Term Rewriting with Shared Memory

Mikhail Ushakov

Radboud University Nijmegen

Supervisor : Dr C.L.M. Kop

June 2, 2026

Why Should We Care?

web servers

databases

concurrent graph algorithms

parallel sorting operations

operating systems

Why Should We Care?

web servers

databases

concurrent graph algorithms

Shared data

parallel sorting operations

operating systems

Why Should We Care?

Concurrent algorithms and data structures are **fundamental** in modern world

Why Should We Care?

Concurrent algorithms and data structures are **fundamental** in modern world

Concurrency gives **performance benefits**, but introduces **extra complexity**

Why Should We Care?

Concurrent algorithms and data structures are **fundamental** in modern world

Concurrency gives **performance benefits**, but introduces **extra complexity**

Concurrent software requires **careful design** and **formal reasoning**

What Can Help Us?

We are looking for a framework that would allow us to do:

What Can Help Us?

We are looking for a framework that would allow us to do:

✓ Program **termination analysis**

What Can Help Us?

We are looking for a framework that would allow us to do:

✓ Program **termination analysis**

✓ Program **equivalence checking**

What Can Help Us?

We are looking for a framework that would allow us to do:

✓ Program **termination analysis**

✓ Program **equivalence checking**

✓ **Formal verification** of programs and models

What Can Help Us?

Term Rewriting Systems

(TRS)

What Can Help Us?

Logically Constrained

Term Rewriting Systems

(**LC**TRS)

The End?

The End?

The End?

This is Not The End!

This is Not The End!

✗ Traditional TRS **lacks** direct support for shared mutable memory

This is Not The End!

✗ Traditional TRS **lacks** direct support for shared mutable memory

✗ Existing approaches **do not** combine parallel rewriting with explicit memory effects

This is Not The End!

✗ Traditional TRS **lacks** direct support for shared mutable memory

✗ Existing approaches **do not** combine parallel rewriting with explicit memory effects

✗ This **limits** rewriting-based modelling of concurrent data structures

Goal of the Research Project

Main Research Question

Can we extend term rewriting with explicit shared memory so that it is **formal**, **implementable**, and **useful for modelling concurrent algorithms**?

- ☐ Introduce **Memory Term Rewriting Systems (MemTRSs)** as an extension of LCTRSs.
- ☐ Add explicit shared-memory operations
- ☐ Evaluate the expressiveness and practicality of this model through data structures, algorithms, and experiments.

LCTRS Recap: Rewriting Rules

$$r_1 : \mathbf{max}(a, \mathbf{cons}(b, l)) \to \mathbf{max}(b, l)\ [a < b]$$
$$r_2 : \mathbf{max}(a, \mathbf{cons}(b, l)) \to \mathbf{max}(a, l)\ [a \geq b]$$
$$r_3 : \mathbf{max}(a, \mathtt{nil}) \to a$$

LCTRS Recap: Reduction Example

$$\mathbf{max}(-1, \mathbf{cons}(2, \mathbf{cons}(3, \mathbf{cons}(0, \mathtt{nil}))))$$

LCTRS Recap: Reduction Example

$$\mathbf{max}(-1, \mathbf{cons}(2, \mathbf{cons}(3, \mathbf{cons}(0, \mathtt{nil}))))$$
$$\xrightarrow{r1} \mathbf{max}(2, \mathbf{cons}(3, \mathbf{cons}(0, \mathtt{nil})))$$

LCTRS Recap: Reduction Example

$$\mathbf{max}(-1, \mathbf{cons}(2, \mathbf{cons}(3, \mathbf{cons}(0, \mathtt{nil}))))$$
$$\xrightarrow{r1} \mathbf{max}(2, \mathbf{cons}(3, \mathbf{cons}(0, \mathtt{nil}))) \qquad \text{(because } -1 < 2\text{)}$$
$$\xrightarrow{r1} \mathbf{max}(3, \mathbf{cons}(0, \mathtt{nil}))$$

LCTRS Recap: Reduction Example

$$\mathbf{max}(-1, \mathbf{cons}(2, \mathbf{cons}(3, \mathbf{cons}(0, \mathtt{nil}))))$$

$$\xrightarrow{r1} \mathbf{max}(2, \mathbf{cons}(3, \mathbf{cons}(0, \mathtt{nil}))) \qquad (\text{because } -1 < 2)$$

$$\xrightarrow{r1} \mathbf{max}(3, \mathbf{cons}(0, \mathtt{nil})) \qquad (\text{because } 2 < 3)$$

$$\xrightarrow{r2} \mathbf{max}(3, \mathtt{nil})$$

LCTRS Recap: Reduction Example

$$\mathbf{max}(-1, \mathbf{cons}(2, \mathbf{cons}(3, \mathbf{cons}(0, \mathtt{nil}))))$$

$$\xrightarrow{r1} \mathbf{max}(2, \mathbf{cons}(3, \mathbf{cons}(0, \mathtt{nil}))) \qquad (\text{because } -1 < 2)$$

$$\xrightarrow{r1} \mathbf{max}(3, \mathbf{cons}(0, \mathtt{nil})) \qquad (\text{because } 2 < 3)$$

$$\xrightarrow{r2} \mathbf{max}(3, \mathtt{nil}) \qquad (\text{because } 3 \geq 0)$$

$$\xrightarrow{r3} 3$$

Where is Shared State? MemTRS Core Idea

term

Where is Shared State? MemTRS Core Idea

$$\langle term, mem \rangle$$

Where is Shared State? MemTRS Core Idea

$$\langle term, mem \rangle$$

$$mem : \mathbb{Z} \rightharpoonup \mathbb{Z}$$

Kernel Memory Operations

GET : $[int] \Rightarrow int$

Reads from memory.

If x is valid: $\langle \mathbf{GET}(x), mem \rangle \to \langle mem(x), mem \rangle$

SET : $[int \times int] \Rightarrow bool$

Writes to memory.

If x is valid: $\langle \mathbf{SET}(x, v), mem \rangle \to \langle \mathtt{true}, mem[x \mapsto v] \rangle$

CAS : $[int \times int \times int] \Rightarrow bool$

Atomic compare-and-swap.

If $mem(x) = e$: $\langle \mathbf{CAS}(x, e, v), mem \rangle \to \langle \mathtt{true}, mem[x \mapsto v] \rangle$

