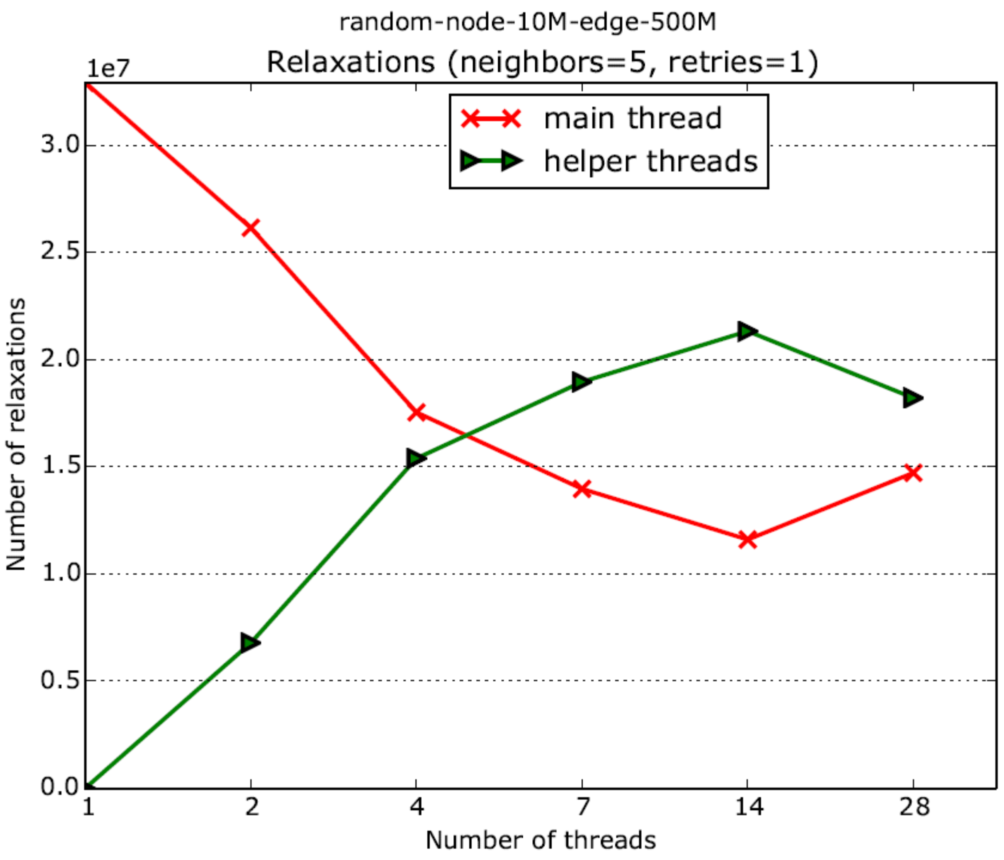
Total time

- Simple skip list: DecreaseKey() time consuming
- Optimized skip list: better scalability
- ✓ Helper threads act more locally and perform more relaxations
- ✓ Binary heap and opt skip list execution have comparable transactional aborts
- 28 threads: NUMA effect

