A Distributed Hash Table for Shared Memory

Wytse Oortwijn

Formal Methods and Tools,
University of Twente

August 31, 2015

Table of Contents

Table of Contents

Distributed Hash Table

Main challenge

Building a *fast* and *CPU-efficient* shared hash table:

- Minimal latency
- Minimal memory overhead
- Not relying on CPU-polling

Distributed Hash Table

Main challenge

Building a *fast* and *CPU-efficient* shared hash table:

- Minimal latency
- Minimal memory overhead
- Not relying on CPU-polling

Many use cases in HPC

- Parallel graph searching
- *Distributed model checking*

Distributed Hash Table

Main challenge

Building a *fast* and *CPU-efficient* shared hash table:

- Minimal latency
- Minimal memory overhead
- Not relying on CPU-polling

Many use cases in HPC

- Parallel graph searching
- *Distributed model checking*

Distributed *(LAN)* vs Parallel

- *Cheaper* scalability
- *Unlimited* scalability, *but*

Distributed Hash Table

Main challenge

Building a *fast* and *CPU-efficient* shared hash table:

- Minimal latency
- Minimal memory overhead
- Not relying on CPU-polling

Many use cases in HPC

- Parallel graph searching
- *Distributed model checking*

Distributed *(LAN)* vs Parallel

- *Cheaper* scalability
- *Unlimited* scalability, *but*
- Performance overhead!

Distributed Hash Table

Main challenge

Building a *fast* and *CPU-efficient* shared hash table:

- Minimal latency
- Minimal memory overhead
- Not relying on CPU-polling

Many use cases in HPC

- Parallel graph searching
- *Distributed model checking*

Distributed *(LAN)* vs Parallel

- *Cheaper* scalability
- *Unlimited* scalability, *but*
- Performance overhead!

Efficient distributed processing

Specialized algorithms and data structures needed!

Distributed Hash Table

Main challenge

Building a *fast* and *CPU-efficient* shared hash table:

- Minimal latency
- Minimal memory overhead
- Not relying on CPU-polling

Many use cases in HPC

- Parallel graph searching
- *Distributed model checking*

Distributed *(LAN)* vs Parallel

- *Cheaper* scalability
- *Unlimited* scalability, *but*
- Performance overhead!

Efficient distributed processing

Specialized algorithms and data structures needed!

- **Contribution**: Reducing roundtrips *while* CPU-efficient

High-performance Networking

Infiniband hardware

Specialized hardware used to construct high-performance networks:

- Comparable in price to Ethernet
- Supports bandwidths up to 100 Gb/s
- *Direct* access to memory via PCI-E bus

High-performance Networking

Infiniband hardware

Specialized hardware used to construct high-performance networks:

- Comparable in price to Ethernet
- Supports bandwidths up to 100 Gb/s
- *Direct* access to memory via PCI-E bus

RDMA: Remote Direct Memory Access

Directly access to remote memory *without* invoking remote CPUs

- Zero-copy networking
- Kernel bypassing
- No participation from remote CPUs

High-performance Networking

Infiniband hardware

Specialized hardware used to construct high-performance networks:

- Comparable in price to Ethernet
- Supports bandwidths up to 100 Gb/s
- *Direct* access to memory via PCI-E bus

RDMA: Remote Direct Memory Access

Directly access to remote memory *without* invoking remote CPUs

- Zero-copy networking
- Kernel bypassing
- No participation from remote CPUs

Performance: *one-sided* RDMA vs TCP

Roundtrips latency: $< 3\mu s$ (Infiniband) vs $60\mu s$ (traditional Ethernet)

Hash Table: Challenges

Notation: Hash table

$T = \langle b_0, \ldots, b_{n-1} \rangle$ as a sequence of buckets b_i, where:

- n the hash table size and m the number of used entries
- $\alpha = \frac{m}{n}$ the load factor

Hash Table: Challenges

Notation: Hash table

$T = \langle b_0, \ldots, b_{n-1} \rangle$ as a sequence of buckets b_i, where:

- n the hash table size and m the number of used entries
- $\alpha = \frac{m}{n}$ the load factor

Operation: *only* `find-or-put`(d)

Takes a data element d as parameter, and:

- if $d \in T$, return `found`
- if $d \notin T$, insert d and return `inserted`
- if $d \notin T$ and d cannot be inserted, return `full`

Hash Table: Challenges

Notation: Hash table

$T = \langle b_0, \ldots, b_{n-1} \rangle$ as a sequence of buckets b_i, where:

- n the hash table size and m the number of used entries
- $\alpha = \frac{m}{n}$ the load factor

Operation: *only* `find-or-put`(d)

Takes a data element d as parameter, and:

- if $d \in T$, return `found`
- if $d \notin T$, insert d and return `inserted`
- if $d \notin T$ and d cannot be inserted, return `full`

Design: Challenges

- How to *distribute* and access $T = \langle b_0, \ldots, b_{n-1} \rangle$ efficiently?
- How to *design* `find-or-put` to perform efficiently?

