High-Performance Distributed RMA Locks

Patrick Schmid, Maciej Besta, Torsten Hoefler
NEED FOR EFFICIENT LARGE-SCALE SYNCHRONIZATION
NEED FOR EFFICIENT LARGE-SCALE SYNCHRONIZATION
NEED FOR EFFICIENT LARGE-SCALE SYNCHRONIZATION
NEED FOR EFFICIENT LARGE-SCALE SYNCHRONIZATION
Locks
Locks
Locks

An example structure

Proc p

Proc q
Locks
Locks

An example structure

Proc p

lock

accesses

...
Locks

An example structure

Proc p

lock

accesses

unlock

Proc q
Locks

An example structure

Proc p

lock

accesses

unlock

Proc q

lock

accesses
Locks

Inuitive semantics

An example structure

Proc p

accesses

lock

unlock

Proc q

accesses

lock

...
**Locks**

Inuitive semantics

Various performance penalties

An example structure

- Proc p
- Proc q

lock
accesses
unlock
lock
accesses
LOCKS: CHALLENGES
LOCKS: CHALLENGES
LOCKS: CHALLENGES
LOCKS: CHALLENGES
LOCKS: CHALLENGES
LOCKS: CHALLENGES
LOCKS: CHALLENGES
LOCKS: CHALLENGES
LOCKS: CHALLENGES

We need intra- and inter-node topology-awareness

We need to cover arbitrary topologies
LOCKS: CHALLENGES
LOCKS: CHALLENGES

Locks: Challenges

We need to distinguish between readers and writers.

We need to distinguish between readers and writers

We need flexible performance for both types of processes

What will we use in the design?
WHAT WE WILL USE
MCS Locks

Proc
Cannot enter
Next proc

... Pointer to the queue tail

Proc
Cannot enter
Next proc

Proc
Can enter
Next proc
WHAT WE WILL USE
MCS Locks

Proc
Cannot enter
Next proc

Proc
Cannot enter
Next proc

Proc
Can enter
Next proc

Pointer to the queue tail
WHAT WE WILL USE
MCS Locks

Proc

Cannot enter

Next proc

Proc

Cannot enter

Next proc

Proc

Cannot enter

Next proc

Proc

Can enter

Next proc

Pointer to the queue tail
WHAT WE WILL USE
MCS Locks

Proc
Cannot enter
Next proc

Proc
Cannot enter
Next proc

Proc
Cannot enter
Next proc

Proc
Can enter
Next proc

Pointer to the queue tail
WHAT WE WILL USE
MCS Locks

Proc
Cannot enter
Next proc

Proc
Cannot enter
Next proc

Proc
Cannot enter
Next proc

Proc
Can enter
Next proc

Pointer to the queue tail
**WHAT WE WILL USE**

MCS Locks

*Diagram showing a queue with pointers and lock status*
WHAT WE WILL USE
MCS Locks

Proc
Cannot enter
Next proc

Proc
Cannot enter
Next proc

Proc
Cannot enter
Next proc

Proc
Can enter
Next proc

Pointer to the queue tail
WHAT WE WILL USE
MCS Locks

Proc
Cannot enter
Next proc

Proc
Cannot enter
Next proc

Proc
Cannot enter
Next proc

Proc
Can enter
Next proc

Pointer to the queue tail
WHAT WE WILL USE
MCS Locks

Proc
Cannot enter
Next proc

Proc
Cannot enter
Next proc

Proc
Next proc

Proc
Can enter
Next proc

Pointer to the queue tail
WHAT WE WILL USE
MCS Locks

Proc
Cannot enter
Next proc

Proc
Cannot enter
Next proc

Proc
Can enter
Next proc

Proc
Can enter
Next proc

Pointer to the queue tail
WHAT WE WILL USE
MCS Locks

Proc

Cannot enter

Next proc

Proc

Cannot enter

Next proc

Proc

Can enter

Next proc

Proc

Can enter

Next proc

Pointer to the queue tail
WHAT WE WILL USE
MCS Locks
WHAT WE WILL USE
Reader-Writer Locks
WHAT WE WILL USE
Reader-Writer Locks
WHAT WE WILL USE
Reader-Writer Locks
What we will use
Reader-Writer Locks
How to manage the design complexity?
How to manage the design complexity?