Parallel Reduction

Memory

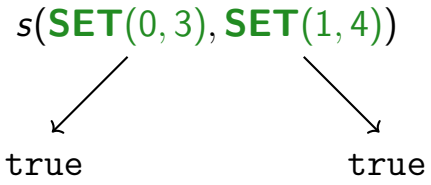
$mem(0) = 0$
$mem(1) = 0$

$$s(\mathbf{SET}(0, 3), \mathbf{SET}(1, 4))$$

Parallel Reduction

Memory

$mem(0) = 3$
$mem(1) = 4$



Parallel Reduction

Memory

$mem(0) = 3$
$mem(1) = 4$

$$s(\ \texttt{true}\ ,\ \texttt{true}\ )$$

Nondeterminism from Parallel Writes

Memory

$mem(0) = 0$

$$s(\mathbf{SET}(0, 3), \mathbf{SET}(0, 4))$$

Nondeterminism from Parallel Writes

Memory

$mem(0) = 3$

$s(\mathbf{SET}(0,3), \mathbf{SET}(0,4))$

`true`

Nondeterminism from Parallel Writes

Memory

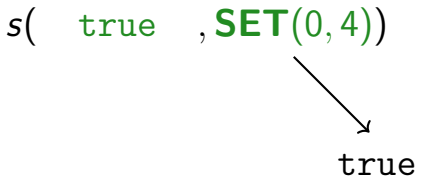
$mem(0) = 3$

$$s(\quad \texttt{true} \quad, \mathbf{SET}(0, 4))$$

Nondeterminism from Parallel Writes

Memory

$mem(0) = 4$



Nondeterminism from Parallel Writes

Memory

$mem(0) = 4$

$$s(\ \texttt{true}\ ,\ \texttt{true}\ )$$

Nondeterminism from Parallel Writes

Memory

$mem(0) = 0$

$$s(\mathbf{SET}(0, 3), \mathbf{SET}(0, 4))$$

Nondeterminism from Parallel Writes

Memory

$mem(0) = 4$

$$s(\mathbf{SET}(0,3), \mathbf{SET}(0,4)) \to \texttt{true}$$

Nondeterminism from Parallel Writes

Memory

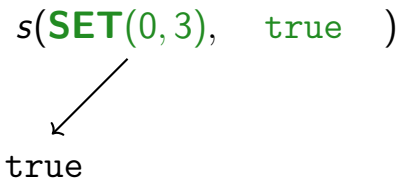
$mem(0) = 4$

$$s(\mathbf{SET}(0, 3), \quad \texttt{true} \quad)$$

Nondeterminism from Parallel Writes

Memory

$mem(0) = 3$



Nondeterminism from Parallel Writes

Memory

$mem(0) = 3$

$$s(\ \texttt{true}\ ,\ \texttt{true}\ )$$

Innermost Reduction Strategy

$\mathbf{s}(x, y) \to x$
$\mathbf{t}(x) \to x$

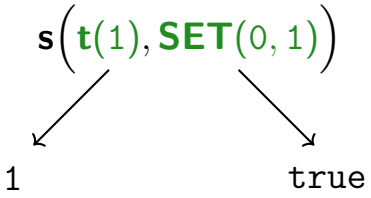
Memory
$mem(0) = 0$

$$\mathbf{s}\Big(\mathbf{t}(1), \mathbf{SET}(0, 1)\Big)$$

Innermost Reduction Strategy

$\mathbf{s}(x, y) \to x$
$\mathbf{t}(x) \to x$

Memory
$mem(0) = 0$



Innermost Reduction Strategy

$\mathbf{s}(x, y) \to x$
$\mathbf{t}(x) \to x$

Memory
$mem(0) = 1$

$$\mathbf{s}\Big(\ 1\ ,\ \ \texttt{true}\ \Big)$$

Innermost Reduction Strategy

$\mathbf{s}(x, y) \to x$
$\mathbf{t}(x) \to x$

Memory
$mem(0) = 1$

$$\mathbf{s}(\texttt{1}, \texttt{true}) \downarrow \texttt{1}$$

Innermost Reduction Strategy

$\mathbf{s}(x, y) \to x$
$\mathbf{t}(x) \to x$

Memory
$mem(0) = 1$

1

Outermost Reduction Strategy

Rules

$\mathbf{s}(x, y) \to x$
$\mathbf{t}(x) \to x$

Memory

$mem(0) = 0$

$$\mathbf{s}\Big(\mathbf{t}(1), \mathbf{SET}(0, 1)\Big)$$

Outermost Reduction Strategy

Rules

$\mathbf{s}(x, y) \to x$
$\mathbf{t}(x) \to x$

Memory

$mem(0) = 0$

$$\mathbf{s}\Big(\mathbf{t}(1), \mathbf{SET}(0, 1)\Big)$$

Outermost Reduction Strategy

Rules

$\mathbf{s}(x, y) \to x$
$\mathbf{t}(x) \to x$

Memory

$mem(0) = 0$

$$\mathbf{s}\Big(\mathbf{t}(1), \mathbf{SET}(0, 1)\Big)$$

$$\downarrow$$

$$\mathbf{t}(1)$$

Outermost Reduction Strategy

Rules

$\mathbf{s}(x, y) \to x$
$\mathbf{t}(x) \to x$

Memory

$mem(0) = 0$

$$\mathbf{t}(1)$$

Outermost Reduction Strategy

Rules

$\mathbf{s}(x, y) \to x$
$\mathbf{t}(x) \to x$

Memory

$mem(0) = 0$

$\mathbf{t}(1)$

Outermost Reduction Strategy

Rules

$\mathbf{s}(x, y) \to x$
$\mathbf{t}(x) \to x$

Memory

$mem(0) = 0$

$$\mathbf{t}(1)$$

$$\downarrow$$

$$1$$

Outermost Reduction Strategy

Rules

$\mathbf{s}(x, y) \to x$
$\mathbf{t}(x) \to x$

Memory

$mem(0) = 0$

1

Reflection: Design Evolution of Memory Operations

- Initial idea: one-address **READ**, **WRITE**

Reflection: Design Evolution of Memory Operations

- Initial idea: one-address **READ**, **WRITE**
- Too limited for realistic data structures

Reflection: Design Evolution of Memory Operations

- Initial idea: one-address **READ**, **WRITE**
- Too limited for realistic data structures
- Next idea: make **GET**, **SET**, and **CAS** logical/theory symbols

Reflection: Design Evolution of Memory Operations

- Initial idea: one-address **READ**, **WRITE**
- Too limited for realistic data structures
- Next idea: make **GET**, **SET**, and **CAS** logical/theory symbols
- Problem: they could appear inside rule constraints

Reflection: Design Evolution of Memory Operations

$$\mathbf{f}(x) \to \mathbf{g}(x)\ [\mathbf{GET}(0) = 1]$$

Reflection: Design Evolution of Memory Operations

$$\mathbf{f}(x) \to \mathbf{g}(x)\ [\mathbf{SET}(0,1) = \mathtt{true} \wedge x > 0]$$

Reflection: Design Evolution of Memory Operations

$$\mathbf{f}(x) \to \mathbf{g}(x)\ [\mathbf{SET}(0, 1) = \mathtt{true} \wedge x > 0]$$

Reflection: Design Evolution of Memory Operations

- Initial idea: one-address **READ**, **WRITE**
- Too limited for realistic data structures
- Next idea: make **GET**, **SET**, and **CAS** logical/theory symbols
- Problem: they could appear inside rule constraints
- Final lesson: memory operations should be explicit memory-reduction steps

MemTRS in Cora: Prototype REPL

```
Welcome to MemTRS REPL v0.1

memtrs>
```

MemTRS in Cora: Prototype REPL

```
Welcome to MemTRS REPL v0.1

memtrs> include memtrs/stdlib/mem_ds/array.lctrs
OK
memtrs>
```

MemTRS in Cora: Prototype REPL

```
Welcome to MemTRS REPL v0.1

memtrs> include memtrs/stdlib/mem_ds/array.lctrs
OK
memtrs> include memtrs/stdlib/algorithms/quicksort.lctrs
OK
memtrs>
```

MemTRS in Cora: Prototype REPL

```
Welcome to MemTRS REPL v0.1

memtrs> include memtrs/stdlib/mem_ds/array.lctrs
OK
memtrs> include memtrs/stdlib/algorithms/quicksort.lctrs
OK
memtrs> reduce listToArray(cons(3, cons(1, cons(5, cons(2, cons(4, nil))))))
Reduced to:  2
memtrs>
```

MemTRS in Cora: Prototype REPL

```
Welcome to MemTRS REPL v0.1

memtrs> include memtrs/stdlib/mem_ds/array.lctrs
OK
memtrs> include memtrs/stdlib/algorithms/quicksort.lctrs
OK
memtrs> reduce listToArray(cons(3, cons(1, cons(5, cons(2, cons(4, nil))))))
Reduced to:  2
memtrs> print_mem
Memory:
[8, 0, 5, 3, 1, 5, 2, 4, ...]
memtrs>
```

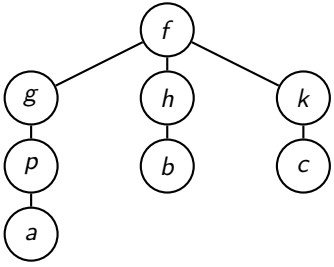
MemTRS in Cora: Prototype REPL

```
Welcome to MemTRS REPL v0.1

memtrs> include memtrs/stdlib/mem_ds/array.lctrs
OK
memtrs> include memtrs/stdlib/algorithms/quicksort.lctrs
OK
memtrs> reduce listToArray(cons(3, cons(1, cons(5, cons(2, cons(4, nil))))))
Reduced to:  2
memtrs> print_mem
Memory:
[8, 0, 5, 3, 1, 5, 2, 4, ...]
memtrs> reduce qsortArray(2, 0, 4)
Reduced to:  true
memtrs>
```

MemTRS in Cora: Prototype REPL

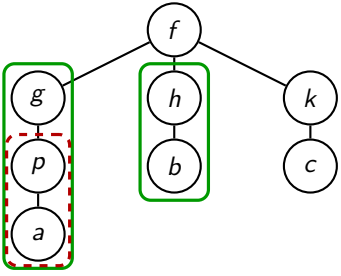
```
Welcome to MemTRS REPL v0.1

memtrs> include memtrs/stdlib/mem_ds/array.lctrs
OK
memtrs> include memtrs/stdlib/algorithms/quicksort.lctrs
OK
memtrs> reduce listToArray(cons(3, cons(1, cons(5, cons(2, cons(4, nil))))))
Reduced to:  2
memtrs> print_mem
Memory:
[8, 0, 5, 3, 1, 5, 2, 4, ...]
memtrs> reduce qsortArray(2, 0, 4)
Reduced to:  true
memtrs> print_mem
Memory:
[8, 0, 5, 1, 2, 3, 4, 5, ...]
memtrs>
```

Parallel Rewriting in Cora



$f(g(p(a)), h(b), k(c))$

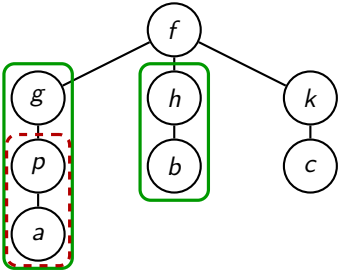
Parallel Rewriting in Cora

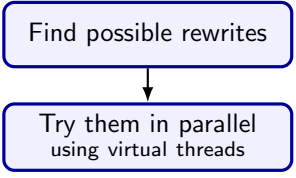


Find possible rewrites

$f(g(p(a)), h(b), k(c))$

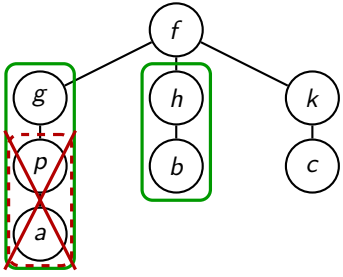
Parallel Rewriting in Cora

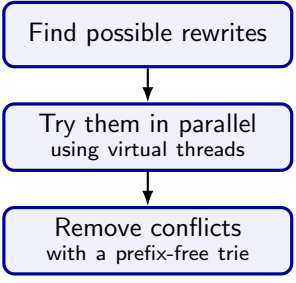




$f(g(p(a)), h(b), k(c))$

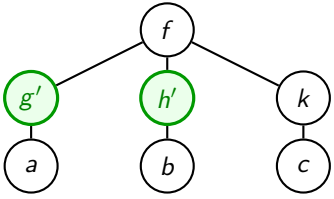
Parallel Rewriting in Cora



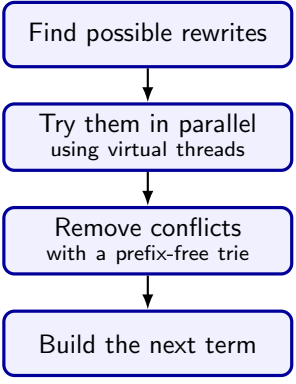


$f(g(p(a)), h(b), k(c))$

Parallel Rewriting in Cora



$$f(g(p(a)), h(b), k(c)) \rightrightarrows f(g'(a), h'(b), k(c))$$



Reflection: Implementation Trade-offs in Cora

- Early memory implementations were either **unsafe** or **inefficient**

Reflection: Implementation Trade-offs in Cora

- Early memory implementations were either **unsafe** or **inefficient**
- `ConcurrentMap` solved thread safety, but added too much **overhead**