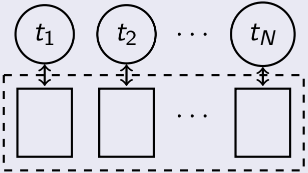
PGAS: Partitioned Global Address Space

Details

Assuming N participating threads:

PGAS: Partitioned Global Address Space

PGAS



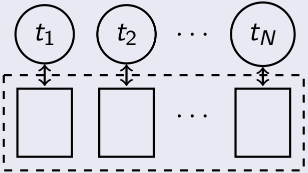
Details

Assuming N participating threads:

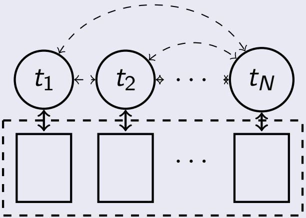
- **PGAS:** shared + distributed memory model

PGAS: Partitioned Global Address Space

PGAS



Hybrid PGAS



Details

Assuming N participating threads:

- **PGAS:** shared + distributed memory model
- **Hybrid PGAS:** PGAS + message passing *(dashed edges)*

Table of Contents

Hash Collisions

Efficiency: Resolving Hash Collisions

Occurs when $h(x) = h(y)$ for data elements $x \neq y$

- Efficiency of `find-or-put` depends on hashing strategy!

Hash Collisions

Efficiency: Resolving Hash Collisions

Occurs when $h(x) = h(y)$ for data elements $x \neq y$

- Efficiency of `find-or-put` depends on hashing strategy!

Existing Work

- Pilaf, 2014 *(Cuckoo)*
- Nessie, 2014 *(Cuckoo)*
- FaRM, 2014 *(Hopscotch)*
- HERD, 2014 *(CPU-intensive)*

Hash Collisions

Efficiency: Resolving Hash Collisions

Occurs when $h(x) = h(y)$ for data elements $x \neq y$

- Efficiency of `find-or-put` depends on hashing strategy!

Existing Work

- Pilaf, 2014 *(Cuckoo)*
- Nessie, 2014 *(Cuckoo)*
- FaRM, 2014 *(Hopscotch)*
- HERD, 2014 *(CPU-intensive)*

Contributions

Existing implementations either:

- Require *more* roundtrips
- Require *locking* schemes
- Are CPU-*intensive*

Hash Collisions

Efficiency: Resolving Hash Collisions

Occurs when $h(x) = h(y)$ for data elements $x \neq y$

- Efficiency of `find-or-put` depends on hashing strategy!

Existing Work

- Pilaf, 2014 *(Cuckoo)*
- Nessie, 2014 *(Cuckoo)*
- FaRM, 2014 *(Hopscotch)*
- HERD, 2014 *(CPU-intensive)*

Contributions

Existing implementations either:

- Require *more* roundtrips
- Require *locking* schemes
- Are CPU-*intensive*

Best strategy for `find-or-put`

Which strategy requires the *least* number of roundtrips?

Comparing Hashing Strategies

Chained Hashing

- \+ Theoretical comp. $\Theta(1+\alpha)$
- \- Dynamic mem. management
- \- Storing pointers

Comparing Hashing Strategies

Chained Hashing

- \+ Theoretical comp. $\Theta(1 + \alpha)$
- \- Dynamic mem. management
- \- Storing pointers

Cuckoo Hashing

- \+ Uses k hash functions
- \- Lookups require k roundtrips
- \- Relocations require locks

Comparing Hashing Strategies

Chained Hashing

+ Theoretical comp. $\Theta(1 + \alpha)$
- Dynamic mem. management
- Storing pointers

Cuckoo Hashing

+ Uses k hash functions
- Lookups require k roundtrips
- Relocations require locks

Hopscotch Hashing

+ Using neighbourhoods
+ Lookups require 1 roundtrip
- Relocations require locks

Comparing Hashing Strategies

Chained Hashing

- \+ Theoretical comp. $\Theta(1 + \alpha)$
- \- Dynamic mem. management
- \- Storing pointers

Cuckoo Hashing

- \+ Uses k hash functions
- \- Lookups require k roundtrips
- \- Relocations require locks

Hopscotch Hashing

- \+ Using neighbourhoods
- \+ Lookups require 1 roundtrip
- \- Relocations require locks

Linear Probing

- \+ Buckets are consecutive
- \+ *No* locking or relocations
- \- Roundtrips for lookups?