What mechanism to use for efficient implementation?
How to manage the design complexity?

How to ensure tunable performance?

What mechanism to use for efficient implementation?
How to manage the design complexity?

How to ensure tunable performance?

What mechanism to use for efficient implementation?
REMOTE MEMORY ACCESS (RMA) PROGRAMMING
REMOTE MEMORY ACCESS (RMA) PROGRAMMING
REMOTE MEMORY ACCESS (RMA) PROGRAMMING

Process p

Memory

A

Process q

Memory

B
REMOTE MEMORY ACCESS (RMA) PROGRAMMING

Process p
Memory
A

Process q
Memory
B

Cray BlueWaters
REMOTE MEMORY ACCESS (RMA) PROGRAMMING
REMOTE MEMORY ACCESS (RMA) PROGRAMMING

- Process p
  - Memory A

- Process q
  - Memory B

Cray BlueWaters
REMOTE MEMORY ACCESS (RMA) PROGRAMMING

Process p

Memory

A

Cray
BlueWaters

Process q

Memory

B
REMOTE MEMORY ACCESS (RMA) PROGRAMMING

Process p
Memory
A

A put

Process q
Memory
A
B

Cray
BlueWaters
REMOTE MEMORY ACCESS (RMA) PROGRAMMING

Process p

Memory

A

B

Process q

Memory

A

B

A put

get B

Cray BlueWaters
REMOTE MEMORY ACCESS (RMA) PROGRAMMING

Process p
Memory
A
B

Process q
Memory
A
B

put
get
flush

Cray BlueWaters
REMOTE MEMORY ACCESS (RMA) PROGRAMMING

Process p

Memory

A

B

put

get

flush

Process q

Memory

A

B

Cray BlueWaters
REMOTE MEMORY ACCESS PROGRAMMING

- Implemented in hardware in NICs in the majority of HPC networks (RDMA support).
REMOTE MEMORY ACCESS PROGRAMMING

- Implemented in hardware in NICs in the majority of HPC networks (RDMA support).
REMOTE MEMORY ACCESS PROGRAMMING

- Implemented in hardware in NICs in the majority of HPC networks (RDMA support).
REMOTE MEMORY ACCESS PROGRAMMING

- Implemented in hardware in NICs in the majority of HPC networks (RDMA support).
REMOTE MEMORY ACCESS PROGRAMMING

- Implemented in hardware in NICs in the majority of HPC networks (RDMA support).
REMOTE MEMORY ACCESS PROGRAMMING

- Supported by many HPC libraries and languages
REMOTE MEMORY ACCESS PROGRAMMING

- Supported by many HPC libraries and languages
REMOTE MEMORY ACCESS PROGRAMMING

- Supported by many HPC libraries and languages
How to manage the design complexity?

How to ensure tunable performance?

What mechanism to use for efficient implementation?
How to manage the design complexity?

How to ensure tunable performance?

What mechanism to use for efficient implementation?
How to manage the design complexity?

How to ensure tunable performance?

What mechanism to use for efficient implementation?
How to manage the design complexity?
How to manage the design complexity?
How to manage the design complexity?

Modular design
How to manage the design complexity?

Modular design

W1, W2, W3

W4, W5

W6, W7, W8
How to manage the design complexity?

Modular design
How to manage the design complexity?

Each element has its own distributed MCS queue (DQ) of writers.
How to manage the design complexity?

Each element has its own distributed MCS queue (DQ) of writers.

Modular design
How to manage the design complexity?

Each element has its own distributed MCS queue (DQ) of writers.

Modular design
How to manage the design complexity?

Each element has its own distributed MCS queue (DQ) of writers

Modular design
How to manage the design complexity?

Each element has its own distributed MCS queue (DQ) of writers.