Reflection: Implementation Trade-offs in Cora

- Early memory implementations were either **unsafe** or **inefficient**
- `ConcurrentMap` solved thread safety, but added too much **overhead**
- `AtomicIntegerArray` was less flexible, but more **predictable** and **efficient**

Reflection: Implementation Trade-offs in Cora

- Early memory implementations were either **unsafe** or **inefficient**
- `ConcurrentMap` solved thread safety, but added too much **overhead**
- `AtomicIntegerArray` was less flexible, but more **predictable** and **efficient**
- Parallel reduction had to be corrected to rewrite only **disjoint** redexes

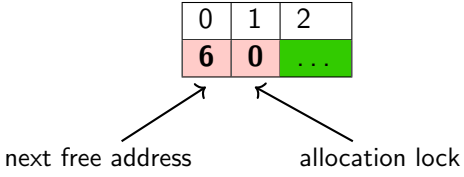
Reflection: Implementation Trade-offs in Cora

- Early memory implementations were either **unsafe** or **inefficient**
- `ConcurrentMap` solved thread safety, but added too much **overhead**
- `AtomicIntegerArray` was less flexible, but more **predictable** and **efficient**
- Parallel reduction had to be corrected to rewrite only **disjoint** redexes
- **Virtual threads** improved the practical behavior of parallel rewriting

Reflection: Implementation Trade-offs in Cora

- Early memory implementations were either **unsafe** or **inefficient**
- `ConcurrentMap` solved thread safety, but added too much **overhead**
- `AtomicIntegerArray` was less flexible, but more **predictable** and **efficient**
- Parallel reduction had to be corrected to rewrite only **disjoint** redexes
- **Virtual threads** improved the practical behavior of parallel rewriting
- The experiments reflect the **Cora-based implementation**, not only abstract MemTRS model

MemTRS: Global Memory Model



Memory Reduction: Parallel Allocation I

Memory

0	1	2	3	4
2	0	...	...	...

$s($ **alloc**(1) , **alloc**(1) $)$

Memory Reduction: Parallel Allocation I

Memory

0	1	2	3	4
2	0	...	...	...

$s($ **alloc**(1) , **alloc**(1) $)$

allocLock$(1, \textbf{CAS}(1, 0, 1))$ **allocLock**$(1, \textbf{CAS}(1, 0, 1))$

Memory Reduction: Parallel Allocation I

Memory

0	1	2	3	4
2	0	...	...	...

$$s(\mathbf{allocLock}(1, \mathbf{CAS}(1, 0, 1)), \mathbf{allocLock}(1, \mathbf{CAS}(1, 0, 1)))$$

Memory Reduction: Parallel Allocation I

Memory

0	1	2	3	4
2	0	...	...	...

$s(\mathbf{allocLock}(1, \mathbf{CAS}(1,0,1)), \mathbf{allocLock}(1, \mathbf{CAS}(1,0,1)))$

Memory Reduction: Parallel Allocation I

Memory

0	1	2	3	4
2	**1**	...	...	...

$$s(\mathbf{allocLock}(1, \texttt{true}), \mathbf{allocLock}(1, \texttt{false}))$$

Memory Reduction: Parallel Allocation I

Memory

0	1	2	3	4
2	**1**	...	...	...

$s($ **allocLock**$(1, \texttt{true})$, **allocLock**$(1, \texttt{false})$ $)$

allocRead$(1,$ **GET**$(0))$ **alloc**(1)

Memory Reduction: Parallel Allocation I

Memory

0	1	2	3	4
2	**1**	...	...	...

$s(\mathbf{allocRead}(1, \mathbf{GET}(0)),$ $\mathbf{alloc}(1)$ $)$

Memory Reduction: Parallel Allocation I

Memory

0	1	2	3	4
2	**1**	...	...	...

Memory Reduction: Parallel Allocation I

Memory

0	1	2	3	4
2	**1**	...	...	...

$$s(\mathbf{allocRead}(1, \quad 2 \quad), \mathbf{allocLock}(1, \mathbf{CAS}(1, 0, 1)))$$

Memory Reduction: Parallel Allocation I

Memory

0	1	2	3	4
2	**1**	...	...	...

$s($ **allocRead**$(1, 2)$, **allocLock**$(1,$ **CAS**$(1, 0, 1)))$

allocReserve$(2,$ **SET**$(0, 2 + 1))$

`false`

Memory Reduction: Parallel Allocation I

Memory

0	1	2	3	4
2	**1**	...	...	...

$$s(\textbf{allocReserve}(2, \textbf{SET}(0, 2+1)), \textbf{allocLock}(1, \texttt{false}))$$

Memory Reduction: Parallel Allocation I

Memory

0	1	2	3	4
2	**1**	...	...	...

$s(\textbf{allocReserve}(2, \textbf{SET}(0, 2+1)),$ $\textbf{allocLock}(1, \texttt{false})$ $)$

3

$\textbf{alloc}(1)$

Memory Reduction: Parallel Allocation I

Memory

0	1	2	3	4
2	**1**	...	...	...

$s($**allocReserve**$(2,$ **SET**$(0,\ 3\ )),$ **alloc**$(1)\ )$

Memory Reduction: Parallel Allocation I

Memory

0	1	2	3	4
2	**1**	...	...	...

$s(\textbf{allocReserve}(2, \textbf{SET}(0,3)),$ $\textbf{alloc}(1)$ $)$

$\texttt{true}$

$\textbf{allocLock}(1, \textbf{CAS}(1,0,1))$

Memory Reduction: Parallel Allocation I

Memory

0	1	2	3	4
3	**1**	...	...	...

$$s(\mathbf{allocReserve}(2,\ \texttt{true}\ ), \mathbf{allocLock}(1, \mathbf{CAS}(1, 0, 1)))$$

Memory Reduction: Parallel Allocation I

Memory

0	1	2	3	4
3	**1**	...	...	...

$s($ **allocReserve**$(2, \texttt{true})$, **allocLock**$(1,$ **CAS**$(1, 0, 1)))$

allocUnlock$(2,$ **SET**$(1, 0))$

`false`

Memory Reduction: Parallel Allocation I

Memory

0	1	2	3	4
3	**1**	...	...	...

$$s(\mathbf{allocUnlock}(2, \mathbf{SET}(1, 0)), \mathbf{allocLock}(1, \texttt{false}))$$

Memory Reduction: Parallel Allocation I

Memory

0	1	2	3	4
3	**1**	...	...	...

$s($**allocUnlock**$(2,$ **SET**$(1,0)),$ **allocLock**$(1,$ `false`$)$ $)$

↓ `true`

↓ **alloc**(1)

Memory Reduction: Parallel Allocation I

Memory

0	1	2	3	4
3	**0**	...	...	...

$s($**allocUnlock**$(2,$ `true` $),$ **alloc**(1) $)$

Memory Reduction: Parallel Allocation I

Memory

0	1	2	3	4
3	**0**	...	...	...

$s($**allocUnlock**$(2, \texttt{true}),$ **alloc**(1) $)$

2

allocLock$(1,$ **CAS**$(1, 0, 1))$

Memory Reduction: Parallel Allocation I

Memory

0	1	2	3	4
3	**0**	...	...	...

$s($ 2 , $\mathbf{allocLock}(1, \mathbf{CAS}(1, 0, 1)))$

Memory Reduction: Parallel Allocation II

Memory

0	1	2	3	4
3	0	...	...	...

$$s(2, \textbf{allocLock}(1, \textbf{CAS}(1, 0, 1)))$$

Memory Reduction: Parallel Allocation II

Memory

0	1	2	3	4
3	0	...	...	...

$$s(2, \textbf{allocLock}(1, \textbf{CAS}(1, 0, 1))) \downarrow \texttt{true}$$

Memory Reduction: Parallel Allocation II

Memory

0	1	2	3	4
3	**1**	...	...	...

$s(2, \textbf{allocLock}(1, \quad \texttt{true} \quad))$

Memory Reduction: Parallel Allocation II

Memory

0	1	2	3	4
3	**1**	...	...	...

$$s(2,\ \textbf{allocLock}(1, \texttt{true})\ ) \downarrow \textbf{allocRead}(1, \textbf{GET}(0))$$

Memory Reduction: Parallel Allocation II

Memory

0	1	2	3	4
3	**1**	...	...	...

$$s(2, \textbf{allocRead}(1, \textbf{GET}(0)))$$

Memory Reduction: Parallel Allocation II

Memory

0	1	2	3	4
3	**1**	...	...	...

$$s(2, \textbf{allocRead}(1, \textbf{GET}(0)))$$

$$\downarrow$$

$$3$$

Memory Reduction: Parallel Allocation II

Memory

0	1	2	3	4
3	**1**	...	...	...

$$s(2, \textbf{allocRead}(1, \quad 3 \quad))$$

Memory Reduction: Parallel Allocation II

Memory

0	1	2	3	4
3	**1**	...	...	...

$$s(2, \quad \textbf{allocRead}(1, 3) \quad)$$

$$\downarrow$$

$$\textbf{allocReserve}(3, \textbf{SET}(0, 3 + 1))$$

Memory Reduction: Parallel Allocation II

Memory

0	1	2	3	4
3	**1**	...	...	...

$$s(2, \textbf{allocReserve}(3, \textbf{SET}(0, 3 + 1)))$$

Memory Reduction: Parallel Allocation II

Memory

0	1	2	3	4
3	**1**	...	...	...

$$s(2, \textbf{allocReserve}(3, \textbf{SET}(0, 3 + 1)))$$

$$\downarrow$$

$$4$$

Memory Reduction: Parallel Allocation II

Memory

0	1	2	3	4
3	**1**	...	...	...

$$s(2, \textbf{allocReserve}(3, \textbf{SET}(0, \quad 4 \quad)))$$

Memory Reduction: Parallel Allocation II

Memory

0	1	2	3	4
3	**1**	...	...	...

$$s(2, \textbf{allocReserve}(3, \textbf{SET}(0, 4)))$$

$$\downarrow$$

`true`

Memory Reduction: Parallel Allocation II

Memory

0	1	2	3	4
4	**1**	...	...	...

$$s(2, \textbf{allocReserve}(3, \texttt{true}))$$

Memory Reduction: Parallel Allocation II

Memory

0	1	2	3	4
4	**1**	...	...	...

$$s(2,\ \textbf{allocReserve}(3, \texttt{true})\ ) \downarrow \textbf{allocUnlock}(3, \textbf{SET}(1, 0))$$

Memory Reduction: Parallel Allocation II

Memory

0	1	2	3	4
4	**1**	...	...	...

$$s(2, \textbf{allocUnlock}(3, \textbf{SET}(1, 0)))$$

Memory Reduction: Parallel Allocation II

Memory

0	1	2	3	4
4	**1**	...	...	...

$$s(2, \textbf{allocUnlock}(3, \textbf{SET}(1, 0))) \downarrow \texttt{true}$$

Memory Reduction: Parallel Allocation II

Memory

0	1	2	3	4
4	**0**	...	...	...