Comparing Hashing Strategies

Chained Hashing

- \+ Theoretical comp. $\Theta(1 + \alpha)$
- \- Dynamic mem. management
- \- Storing pointers

Cuckoo Hashing

- \+ Uses k hash functions
- \- Lookups require k roundtrips
- \- Relocations require locks

Hopscotch Hashing

- \+ Using neighbourhoods
- \+ Lookups require 1 roundtrip
- \- Relocations require locks

Linear Probing

- \+ Buckets are consecutive
- \+ *No* locking or relocations
- \- Roundtrips for lookups?

Linear Probing versus Hopscotch

- Due to Hopscotch invariant, lookups *may* be more expensive, but
- Inserts are arguably cheaper *(amortized complexity)*

Linear Probing: Efficiency Bounds

Knuth, 1997

The *expected* number of buckets to examine until the intended buckets is found is *at most*:

$$\frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$$

Linear Probing: Efficiency Bounds

Knuth, 1997

The *expected* number of buckets to examine until the intended buckets is found is *at most*:

$$\frac{1}{2C}\left(1 + \frac{1}{(1-\alpha)^2}\right)$$

Linear Probing: Efficiency Bounds

Knuth, 1997

The *expected* number of buckets to examine until the intended buckets is found is *at most*:

$$\frac{1}{2C}\left(1 + \frac{1}{(1-\alpha)^2}\right)$$

Efficiency bound

A chunk is *expected* to contain the intended bucket if:

$$\alpha \leq 1 - \sqrt{\frac{1}{2C-1}}$$

Linear Probing: Efficiency Bounds

Knuth, 1997

The *expected* number of buckets to examine until the intended buckets is found is *at most*:

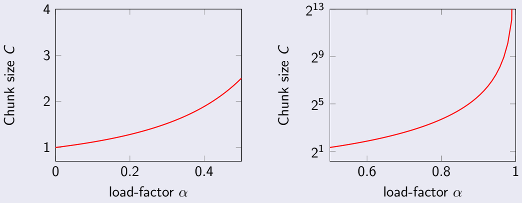
$$\frac{1}{2C}\left(1+\frac{1}{(1-\alpha)^2}\right)$$

Efficiency bound

A chunk is *expected* to contain the intended bucket if:

$$\alpha \leq 1-\sqrt{\frac{1}{2C-1}}$$

Expected load-factor at which a chunk is full



Linear Probing: Efficiency Bounds

Knuth, 1997

The *expected* number of buckets to examine until the intended buckets is found is *at most*:

$$\frac{1}{2C}\left(1 + \frac{1}{(1-\alpha)^2}\right)$$

Efficiency bound

A chunk is *expected* to contain the intended bucket if:

$$\alpha \leq 1 - \sqrt{\frac{1}{2C-1}}$$

Expected load-factor at which a chunk is full

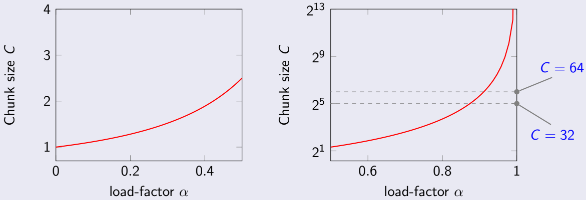


Table of Contents

Linear Probing: Hiding Latency

Contribution: Asynchronous queries

Before chunk iteration, first request the *next* chunk:

- Overlapping roundtrips with computational activity
- Find next chunk with *quadratic probing* to prevent clustering

Linear Probing: Hiding Latency

Contribution: Asynchronous queries

Before chunk iteration, first request the *next* chunk:

- Overlapping roundtrips with computational activity
- Find next chunk with *quadratic probing* to prevent clustering