Modular design
How to manage the design complexity?

Each element has its own distributed MCS queue (DQ) of writers

MCS queues form a distributed tree (DT)

Modular design
How to manage the design complexity?

Each element has its own distributed MCS queue (DQ) of writers.

MCS queues form a distributed tree (DT).

Modular design.
How to manage the design complexity?

Each element has its own distributed MCS queue (DQ) of writers.

Readers and writers synchronize with a distributed counter (DC).

MCS queues form a distributed tree (DT).

How to manage the design complexity?

Modular design.
Each element has its own distributed MCS queue (DQ) of writers.

Readers and writers synchronize with a distributed counter (DC).

MCS queues form a distributed tree (DT).

How to manage the design complexity?

Modular design.
How to ensure tunable performance?
How to ensure tunable performance?

A tradeoff parameter for every structure
How to ensure tunable performance?

Each DQ: fairness vs throughput of writers

A tradeoff parameter for every structure
How to ensure tunable performance?

Each DQ: fairness vs throughput of writers

A tradeoff parameter for every structure

DT: a parameter for the throughput of readers vs writers
How to ensure tunable performance?

Each DQ: fairness vs throughput of writers

DC: a parameter for the latency of readers vs writers

DT: a parameter for the throughput of readers vs writers

A tradeoff parameter for every structure

R1 ... R9

W1 W3
W5 W8
W2 W3
W4 W5
W6 W7 W8

2 2 2 2
R1 R2 R3 R4
R5 R6 R7 R8
R9

How to ensure tunable performance?

Each DQ: fairness vs throughput of writers

DC: a parameter for the latency of readers vs writers

DT: a parameter for the throughput of readers vs writers

A tradeoff parameter for every structure

R1 ... R9

W1 W3
W5 W8
W2 W3
W4 W5
W6 W7 W8

2 2 2 2
R1 R2 R3 R4
R5 R6 R7 R8
R9

DISTRIBUTED MCS QUEUES (DQs)
Throughput vs Fairness
DISTRIBUTED MCS QUEUES (DQs)

Throughput vs Fairness

Each DQ: The maximum number of lock passings within a DQ at level i, before it is passed to another DQ at i. 

\[ T_{L,i} \]
DISTRIBUTED MCS QUEUES (DQs)
Throughput vs Fairness

Each DQ: The maximum number of lock passings within a DQ at level i, before it is passed to another DQ at i.

\[ T_{L,i} \]
**DISTRIBUTED MCS QUEUES (DQs)**

**Throughput vs Fairness**

Each DQ: The maximum number of lock passings within a DQ at level $i$, before it is passed to another DQ at $i$. 

$$T_{L,i}$$
DISTRIBUTED MCS QUEUES (DQs)
Throughput vs Fairness

Each DQ: The maximum number of lock passings within a DQ at level $i$, before it is passed to another DQ at $i$.

$T_{L,i}$
DISTRIBUTED MCS QUEUES (DQs)
Throughput vs Fairness

Larger $T_{L,i}$ : more throughput at level $i$.
Smaller $T_{L,i}$ : more fairness at level $i$.

Each DQ: The maximum number of lock passings within a DQ at level $i$, before it is passed to another DQ at $i$.

$T_{L,i}$
DISTRIBUTED TREE OF QUEUES (DT)
Throughput of readers vs writers
DISTRIBUTED TREE OF QUEUES (DT)

Throughput of readers vs writers

DT: The maximum number of consecutive lock passings within readers ($T_R$).
DISTRIBUTED TREE OF QUEUES (DT)
Throughput of readers vs writers

DT: The maximum number of consecutive lock passings within readers ($T_R$).
DISTRIBUTED COUNTER (DC)
Latency of readers vs writers
**DISTRIBUTED COUNTER (DC)**
Latency of readers vs writers

DC: every $k$th compute node hosts a partial counter, all of which constitute the DC.

$$k = T_{DC}$$
DISTRIBUTED COUNTER (DC)
Latency of readers vs writers

DC: every $k$th compute node hosts a partial counter, all of which constitute the DC.