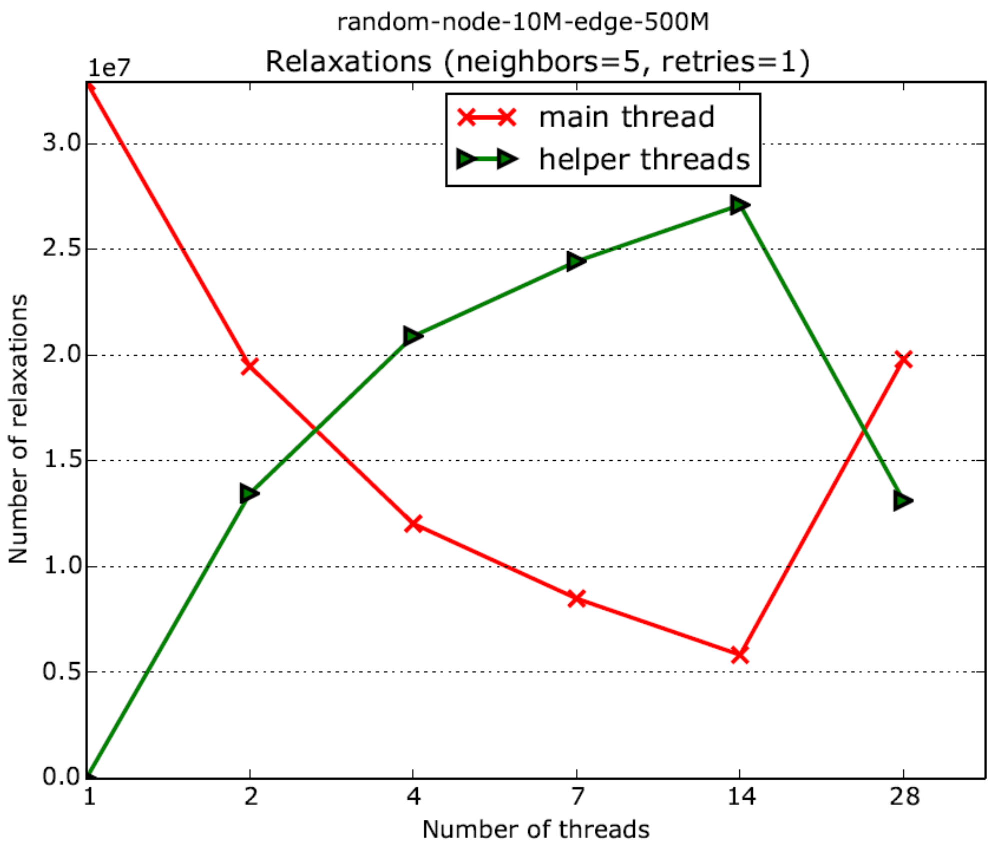
* Fine-grained_no_bar: Fine-grained locking in Decreasekey(), that uses barriers for synchronization. We have removed the time spent in barriers to use it as a baseline.