Defining `query-chunk`(i, d)

Obtains the i-th chunk, starting from bucket $b_{h(d)}$

- Returns a handle s

Linear Probing: Hiding Latency

Contribution: Asynchronous queries

Before chunk iteration, first request the *next* chunk:

- Overlapping roundtrips with computational activity
- Find next chunk with *quadratic probing* to prevent clustering

Defining `query-chunk`(i, d)

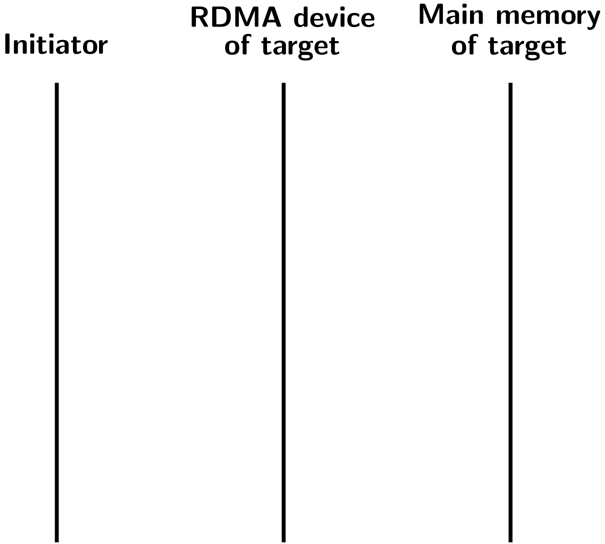
Obtains the i-th chunk, starting from bucket $b_{h(d)}$

- Returns a handle s

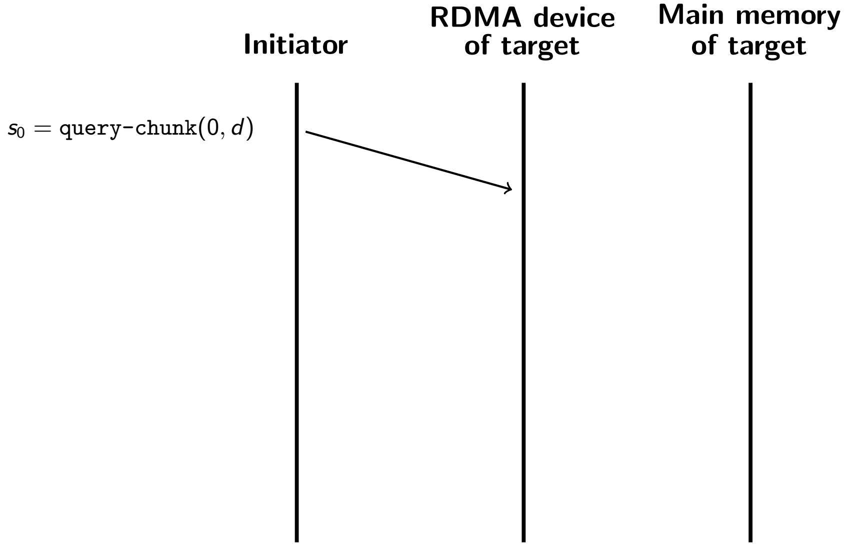
Defining `sync-chunk`(s)

Takes a handle s as parameter, waits until the *corresponding* query has been completed.