$k = T_{DC}$
DISTRIBUTED COUNTER (DC)
Latency of readers vs writers

DC: every \( k \)th compute node hosts a partial counter, all of which constitute the DC.

\[ k = T_{DC} \]

A writer holds the lock
DISTRIBUTED COUNTER (DC)
Latency of readers vs writers

DC: every $k$th compute node hosts a partial counter, all of which constitute the DC.

$k = T_{DC}$

A writer holds the lock
Readers that arrived at the CS
DISTRIBUTED COUNTER (DC)
Latency of readers vs writers

DC: every $k$th compute node hosts a partial counter, all of which constitute the DC.

$$k = T_{DC}$$

A writer holds the lock

Readers that arrived at the CS

Readers that left the CS
DISTRIBUTED COUNTER (DC)
Latency of readers vs writers

DC: every \( k \)th compute node hosts a partial counter, all of which constitute the DC.

\[ k = T_{DC} \]

A writer holds the lock

Readers that arrived at the CS

Readers that left the CS
**DISTRIBUTED COUNTER (DC)**
Latency of readers vs writers

DC: every \( k \)th compute node hosts a partial counter, all of which constitute the DC.

\[
k = T_{DC}
\]

A writer holds the lock

Readers that arrived at the CS

Readers that left the CS

\[T_{DC} = 1\]
**DISTRIBUTED COUNTER (DC)**

Latency of readers vs writers

DC: every $k$th compute node hosts a partial counter, all of which constitute the DC.

$k = T_{DC}$

A writer holds the lock

Readers that arrived at the CS

Readers that left the CS

$T_{DC} = 1$
DISTRIBUTED COUNTER (DC)
Latency of readers vs writers

DC: every $k$th compute node hosts a partial counter, all of which constitute the DC.

$$k = T_{DC}$$

A writer holds the lock
Readers that arrived at the CS
Readers that left the CS
DISTRIBUTED COUNTER (DC)
Latency of readers vs writers

DC: every $k$th compute node hosts a partial counter, all of which constitute the DC.

$$k = T_{DC}$$

A writer holds the lock $b|x|y$

Readers that arrived at the CS

Readers that left the CS

$$T_{DC} = 2$$
DISTRIBUTED COUNTER (DC)
Latency of readers vs writers

DC: every \( k \)th compute node hosts a partial counter, all of which constitute the DC.

\[ k = T_{DC} \]

A writer holds the lock

Readers that arrived at the CS

Readers that left the CS

\[ T_{DC} = 2 \]
THE SPACE OF DESIGNS
THE SPACE OF DESIGNS
THE SPACE OF DESIGNS
THE SPACE OF DESIGNS

Higher throughput of **writers** vs **readers**

$T_R$
THE SPACE OF DESIGNS

Higher throughput of writers vs readers
Higher throughput of **writers** vs **readers**

Lower latency of **writers** vs **readers**

**THE SPACE OF DESIGNS**
**THE SPACE OF DESIGNS**

- Higher throughput of *writers* vs *readers* (T_L, i)
- Lower latency of *writers* vs *readers* (T_DC)
**The Space of Designs**

- Higher throughput of *writers* vs *readers*
- Lower latency of *writers* vs *readers*
- Local vs fairness (for writers)

$T_{\text{DC}}$  
$T_{L,i}$  
$T_{R}$
THE SPACE OF DESIGNS

Locality vs fairness (for writers)

Higher throughput of writers vs readers

Lower latency of writers vs readers

Higher throughput of writers vs readers
THE SPACE OF DESIGNS

Higher throughput of writers vs readers

Lower latency of writers vs readers

Locality vs fairness (for writers)
LOCK ACQUIRE BY READERS
**Lock Acquire by Readers**

A lightweight acquire protocol for readers: only one atomic fetch-and-add (FAA) operation.
LOCK ACQUIRE BY READERS

A lightweight acquire protocol for readers: only one atomic fetch-and-add (FAA) operation