$s(2, \textbf{allocUnlock}(3, \texttt{true}))$

Memory Reduction: Parallel Allocation II

Memory

0	1	2	3	4
4	**0**	...	...	...

$$s(2, \textbf{allocUnlock}(3, \texttt{true}))$$

$$\downarrow$$

$$3$$

Memory Reduction: Parallel Allocation II

Memory

0	1	2	3	4
4	**0**	...	...	...

$s(2, \quad 3 \quad)$

Memory Reduction: Parallel Allocation II

Memory

0	1	2	3	4
4	**0**	...	...	...

$s(2, 3)$

Reflection: Memory Gives Expressiveness, but Adds Low-Level Detail

- Shared memory made arrays, matrices, queues, and graph algorithms **easier** to model

Reflection: Memory Gives Expressiveness, but Adds Low-Level Detail

- Shared memory made arrays, matrices, queues, and graph algorithms **easier** to model
- But the model became more **explicit** and **lower-level**

Reflection: Memory Gives Expressiveness, but Adds Low-Level Detail

- Shared memory made arrays, matrices, queues, and graph algorithms **easier** to model
- But the model became more **explicit** and **lower-level**
- I had to **reason** about addresses, allocation, evaluation order, and synchronization

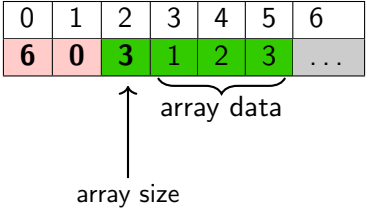
Reflection: Memory Gives Expressiveness, but Adds Low-Level Detail

- Shared memory made arrays, matrices, queues, and graph algorithms **easier** to model
- But the model became more **explicit** and **lower-level**
- I had to **reason** about addresses, allocation, evaluation order, and synchronization
- Some algorithms and procedures started to look closer to **imperative programs** encoded as rewrite rules

Reflection: Memory Gives Expressiveness, but Adds Low-Level Detail

- Shared memory made arrays, matrices, queues, and graph algorithms **easier** to model
- But the model became more **explicit** and **lower-level**
- I had to **reason** about addresses, allocation, evaluation order, and synchronization
- Some algorithms and procedures started to look closer to **imperative programs** encoded as rewrite rules
- This raises the question: can MemTRS keep the declarative strengths of rewriting?

Arrays in MemTRS



Arrays in MemTRS

$$[7, 2, 10, 4, 1, 9, 3, 6, 8, 5]$$

Arrays in MemTRS

0	1	2	3	4	5	6	7	8	9	10	11	12
13	**0**	**10**	7	2	10	4	1	9	3	6	8	5

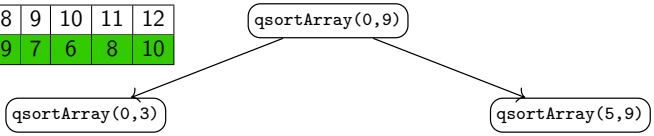
QuickSort with Memory

0	1	2	3	4	5	6	7	8	9	10	11	12
13	**0**	**10**	7	2	10	4	1	9	3	6	8	5

`qsortArray(0,9)`

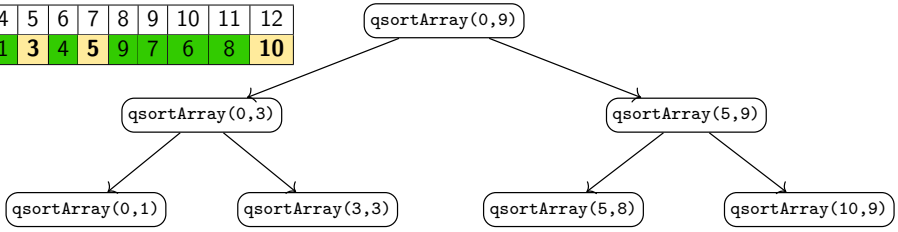
QuickSort with Memory

0	1	2	3	4	5	6	7	8	9	10	11	12
13	**0**	**10**	2	4	1	3	**5**	9	7	6	8	10



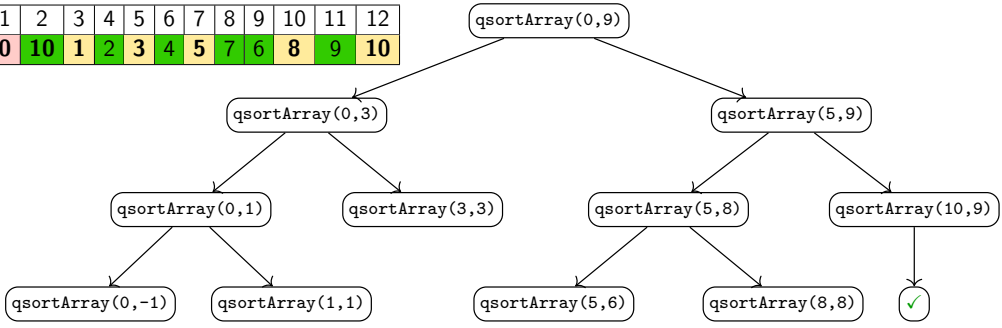
QuickSort with Memory

0	1	2	3	4	5	6	7	8	9	10	11	12
13	**0**	**10**	2	1	**3**	4	**5**	9	7	6	8	**10**



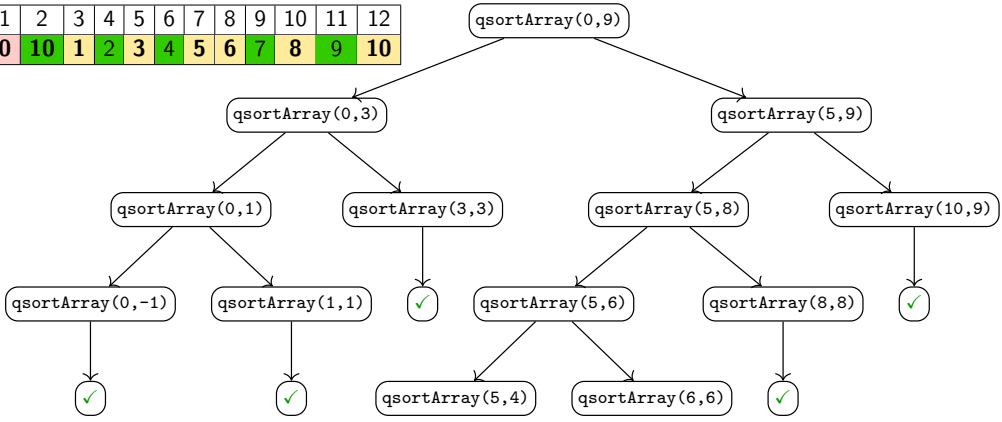
QuickSort with Memory

0	1	2	3	4	5	6	7	8	9	10	11	12
13	**0**	**10**	**1**	**2**	**3**	**4**	**5**	**7**	**6**	**8**	**9**	**10**



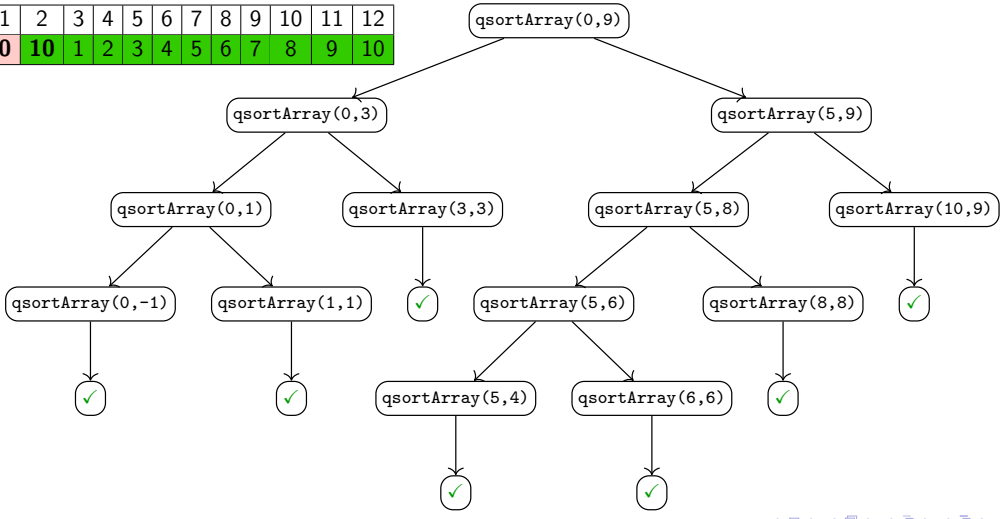
QuickSort with Memory

0	1	2	3	4	5	6	7	8	9	10	11	12
13	**0**	**10**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**