Linear Probing: Querying Visualized



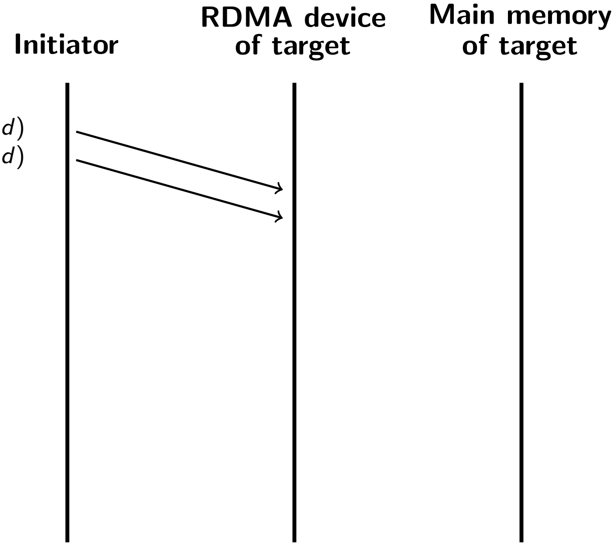
Linear Probing: Querying Visualized



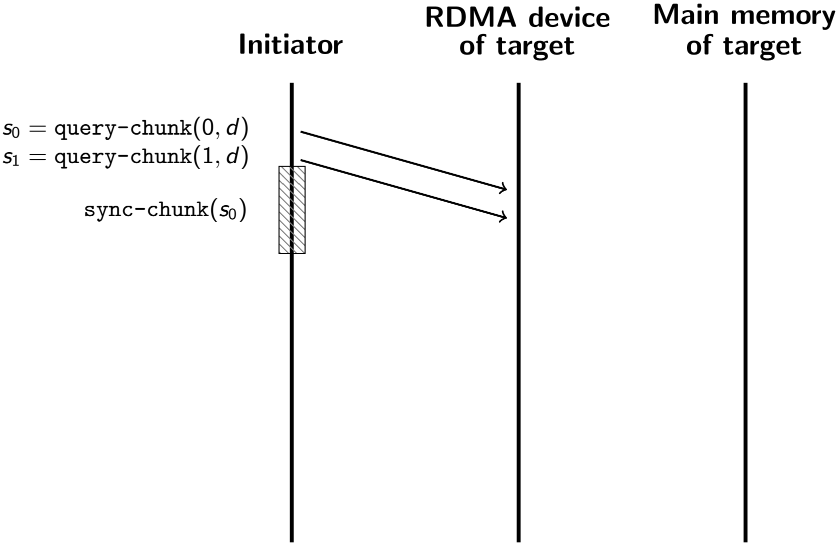
Linear Probing: Querying Visualized

$s_0 = \texttt{query-chunk}(0, d)$

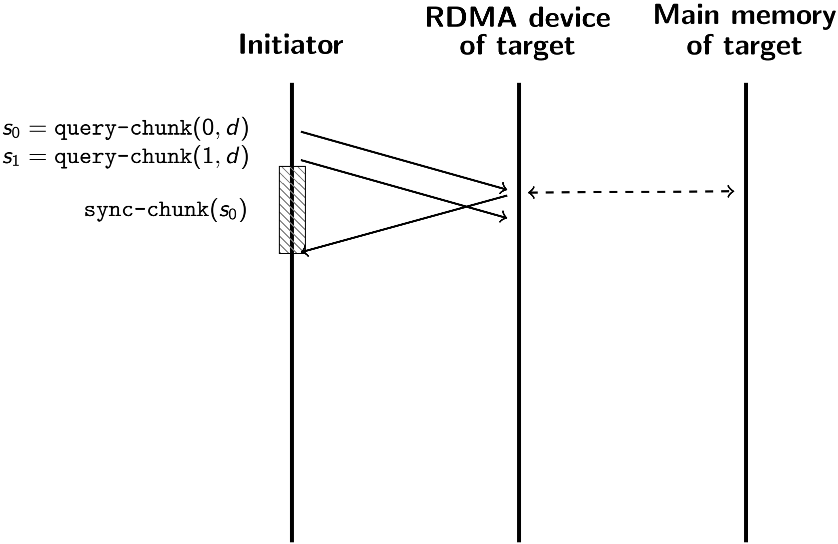
$s_1 = \texttt{query-chunk}(1, d)$



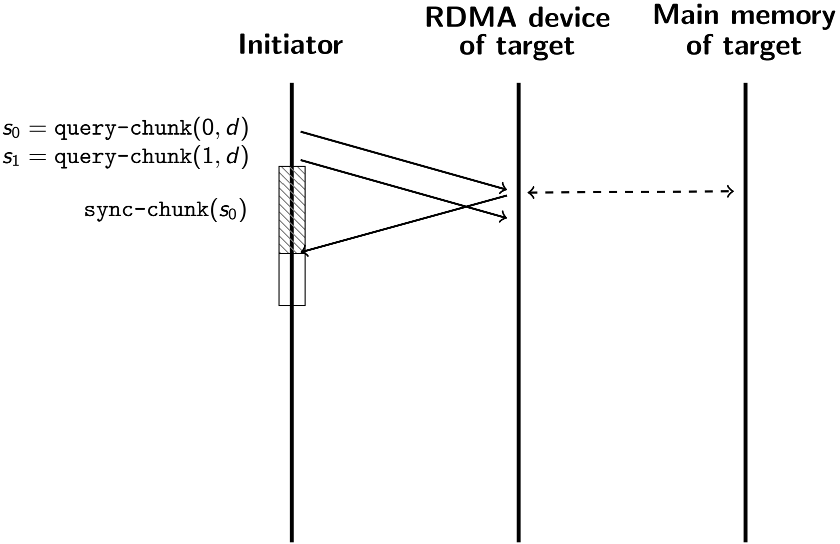
Linear Probing: Querying Visualized