A writer holds the lock

Readers that arrived at the CS

Readers that left the CS
LOCK ACQUIRE BY READERS

A lightweight acquire protocol for readers: only one atomic fetch-and-add (FAA) operation

A writer holds the lock

Readers that arrived at the CS

Readers that left the CS

0|7|7

0|1|1
LOCK ACQUIRE BY READERS

A lightweight acquire protocol for readers: only one atomic fetch-and-add (FAA) operation

A writer holds the lock
Readers that arrived at the CS
Readers that left the CS
LOCK ACQUIRE BY READERS

A lightweight acquire protocol for readers: only one atomic fetch-and-add (FAA) operation.

A writer holds the lock

Readers that arrived at the CS

Readers that left the CS
Lock Acquire by Readers

A lightweight acquire protocol for readers: only one atomic fetch-and-add (FAA) operation.

A writer holds the lock.

Readers that arrived at the CS.

Readers that left the CS.
**Lock Acquire by Readers**

A lightweight acquire protocol for readers: only one atomic fetch-and-add (FAA) operation.

A writer holds the lock. 
Readers that arrived at the CS. 
Readers that left the CS.
LOCK ACQUIRE BY READERS

A lightweight acquire protocol for readers: only one atomic fetch-and-add (FAA) operation.

A writer holds the lock
Readers that arrived at the CS
Readers that left the CS
**Lock Acquire by Readers**

A lightweight acquire protocol for readers: only one atomic fetch-and-add (FAA) operation.

- A writer holds the lock $b|x|y$.
- Readers that arrived at the CS.
- Readers that left the CS.

The diagram illustrates the sequence of readers attempting to acquire the lock and the writer holding it.
**LOCK ACQUIRE BY READERS**

A lightweight acquire protocol for readers: only one atomic fetch-and-add (FAA) operation.

A writer holds the lock.

Readers that arrived at the CS.

Readers that left the CS.
LOCK ACQUIRE BY READERS

A lightweight acquire protocol for readers: only one atomic fetch-and-add (FAA) operation.

A writer holds the lock: \( b|x|y \)

Readers that arrived at the CS: \( R_1, R_2, R_3, R_4 \)

Readers that left the CS: \( R_2, R_3, R_1, R_4 \)
**LOCK ACQUIRE BY READERS**

A lightweight acquire protocol for readers: only one atomic fetch-and-add (FAA) operation.

A writer holds the lock

Readers that arrived at the CS

Readers that left the CS
**Lock Acquire by Writers**
LOCK ACQUIRE BY WRITERS
LOCK ACQUIRE BY WRITERS
LOCK ACQUIRE BY WRITERS
LOCK ACQUIRE BY WRITERS

Acquire MCS

Acquire MCS

0|9|9

W9 → W1 → W3

W9 → W1

W2 → W3

W4 → W5

W6 → W7 → W8

0|3|3

0|8|8

0|5|5

W5 → W8

W5 → W8

W4 → W5

W6 → W7 → W8
**Lock Acquire by Writers**

Acquire MCS

0|9|9
W9 → W1
W9

0|3|3
W2 → W3
W3

0|8|8
W4 → W5
W5

0|5|5
W6 → W7 → W8
W8
Lock Acquire by Writers

Acquire the main MCS lock

Acquire MCS

Acquire MCS
**Lock Acquire by Writers**

- **R1**, **R2**, ..., **R9**
- Acquire the main MCS lock

Acquire MCS:

- **W9** → **W1** → **W3**
- **W9** → **W3** → **W8**
- **W5** → **W8**

Acquire MCS:

- **W9** → **W1**
- **W2** → **W3**
- **W4** → **W5**
- **W6** → **W7** → **W8**
Lock Acquire by Writers