Relaxations performed

Binary heap

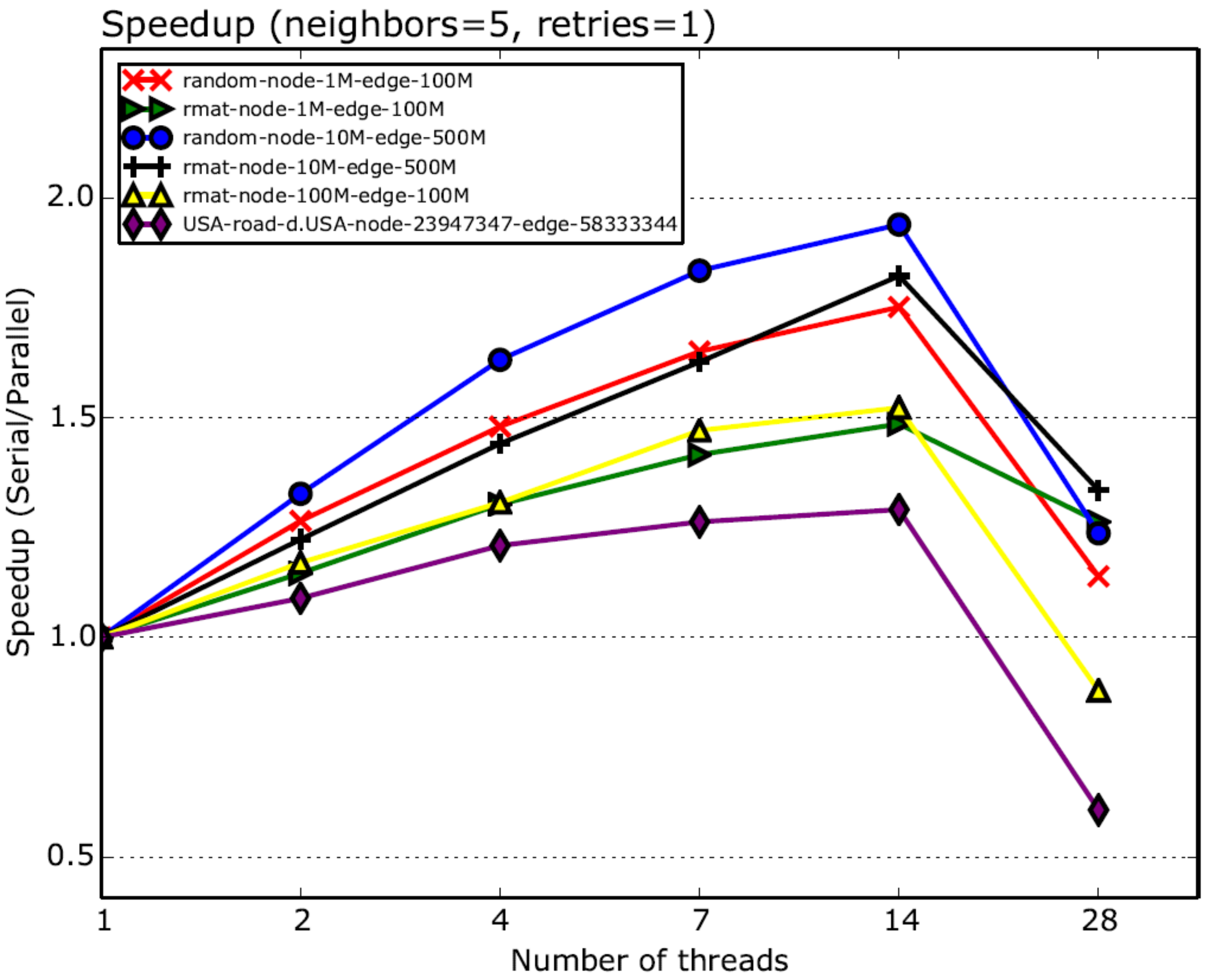


Skip list



- ✓ Binary heap and opt skip list execution have comparable transactional aborts

Performance results



Speedup

- Speedup related to the density of the graph
- Better scalability
- ✓ Helper threads offload more work from the main thread
- Real USA network => there is also a very good performance (estimates for distances have the same values, during the execution of the algorithm, with some strong probability in real networks, too.

- Bin_heap vs Skip_list: comparable transactional aborts, but in the skip list execution, helper threads perform more relaxations (offload more work).

A speedup approximation (14 threads)

Binary heap

Graph	Ideal Speedup	Speedup achieved
random-node-1M-edge-100M	4	1.45
rmat-node-1M-edge-100M	3.58	1.27
random-node-10M-edge-500M	3.74	1.46
rmat-node-10M-edge-500M	3.09	1.35
rmat-node-100M-edge-100M	1.22	1.1
USA-road-node-23M-edge-58M	1.91	1.08

Skip list

Graph	Ideal Speedup	Speedup achieved
random-node-1M-edge-100M	4.57	1.75
rmat-node-1M-edge-100M	5.11	1.49
random-node-10M-edge-500M	5.14	1.94
rmat-node-10M-edge-500M	4.74	1.82
rmat-node-100M-edge-100M	1.79	1.52
USA-road-node-23M-edge-58M	1.48	1.29

$$s = \frac{1+d}{1+a*d}$$

- a: is the ratio of the main thread's DecreaseKey() operations to those executed in the serial case
- d: the average out-degree of the vertices.

FUTURE WORK

Future Work

Concurrent Binary Search Trees

- Evaluate other lock-based and lock free implementations
- Use of transactional memory as sychronization mechanism in BSTs
- Evaluate other structures like FIFO queues, Hash tables, priority queues

Dijkstra's algorithm

- Examine levelmax of our optimized skip list
- Evaluate the algorithm to another HTM implementation with different resolution policy
- Explore adaptive schemes the number of helper threads will be dynamically adjusted
- Opt_skip_list: extract multiple vertices with the same (min) distance in a single step

QUESTIONS ???