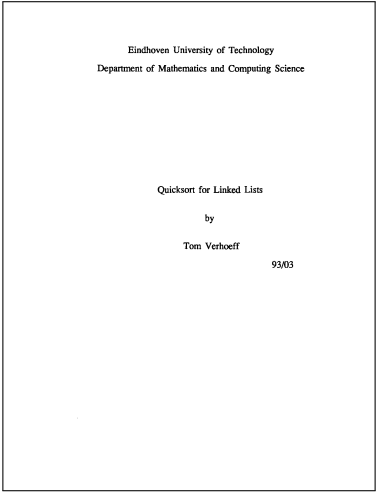


QuickSort with Memory

0	1	2	3	4	5	6	7	8	9	10	11	12
13	**0**	**10**	1	2	3	4	5	6	7	8	9	10



QuickSort without Memory: Comparison Baseline



Eindhoven University of Technology

Department of Mathematics and Computing Science

Quicksort for Linked Lists

by

Tom Verhoeff

93/03

QuickSort without Memory: Comparison Baseline

Input: **qsortList**$([7, 2, 10, 4, 1, 9, 3, 6, 8, 5], \texttt{nil})$

QuickSort without Memory: Comparison Baseline

Input: $\mathbf{qsortList}([7, 2, 10, 4, 1, 9, 3, 6, 8, 5], \mathtt{nil})$

First partition:

pivot: **7**
left: $[5, 6, 3, 1, 4, 2]$
right: $[8, 9, 10]$

QuickSort without Memory: Comparison Baseline

Input: $\mathbf{qsortList}([7, 2, 10, 4, 1, 9, 3, 6, 8, 5], \mathtt{nil})$

First partition:

pivot: **7**
left: $[5, 6, 3, 1, 4, 2]$
right: $[8, 9, 10]$

Rewritten term: $\mathbf{qsortList}([5, 6, 3, 1, 4, 2], 7 :: \mathbf{qsortList}([8, 9, 10], \mathtt{nil}))$

QuickSort without Memory: Comparison Baseline

Input: $\mathbf{qsortList}([7, 2, 10, 4, 1, 9, 3, 6, 8, 5], \mathtt{nil})$

First partition:

pivot: **7**
left: $[5, 6, 3, 1, 4, 2]$
right: $[8, 9, 10]$

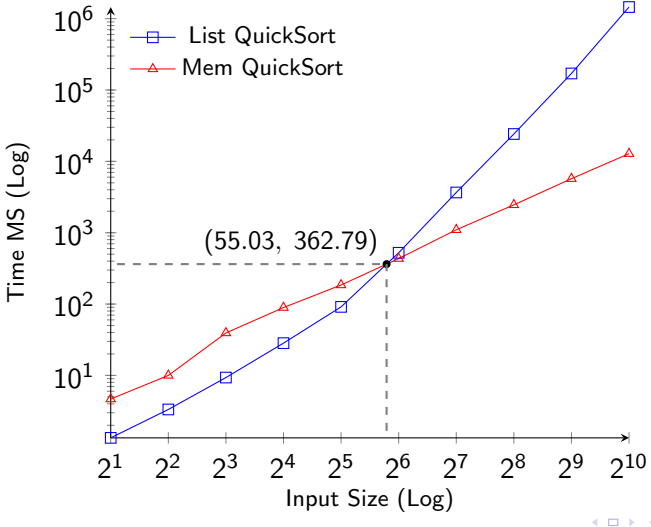
Rewritten term: $\mathbf{qsortList}([5, 6, 3, 1, 4, 2], 7 :: \mathbf{qsortList}([8, 9, 10], \mathtt{nil}))$

Reduced to: $[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$

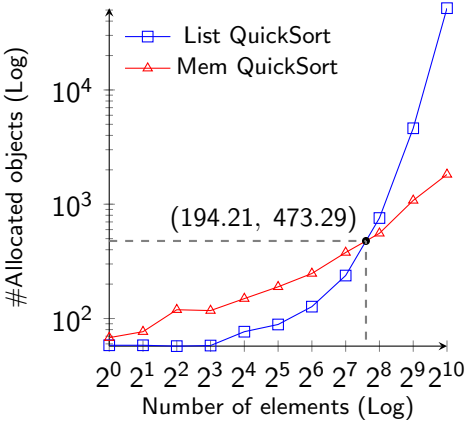
QuickSort Case Study: Experiment Setup

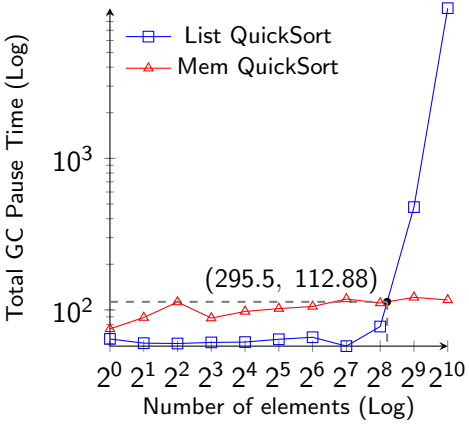
- Comparing two implementations of QuickSort (MemTRS vs LCTRS)
- Both implementations have average runtime complexity of $\mathcal{O}(n \log n)$
- Randomly generated integer arrays
- Input sizes: 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024
- Repeated 3 times each

QuickSort Case Study: Runtime Results



QuickSort Case Study: Profiling with JFR





QuickSort Case Study: CPU-Time Profiling

List QuickSort

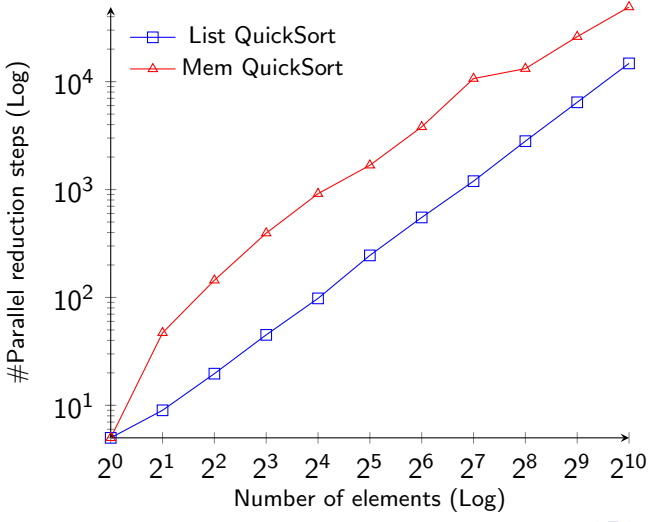
#	Thread	Samples	Percent
1	main	51,010	65.58%
2	*virtual*	13,992	17.99%
3–14	ForkJoinPool workers	12,716	14.89%
15–19	Other	68	0.09%

QuickSort Case Study: CPU-Time Profiling

Memory QuickSort

#	Thread	Samples	Percent
1	*virtual*	8,692	63.55%
2	main	2,394	17.50%
3–14	ForkJoinPool workers	2578	18.86%
15–16	Other	13	0.10%

QuickSort Case Study: Parallel Reduction Steps



QuickSort Case Study: Interpretation of Results

- List QuickSort is faster for **small** inputs

QuickSort Case Study: Interpretation of Results

- List QuickSort is faster for **small** inputs
- Memory QuickSort performs better for **larger** inputs

QuickSort Case Study: Interpretation of Results

- List QuickSort is faster for **small** inputs
- Memory QuickSort performs better for **larger** inputs
- Both versions show **similar** asymptotic growth in parallel reduction steps

QuickSort Case Study: Interpretation of Results

- List QuickSort is faster for **small** inputs
- Memory QuickSort performs better for **larger** inputs
- Both versions show **similar** asymptotic growth in parallel reduction steps
- Main difference comes from **engine-level overhead**

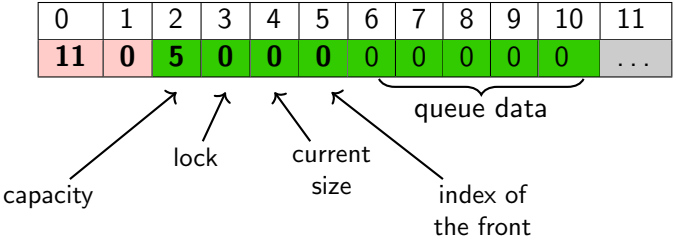
QuickSort Case Study: Interpretation of Results

- List QuickSort is faster for **small** inputs
- Memory QuickSort performs better for **larger** inputs
- Both versions show **similar** asymptotic growth in parallel reduction steps
- Main difference comes from **engine-level overhead**
- List QuickSort stores input inside **large terms**, causing expensive subterm processing

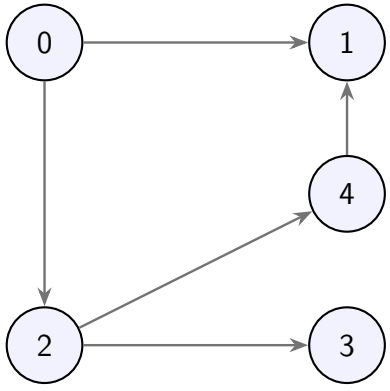
QuickSort Case Study: Interpretation of Results

- List QuickSort is faster for **small** inputs
- Memory QuickSort performs better for **larger** inputs
- Both versions show **similar** asymptotic growth in parallel reduction steps
- Main difference comes from **engine-level overhead**
- List QuickSort stores input inside **large terms**, causing expensive subterm processing
- Memory QuickSort keeps array data in global memory, **reducing** allocation pressure