Acquire the main MCS lock

Acquire MCS

Acquire MCS
Lock Acquire by Writers

Acquire the main MCS lock

Acquire MCS

Acquire MCS

Acquire MCS
Lock Acquire by Writers

Acquire the main MCS lock

Acquire MCS

Acquire MCS
LOCK ACQUIRE BY WRITERS

Acquire the main lock

Acquire MCS

Acquire MCS

Acquire the main MCS lock
EVALUATION

- CSCS Piz Daint (Cray XC30)
- 5272 compute nodes
- 8 cores per node
- 169TB memory
EVALUATION

CONSIDERED BENCHMARKS
EVALUATION

CONSIDERED BENCHMARKS

The latency benchmark
EVALUATION

The \textbf{latency benchmark}

\textbf{Throughput benchmarks:}

- Empty-critical-section
- Single-operation
- Wait-after-release
- Workload-critical-section
EVALUATION
CONSIDERED BENCHMARKS

The latency benchmark

DHT
Distributed hashtable evaluation

Throughput benchmarks:
- Empty-critical-section
- Single-operation
- Wait-after-release
- Workload-critical-section
EVALUATION
DISTRIBUTED COUNTER ANALYSIS

Throughput, 2% writers
Single-operation benchmark

Throughput [mln locks/s]

<table>
<thead>
<tr>
<th>T_{DC}</th>
<th>64</th>
<th>32</th>
<th>16</th>
<th>8</th>
<th>4</th>
<th>2</th>
</tr>
</thead>
</table>

MPI processes (P)
EVALUATION
READER THRESHOLD ANALYSIS

Throughput, 0.2% writers,
Empty-critical-section benchmark

Throughput [mln locks/s]

MPI processes (P)
EVALUATION
COMPARISON TO THE STATE-OF-THE-ART

EVALUATION
COMPARISON TO THE STATE-OF-THE-ART

Throughput, single-operation benchmark

Percentages are values of $F_W$

Throughput [mln locks/s]

MPI processes (P)

EVALUATION
DISTRIBUTED HASHTABLE

20% writers

10% writers

OTHER ANALYSES
OTHER ANALYSES
CONCLUSIONS
CONCLUSIONS

Modular distributed RMA lock, correctness with SPIN
CONCLUSIONS

Modular distributed RMA lock, correctness with SPIN

Parameter-based design, feasible with various RMA libs/languages
CONCLUSIONS

Modular distributed RMA lock, correctness with SPIN

Parameter-based design, feasible with various RMA libs/languages

Improves latency and throughput over state-of-the-art
CONCLUSIONS

Modular distributed RMA lock, correctness with SPIN

Parameter-based design, feasible with various RMA libs/languages

Improves latency and throughput over state-of-the-art

Enables high-performance distributed hashtabled
CONCLUSIONS

Modular distributed RMA lock, correctness with SPIN
Parameter-based design, feasible with various RMA libs/languages
Improves latency and throughput over state-of-the-art
Enables high-performance distributed hashtabled

Thank you for your attention
DISTRIBUTED TREE OF QUEUES (DT)
Throughput of readers vs writers
**Distributed Tree of Queues (DT)**

Throughput of readers vs writers

DT: The maximum number of consecutive lock passings within writers ($T_W$) and readers ($T_R$).
DISTRIBUTED TREE OF QUEUES (DT)
Throughput of readers vs writers

DT: The maximum number of consecutive lock passings within writers (\(T_W\)) and readers (\(T_R\)).

\[ T_W = \prod_{i=1}^{N} T_{L,i} \]
DISTRIBUTED TREE OF QUEUES (DT)
Throughput of readers vs writers

DT: The maximum number of consecutive lock passings within writers ($T_W$) and readers ($T_R$).

$$T_W = \prod_{i=1}^{N} T_{L,i}$$
THE SPACE OF DESIGNS
THE SPACE OF DESIGNS
THE SPACE OF DESIGNS
THE SPACE OF DESIGNS

Higher throughput of writers vs readers $T_R$
THE SPACE OF DESIGNS

Higher throughput of **writers** vs **readers**
THE SPACE OF DESIGNS

Higher throughput of **writers** vs **readers**

$T_R$
THE SPACE OF DESIGNS

Higher throughput of **writers** vs **readers**
THE SPACE OF DESIGNS

Higher throughput of **writers** vs **readers**

- $T_{DC}$: Lower latency of writers vs readers
- $T_R$: Higher throughput of writers vs readers
Higher throughput of **writers** vs **readers**