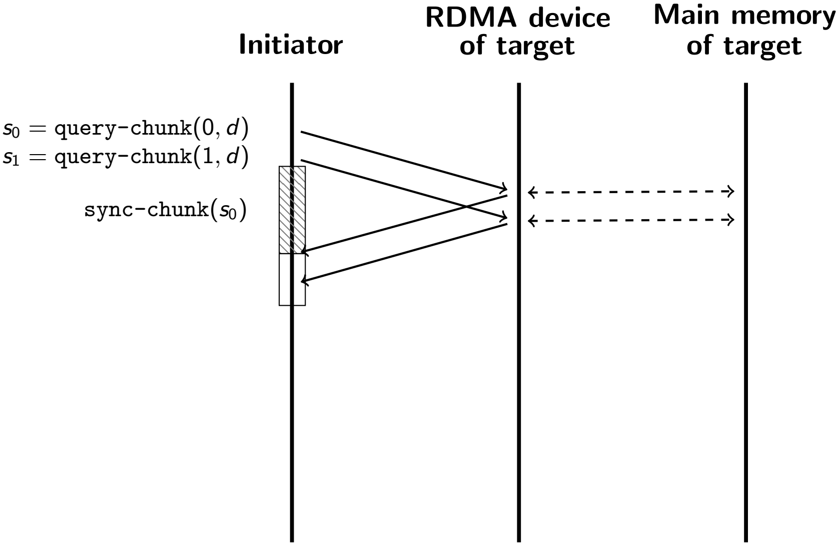
Linear Probing: Querying Visualized



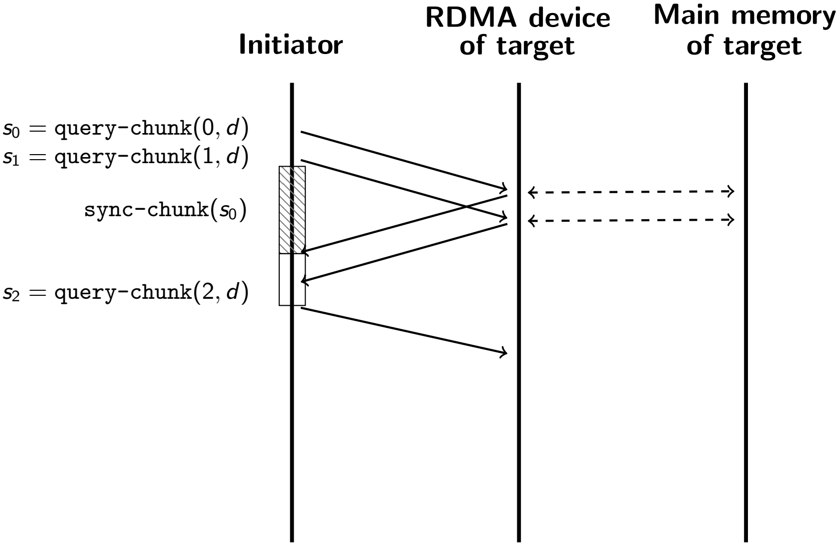
Linear Probing: Querying Visualized



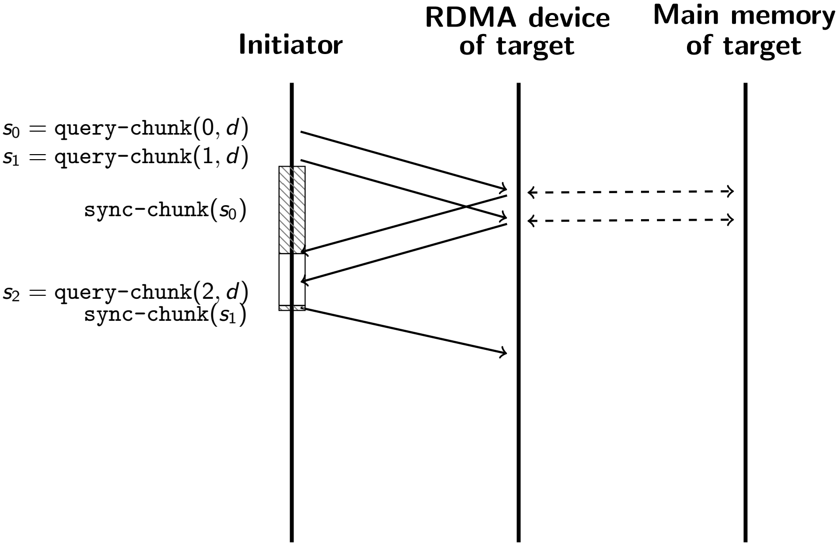
Linear Probing: Querying Visualized



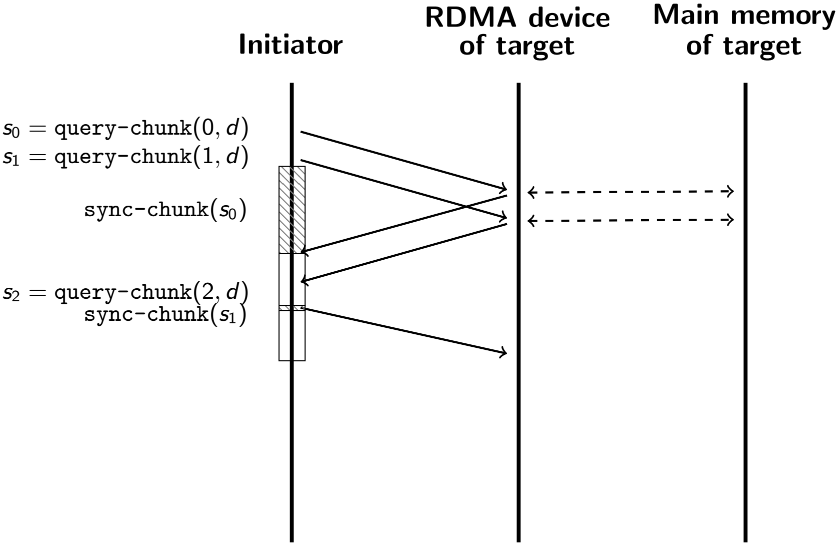
Linear Probing: Querying Visualized



Linear Probing: Querying Visualized



Linear Probing: Querying Visualized