Concurrent Queues in MemTRS: Using Synchronization Primitives



Parallel Breadth-First Search

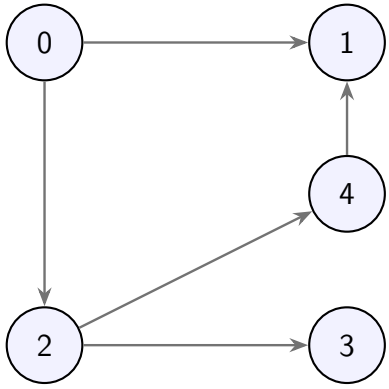


visited queue @ 8 queue @ 38 worker

Init heap metadata

Address	Value
0	**2**
1	**0**

Parallel Breadth-First Search

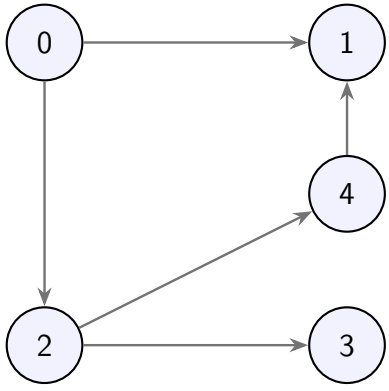


visited queue @ 8 queue @ 38 worker

Allocate visited array

Address	Value
0	8
1	0
2	**5**
3–7	**0, 0, 0, 0, 0**

Parallel Breadth-First Search

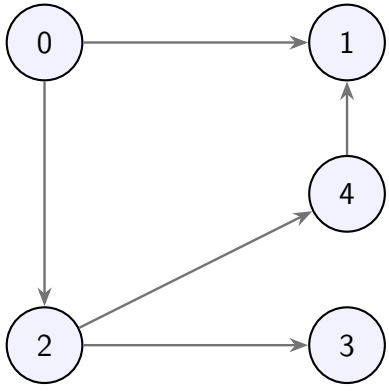


visited queue @ 8 queue @ 38 worker

Allocate queue @ 8

Address	Value
0	**38**
1	0
2	5
3–7	0, 0, 0, 0, 0
8–11	**26, 0, 0, 0**
12	**0**
13	**0**
14	**0**
15–37	**0**, . . . , **0**

Parallel Breadth-First Search

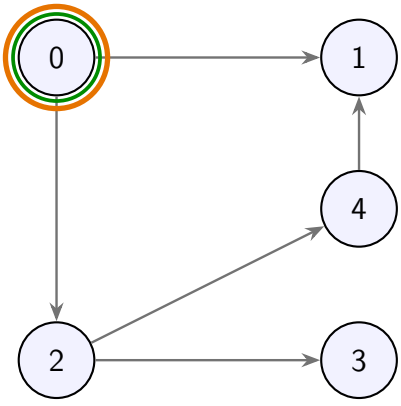


visited queue @ 8 queue @ 38 worker

Allocate queue @ 38

Address	Value
0	**68**
1	0
2	5
3–7	0, 0, 0, 0, 0
8–11	26, 0, 0, 0
12	0
13	0
14	0
15–37	0, . . . , 0
38–41	**26, 0, 0, 0**
42	**0**
43	**0**
44–67	**0**, . . . , **0**

Parallel Breadth-First Search

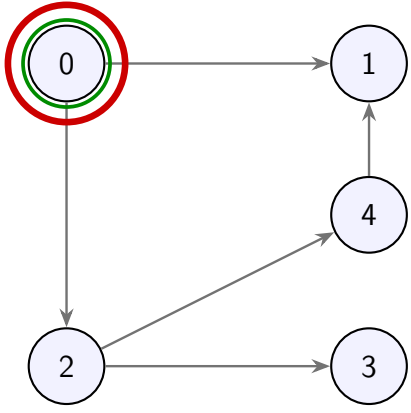


visited queue @ 8 queue @ 38 worker

Seed: visit 0, push 0 to Q8

Address	Value
0	68
1	0
2	5
3–7	**1**, 0, 0, 0, 0
8–11	26, 0, **1**, 0
12	**0**
13	0
14	0
15–37	0, . . . , 0
38–41	26, 0, 0, 0
42	0
43	0
44–67	0, . . . , 0

Parallel Breadth-First Search

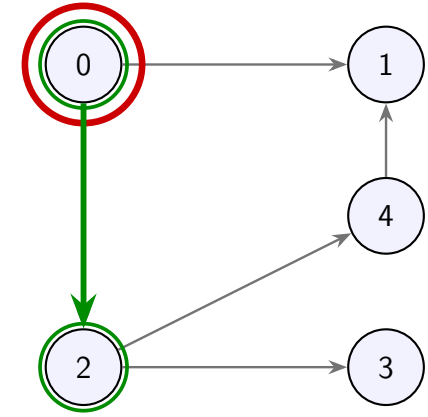


visited queue @ 8 queue @ 38 worker

Pop 0 from Q8

Address	Value
0	68
1	0
2	5
3–7	1, 0, 0, 0, 0
8–11	26, 0, **0**, **1**
12	0
13	0
14	0
15–37	0, . . . , 0
38–41	26, 0, 0, 0
42	0
43	0
44–67	0, . . . , 0

Parallel Breadth-First Search

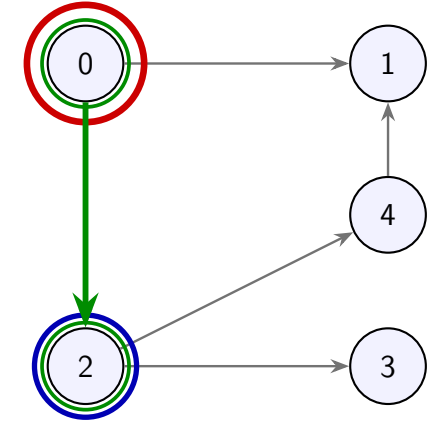


visited queue @ 8 queue @ 38 worker

Discover 2: set visited

Address	Value
0	68
1	0
2	5
3–7	1, 0, **1**, 0, 0
8–11	26, 0, 0, 1
12	0
13	0
14	0
15–37	0, . . . , 0
38–41	26, 0, 0, 0
42	0
43	0
44–67	0, . . . , 0

Parallel Breadth-First Search

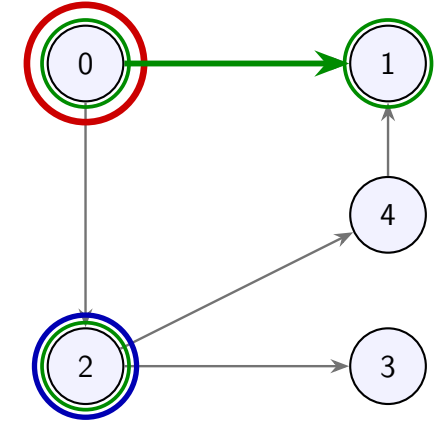


visited queue @ 8 queue @ 38 worker

Push 2 to Q38

Address	Value
0	68
1	0
2	5
3–7	1, 0, 1, 0, 0
8–11	26, 0, 0, 1
12	0
13	0
14	0
15–37	0, . . . , 0
38–41	26, 0, **1**, 0
42	**2**
43	0
44–67	0, . . . , 0

Parallel Breadth-First Search

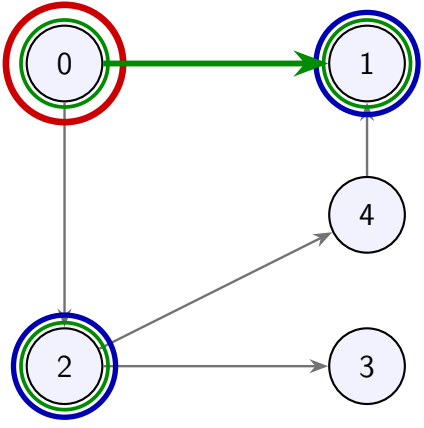


visited queue @ 8 queue @ 38 worker

Discover 1: set visited

Address	Value
0	68
1	0
2	5
3–7	1, **1**, 1, 0, 0
8–11	26, 0, 0, 1
12	0
13	0
14	0
15–37	0, . . . , 0
38–41	26, 0, 1, 0
42	2
43	0
44–67	0, . . . , 0

Parallel Breadth-First Search

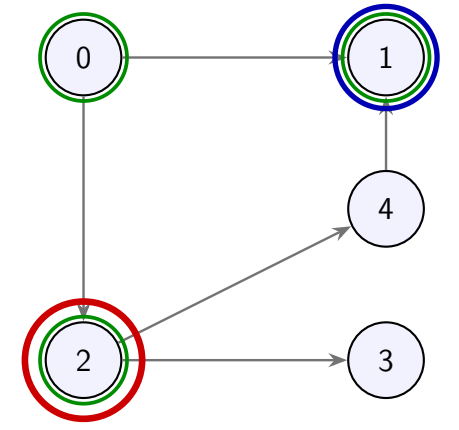


visited queue @ 8 queue @ 38 worker

Push 1 to Q38

Address	Value
0	68
1	0
2	5
3–7	1, 1, 1, 0, 0
8–11	26, 0, 0, 1
12	0
13	0
14	0
15–37	0, . . . , 0
38–41	26, 0, **2**, 0
42	2
43	**1**
44–67	0, . . . , 0