Lower latency of **writers** vs **readers**

**THE SPACE OF DESIGNS**
THE SPACE OF DESIGNS

Higher throughput of writers vs readers

Lower latency of writers vs readers

$T_{DC}$

$T_R$
THE SPACE OF DESIGNS

Higher throughput of **writers** vs **readers**

Lower latency of **writers** vs **readers**

$T_D C$

$T_{L,i}$

$T_R$
THE SPACE OF DESIGNS

Higher throughput of writers vs readers

Lower latency of writers vs readers

Locality vs fairness (for writers)
THE SPACE OF DESIGNS

Higher throughput of writers vs readers

Locality vs fairness (for writers)

Lower latency of writers vs readers

$T_{L,i}$

$T_{R}$

$T_{DC}$
**THE SPACE OF DESIGNS**

- **Higher throughput of writers vs readers**
- **Locality vs fairness (for writers)**
- **Lower latency of writers vs readers**

Equation: $$T_{L,i}$$
THE SPACE OF DESIGNS

Higher throughput of **writers** vs **readers**

- Locality vs fairness (for writers)
- Lower latency of writers vs readers

$T_{L,i}$

$T_{DC}$

$T_{R}$
THE SPACE OF DESIGNS

Higher throughput of writers vs readers

Locality vs fairness (for writers)

Lower latency of writers vs readers

Design A

Design B
**EVALUATION**

D-MCS VS OTHERS

**Latency (LB)**

- **Scheme**
  - foMPI-Spin
  - D-MCS
  - RMA-MCS

**Throughput (ECSB)**

- **Scheme**
  - foMPI-Spin
  - D-MCS
  - RMA-MCS

Performance initially increases due to high intra-node bandwidth.
EVALUATION
WRITER THRESHOLD ANALYSIS

Throughput, 25% of writers
Single-operation benchmark

![Diagram showing throughput in million locks per second against MPI processes (P) for different values of T_Li product (500, 1000, 2500, 5000, 7500).]
EVALUATION

FAIRNESS VS THROUGHPUT ANALYSIS

Throughput, 25% of writers, Single-operation benchmark

![Graph showing throughput versus MPI processes for different load ranges.](image-url)
EVALUATION
READER THRESHOLD ANALYSIS

Throughput, 2% and 5% writers, Empty-critical-section benchmark

![Graph showing throughput vs. number of MPI processes (P) for different reader thresholds and writer percentages.](image-url)
EVALUATION
DISTRIBUTED HASHTABLE

2% of writers

0% of writers

FEASIBILITY ANALYSIS
# Feasibility Analysis

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Put</td>
<td>UPC_SET</td>
<td>bupc_atomicX_set_RS</td>
<td>shmem_swap</td>
</tr>
<tr>
<td>Get</td>
<td>UPC_GET</td>
<td>bupc_atomicX_read_RS</td>
<td>shmem_mswap</td>
</tr>
<tr>
<td>Accumulate</td>
<td>UPC_INC</td>
<td>bupc_atomicX_fetchadd_RS</td>
<td>shmem_fadd</td>
</tr>
<tr>
<td>FAO (SUM)</td>
<td>UPC_INC, UPC_DEC</td>
<td>bupc_atomicX_fetchadd_RS</td>
<td>shmem_fadd</td>
</tr>
<tr>
<td>FAO (REPLACE)</td>
<td>UPC_SET</td>
<td>bupc_atomicX_swap_RS</td>
<td>shmem_swap</td>
</tr>
<tr>
<td>CAS</td>
<td>UPC_CSWAP</td>
<td>bupc_atomicX_cswap_RS</td>
<td>shmem_cswap</td>
</tr>
</tbody>
</table>
## Feasibility Analysis