Linear Probing: Querying Visualized

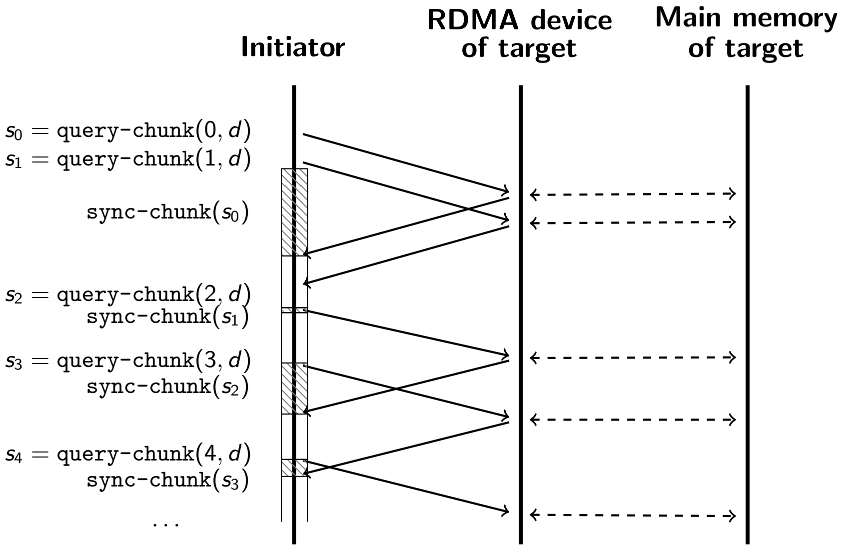


Table of Contents

Hash Table: Evaluation

Experimental Setup

All experiments have been perfomed on the *DAS-5* cluster:

- 66 machines
- 16 cores each (Intel E5-2630v3)
- 64 GB internal memory each
- connected via 48Gb/s Infiniband

Hash Table: Evaluation

Experimental Setup

All experiments have been perfomed on the *DAS-5* cluster:

- 66 machines
- 16 cores each (Intel E5-2630v3)
- 64 GB internal memory each
- connected via 48Gb/s Infiniband

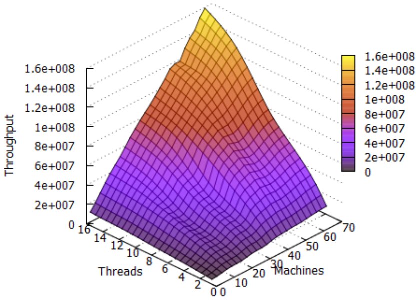
Benchmarks

Under different workloads, we measured:

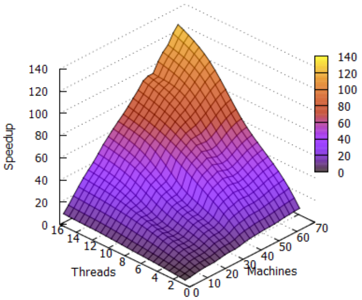
- Throughput of `find-or-put`
- Latency of `find-or-put`
- Roundtrips of `find-or-put`

Hash Table: Throughput

Total Throughput

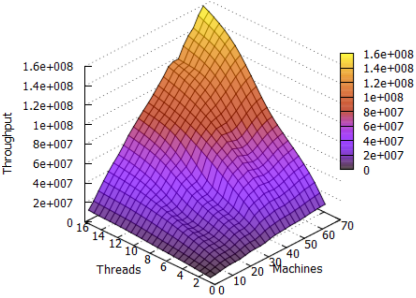


Speedup

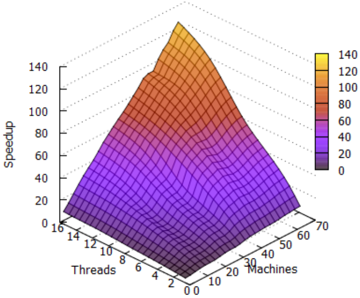


Hash Table: Throughput

Total Throughput



Speedup

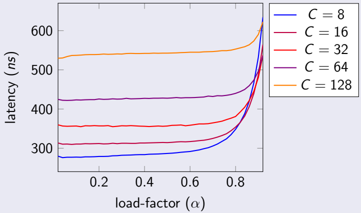


Observations

- Throughputs up to 140×10^6 reached (66 machines)
- Remote speedup up to 110 obtained
- Local throughput of 495×10^6 reached (1 threads)

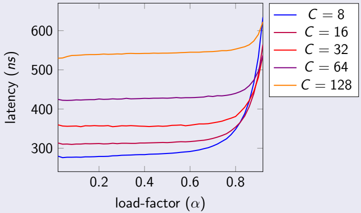
Hash Table: Latency

Local latency

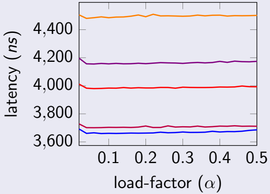


Hash Table: Latency

Local latency



Remote latency



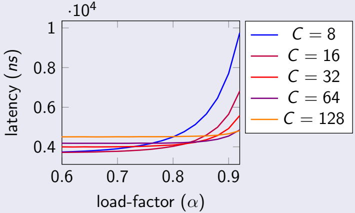


Table of Contents

Conclusions

General

- Minimizing roundtrips increases performance
- Overlapping queries reduces waiting-times and decreases latency
- Linear probing requires less roundtrips than Hopscotch and Cuckoo

Conclusions

General

- Minimizing roundtrips increases performance
- Overlapping queries reduces waiting-times and decreases latency
- Linear probing requires less roundtrips than Hopscotch and Cuckoo

Performance

- `find-or-put` takes $4.5\mu s$ on average with $\alpha = 0.9$ and $C = 64$
- Peak-throughput of 140×10^6 op/s obtained

Conclusions

General

- Minimizing roundtrips increases performance
- Overlapping queries reduces waiting-times and decreases latency
- Linear probing requires less roundtrips than Hopscotch and Cuckoo

Performance

- `find-or-put` takes $4.5\mu s$ on average with $\alpha = 0.9$ and $C = 64$
- Peak-throughput of 140×10^6 op/s obtained

Performance Indication

- **FaRM**: Inserts take $\sim 35\mu s$
- **Pilaf**: Operations take $\sim 30\mu s$
- **Nessie**: Inserts take $\sim 25\mu s$