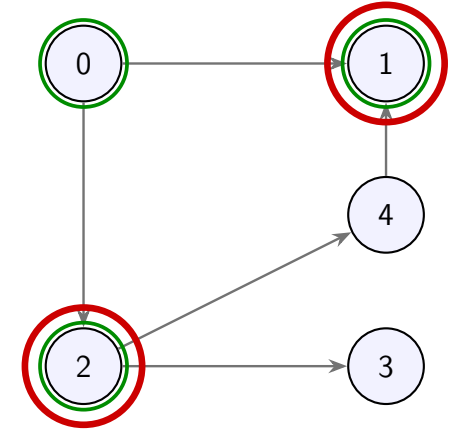
Parallel Breadth-First Search



Pop 2 from Q38

Address	Value
0	68
1	0
2	5
3–7	1, 1, 1, 0, 0
8–11	26, 0, 0, 1
12	0
13	0
14	0
15–37	0, . . . , 0
38–41	26, 0, **1**, **1**
42	2
43	1
44–67	0, . . . , 0

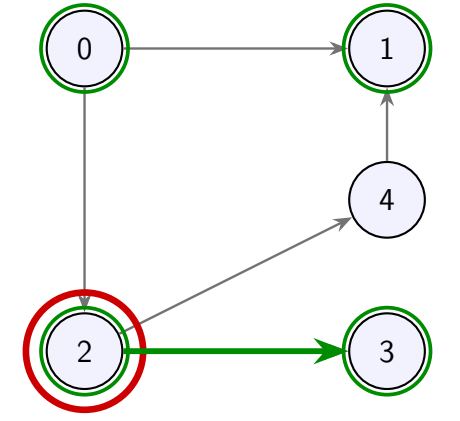
Parallel Breadth-First Search



Pop 1 from Q38

Address	Value
0	68
1	0
2	5
3–7	1, 1, 1, 0, 0
8–11	26, 0, 0, 1
12	0
13	0
14	0
15–37	0, . . . , 0
38–41	26, 0, **0**, **2**
42	2
43	1
44–67	0, . . . , 0

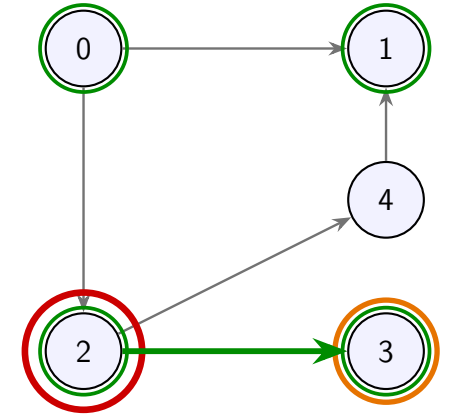
Parallel Breadth-First Search



Discover 3: set visited

Address	Value
0	68
1	0
2	5
3–7	1, 1, 1, **1**, 0
8–11	26, 0, 0, 1
12	0
13	0
14	0
15–37	0, . . . , 0
38–41	26, 0, 0, 2
42	2
43	1
44–67	0, . . . , 0

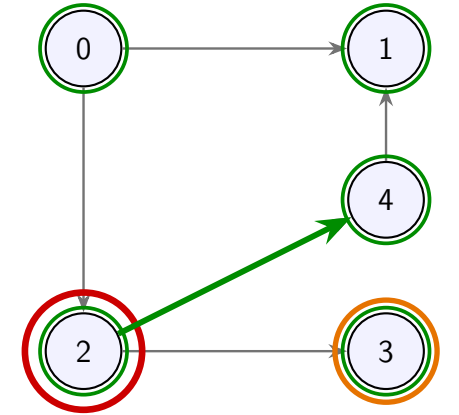
Parallel Breadth-First Search



Push 3 to Q8

Address	Value
0	68
1	0
2	5
3–7	1, 1, 1, 1, 0
8–11	26, 0, **1**, 1
12	0
13	**3**
14	0
15–37	0, . . . , 0
38–41	26, 0, 0, 2
42	2
43	1
44–67	0, . . . , 0

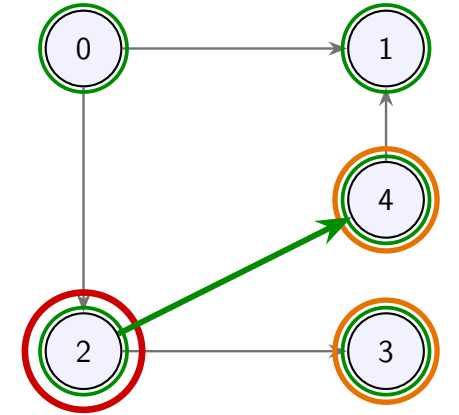
Parallel Breadth-First Search



Discover 4: set visited

Address	Value
0	68
1	0
2	5
3–7	1, 1, 1, 1, **1**
8–11	26, 0, 1, 1
12	0
13	3
14	0
15–37	0, . . . , 0
38–41	26, 0, 0, 2
42	2
43	1
44–67	0, . . . , 0

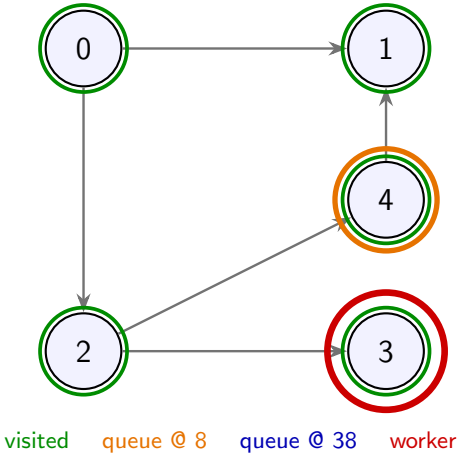
Parallel Breadth-First Search



Push 4 to Q8

Address	Value
0	68
1	0
2	5
3–7	1, 1, 1, 1, 1
8–11	26, 0, **2**, 1
12	0
13	3
14	**4**
15–37	0, . . . , 0
38–41	26, 0, 0, 2
42	2
43	1
44–67	0, . . . , 0

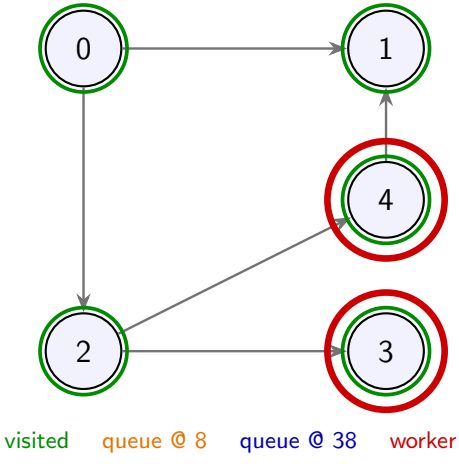
Parallel Breadth-First Search



Pop 3 from Q8

Address	Value
0	68
1	0
2	5
3–7	1, 1, 1, 1, 1
8–11	26, 0, **1**, **2**
12	0
13	3
14	4
15–37	0, . . . , 0
38–41	26, 0, 0, 2
42	2
43	1
44–67	0, . . . , 0

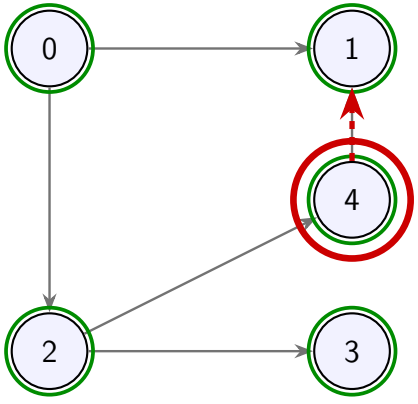
Parallel Breadth-First Search



Pop 4 from Q8

Address	Value
0	68
1	0
2	5
3–7	1, 1, 1, 1, 1
8–11	26, 0, **0**, **3**
12	0
13	3
14	4
15–37	0, . . . , 0
38–41	26, 0, 0, 2
42	2
43	1
44–67	0, . . . , 0

Parallel Breadth-First Search

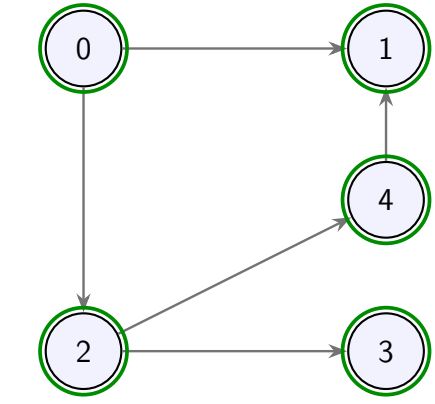


visited queue @ 8 queue @ 38 worker

$4 \to 1$ fails: 1 already visited

Address	Value
0	68
1	0
2	5
3–7	1, 1, 1, 1, 1
8–11	26, 0, 0, 3
12	0
13	3
14	4
15–37	$0, \ldots, 0$
38–41	26, 0, 0, 2
42	2
43	1
44–67	$0, \ldots, 0$

Parallel Breadth-First Search



visited queue @ 8 queue @ 38 worker

Done!