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Put</td>
<td>UPC_SET</td>
<td>bupc_atomicX_set_RS</td>
</tr>
<tr>
<td>Get</td>
<td>UPC_GET</td>
<td>bupc_atomicX_read_RS</td>
</tr>
<tr>
<td>Accumulate</td>
<td>UPC_INC</td>
<td>bupc_atomicX_fetchadd_RS</td>
</tr>
<tr>
<td>FAO (SUM)</td>
<td>UPC_INC, UPC_DEC</td>
<td>bupc_atomicX_fetchadd_RS</td>
</tr>
<tr>
<td>FAO (REPLACE)</td>
<td>UPC_SET</td>
<td>bupc_atomicX_swap_RS</td>
</tr>
<tr>
<td>CAS</td>
<td>UPC_CSAMP</td>
<td>bupc_atomicX_cswap_RS</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Fortran 2008 [27]</th>
<th>Linux RDMA/IB [33, 43]</th>
<th>iWARP [18, 41]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Put</td>
<td>atomic_define</td>
<td>masked CmpSwap</td>
</tr>
<tr>
<td>Get</td>
<td>atomic_ref</td>
<td>masked CmpSwap</td>
</tr>
<tr>
<td>Accumulate</td>
<td>atomic_add</td>
<td>FetchAdd</td>
</tr>
<tr>
<td>FAO (SUM)</td>
<td>atomic_add</td>
<td>FetchAdd</td>
</tr>
<tr>
<td>FAO (REPLACE)</td>
<td>atomic_define*</td>
<td>Masked CmpSwap</td>
</tr>
<tr>
<td>CAS</td>
<td>atomic_cas</td>
<td>CmpSwap</td>
</tr>
</tbody>
</table>
RMA-RW
Required Operations

Process p
Memory

Process q
Memory
3
3
3
3
RMA-RW
Required Operations

Process p

Memory

6 put

Process q

Memory

3

3

3

3
RMA-RW
Required Operations

Process p
Memory

6 put

Process q
Memory

6
3
3
3
RMA-RW
Required Operations
RMA-RW
Required Operations

Process p

Memory

3

Process q

Memory

6

6 put

get

3

3

3

3

3
RMA-RW
Required Operations

Process p
Memory

6 put

get

6 Fetch-and-Add (FAA)

3

6

3

Process q
Memory

6

3

3

3
**RMA-RW**

**Required Operations**

**Process p**

- Memory

  - **3** put
  - **3**

**Process q**

- Memory

  - **6** get
  - **3**
  - **6** Fetch-and-Add (FAA)
  - **3**
  - **3**
  - **3**
RMA-RW
Required Operations

Process p

Memory

Process q

Memory

6 put

get

6 Fetch-and-Add (FAA)
RMA-RW
Required Operations

Process p
Memory

6 put
3 get
3 Fetch-and-Add (FAA)
3 replace

6 put
3 get
9 Fetch-and-Add (FAA)
3 replace

Process q
Memory
RMA-RW
Required Operations

Process p

<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 put</td>
</tr>
<tr>
<td>3 get</td>
</tr>
<tr>
<td>3 FAA</td>
</tr>
<tr>
<td>3 replace</td>
</tr>
</tbody>
</table>

Process q

<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 put</td>
</tr>
<tr>
<td>3 get</td>
</tr>
<tr>
<td>6 FAA</td>
</tr>
<tr>
<td>6 replace</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>
RMA-RW
Required Operations

Process p

Memory

3

6 put

get

6 Fetch-and-Add (FAA)

6 replace

3 8 Compare-and-Swap (CAS)

Process q

Memory

6

3

9

6

6

3
RMA-RW
Required Operations

Process p

Memory

3 put

get

6 Fetch-and-Add (FAA)

3 replace

3 Compare-and-Swap (CAS)

Process q

Memory

6

3

6

9

6

8
RMA-RW
Required Operations

Process p

Memory

Process q

Memory

put

get

6

3

3

6

6

3

6

9

6

6

Compare-and-Swap (CAS)

3

8