Address	Value
0	68
1	0
2	5
3–7	1, 1, 1, 1, 1
8–11	26, 0, 0, 3
12	0
13	3
14	4
15–37	0, . . . , 0
38–41	26, 0, 0, 2
42	2
43	1
44–67	0, . . . , 0

Sequential Breadth-First Search: Comparison Baseline

- Graph as nested linked lists

$$\begin{aligned}&\mathbf{consG}(\mathbf{cons}(1, \mathbf{cons}(2, \mathtt{nil})),\\ &\quad \mathbf{consG}(\mathtt{nil},\\ &\quad \mathbf{consG}(\mathbf{cons}(3, \mathbf{cons}(4, \mathtt{nil})),\\ &\quad \mathbf{consG}(\mathtt{nil},\\ &\quad \mathbf{consG}(\mathbf{cons}(1, \mathtt{nil}),\\ &\mathtt{nilG})))))\end{aligned}$$

Sequential Breadth-First Search: Comparison Baseline

- Graph as nested linked lists
- Visited array as a linked list

$$\mathbf{cons}(1, \mathbf{cons}(0, \mathbf{cons}(1, \mathbf{cons}(0, \mathbf{cons}(0, \mathtt{nil})))))$$

Sequential Breadth-First Search: Comparison Baseline

- Graph as nested linked lists
- Visited array as a linked list
- Queue as a linked-list term

$$\mathbf{cons}(2, \mathbf{cons}(2, \mathbf{cons}(1, \mathtt{nil})))$$

Sequential Breadth-First Search: Comparison Baseline

- Graph as nested linked lists
- Visited array as a linked list
- Queue as a linked-list term
- Linear **pushBack** traversal

$$\mathbf{pushBack}(e, \mathbf{cons}(u, l)) \to \mathbf{cons}(u, \mathbf{pushBack}(e, l))$$

$$\mathbf{pushBack}(e, \mathtt{nil}) \to \mathbf{cons}(e, \mathtt{nil})$$

Sequential Breadth-First Search: Comparison Baseline

- Graph as nested linked lists
- Visited array as a linked list
- Queue as a linked-list term
- Linear **pushBack** traversal
- Sequential processing

Baseline idea

Graphs, visited states, and queues all live inside terms. The cost of updates is paid by rebuilding term structure.

$$\text{baseline} = \text{large terms} + \text{sequential processing}$$

Breadth-First Search Case Study: Experiment Setup

- Comparing two BFS encodings: parallel MemTRS vs sequential list-based baseline

Breadth-First Search Case Study: Experiment Setup

- Comparing two BFS encodings: parallel MemTRS vs sequential list-based baseline
- Randomly generated directed graphs

Breadth-First Search Case Study: Experiment Setup

- Comparing two BFS encodings: parallel MemTRS vs sequential list-based baseline
- Randomly generated directed graphs
- Number of nodes: $n = 1, 2, \ldots, 15$
- Number of edges: $1, 2, \ldots, n^2 - 1$

Breadth-First Search Case Study: Experiment Setup

- Comparing two BFS encodings: parallel MemTRS vs sequential list-based baseline
- Randomly generated directed graphs
- Number of nodes: $n = 1, 2, \ldots, 15$
- Number of edges: $1, 2, \ldots, n^2 - 1$
- Starting vertex: 0

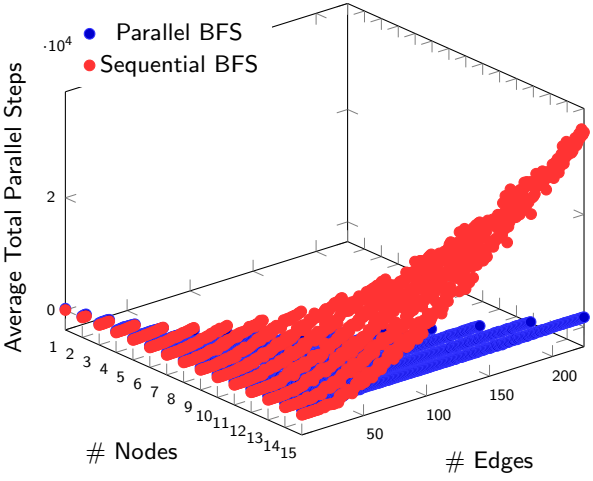
Breadth-First Search Case Study: Experiment Setup

- Comparing two BFS encodings: parallel MemTRS vs sequential list-based baseline
- Randomly generated directed graphs
- Number of nodes: $n = 1, 2, \ldots, 15$
- Number of edges: $1, 2, \ldots, n^2 - 1$
- Starting vertex: 0
- Repeated 3 times for each configuration

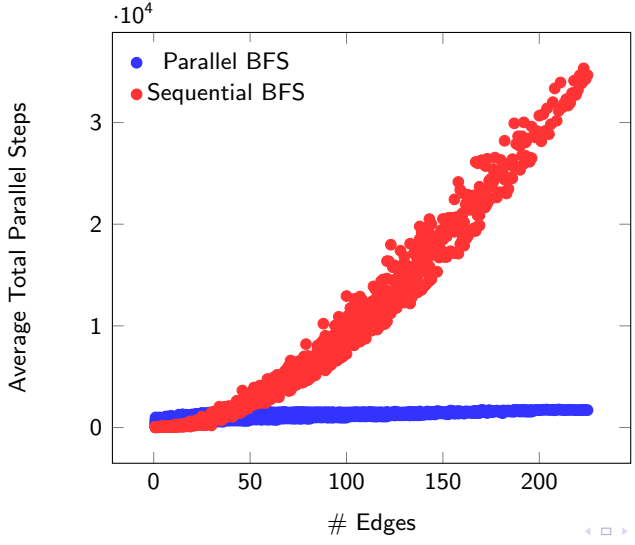
Breadth-First Search Case Study: Experiment Setup

- Comparing two BFS encodings: parallel MemTRS vs sequential list-based baseline
- Randomly generated directed graphs
- Number of nodes: $n = 1, 2, \ldots, 15$
- Number of edges: $1, 2, \ldots, n^2 - 1$
- Starting vertex: 0
- Repeated 3 times for each configuration
- Main metric: total number of parallel reduction steps

Breadth-First Search Case Study: Parallel Reduction Steps



Breadth-First Search Case Study: Parallel Reduction Steps



Breadth-First Search Case Study: Interpretation of Results

- Parallel MemTRS BFS requires far **fewer** parallel reduction steps

Breadth-First Search Case Study: Interpretation of Results

- Parallel MemTRS BFS requires far **fewer** parallel reduction steps
- For very sparse graphs, parallel BFS has **scheduling overhead** from **bfsWorker**s

Breadth-First Search Case Study: Interpretation of Results

- Parallel MemTRS BFS requires far **fewer** parallel reduction steps
- For very sparse graphs, parallel BFS has **scheduling overhead** from **bfsWorker**s
- Sequential baseline grows **faster** as graphs become denser

Breadth-First Search Case Study: Interpretation of Results

- Parallel MemTRS BFS requires far **fewer** parallel reduction steps
- For very sparse graphs, parallel BFS has **scheduling overhead** from **bfsWorker**s
- Sequential baseline grows **faster** as graphs become denser
- Main reason: visited-state lookup/update and **pushBack** are **linear** over linked lists

Breadth-First Search Case Study: Interpretation of Results

- Parallel MemTRS BFS requires far **fewer** parallel reduction steps
- For very sparse graphs, parallel BFS has **scheduling overhead** from **bfsWorker**s
- Sequential baseline grows **faster** as graphs become denser
- Main reason: visited-state lookup/update and **pushBack** are **linear** over linked lists
- Parallel BFS scales **better** with the number of edges

Reflection: What Is Still Difficult?

- MemTRS can model **realistic** shared-memory algorithms

Reflection: What Is Still Difficult?

- MemTRS can model **realistic** shared-memory algorithms
- But the user must still reason **manually** about races, evaluation order, and **CAS**-synchronization

Reflection: What Is Still Difficult?

- MemTRS can model **realistic** shared-memory algorithms
- But the user must still reason **manually** about races, evaluation order, and **CAS**-synchronization
- Termination analysis is **not** a direct extension of LCTRS

Reflection: What Is Still Difficult?

- MemTRS can model **realistic** shared-memory algorithms
- But the user must still reason **manually** about races, evaluation order, and **CAS**-synchronization
- Termination analysis is **not** a direct extension of LCTRS
- Scheduling, locks, and busy waiting **complicate** the meaning of termination

Reflection: What Is Still Difficult?

- MemTRS can model **realistic** shared-memory algorithms
- But the user must still reason **manually** about races, evaluation order, and **CAS**-synchronization
- Termination analysis is **not** a direct extension of LCTRS
- Scheduling, locks, and busy waiting **complicate** the meaning of termination
- Future work should focus on abstractions and analysis techniques

Research Project Summary

Main Research Question Revisited

Can we extend term rewriting with explicit shared memory so that it is **formal**, **implementable**, and **useful for modelling concurrent algorithms**?

✓ Introduced **Memory Term Rewriting Systems (MemTRS)**

Research Project Summary

Main Research Question Revisited

Can we extend term rewriting with explicit shared memory so that it is **formal**, **implementable**, and **useful for modelling concurrent algorithms**?

- ✓ Introduced **Memory Term Rewriting Systems (MemTRS)**
- ✓ Formalized memory operations, side effects, and parallel reduction

Research Project Summary

Main Research Question Revisited

Can we extend term rewriting with explicit shared memory so that it is **formal**, **implementable**, and **useful for modelling concurrent algorithms**?

- ✓ Introduced **Memory Term Rewriting Systems (MemTRS)**
- ✓ Formalized memory operations, side effects, and parallel reduction
- ✓ Analysed nondeterminism, data races, and CAS-based synchronization

Research Project Summary

Main Research Question Revisited

Can we extend term rewriting with explicit shared memory so that it is **formal**, **implementable**, and **useful for modelling concurrent algorithms**?

- ✓ Introduced **Memory Term Rewriting Systems (MemTRS)**
- ✓ Formalized memory operations, side effects, and parallel reduction
- ✓ Analysed nondeterminism, data races, and CAS-based synchronization
- ✓ Modelled arrays, matrices, concurrent queues, sorting, BFS, and Floyd–Warshall

Research Project Summary

Main Research Question Revisited

Can we extend term rewriting with explicit shared memory so that it is **formal**, **implementable**, and **useful for modelling concurrent algorithms**?

- ✓ Introduced **Memory Term Rewriting Systems (MemTRS)**
- ✓ Formalized memory operations, side effects, and parallel reduction
- ✓ Analysed nondeterminism, data races, and CAS-based synchronization
- ✓ Modelled arrays, matrices, concurrent queues, sorting, BFS, and Floyd–Warshall
- ✓ Implemented MemTRS in *Cora* and evaluated it through experiments

Thank you!

Do you have any questions?