Programming a multicore architecture without coherency and atomic operations

Jochem Rutgers, Marco Bekooij, Gerard Smit

2014-02-15

Parallel render example

One master thread:

```
1 data = read_3d_model_from_file();
2 go = 1;
3 while(done!=N) sleep();
4 display_frame(frame);
```

N slave threads:

```
1 while(!go) sleep();
2 render_my_part_of_frame(data,frame);
3 done++;
```

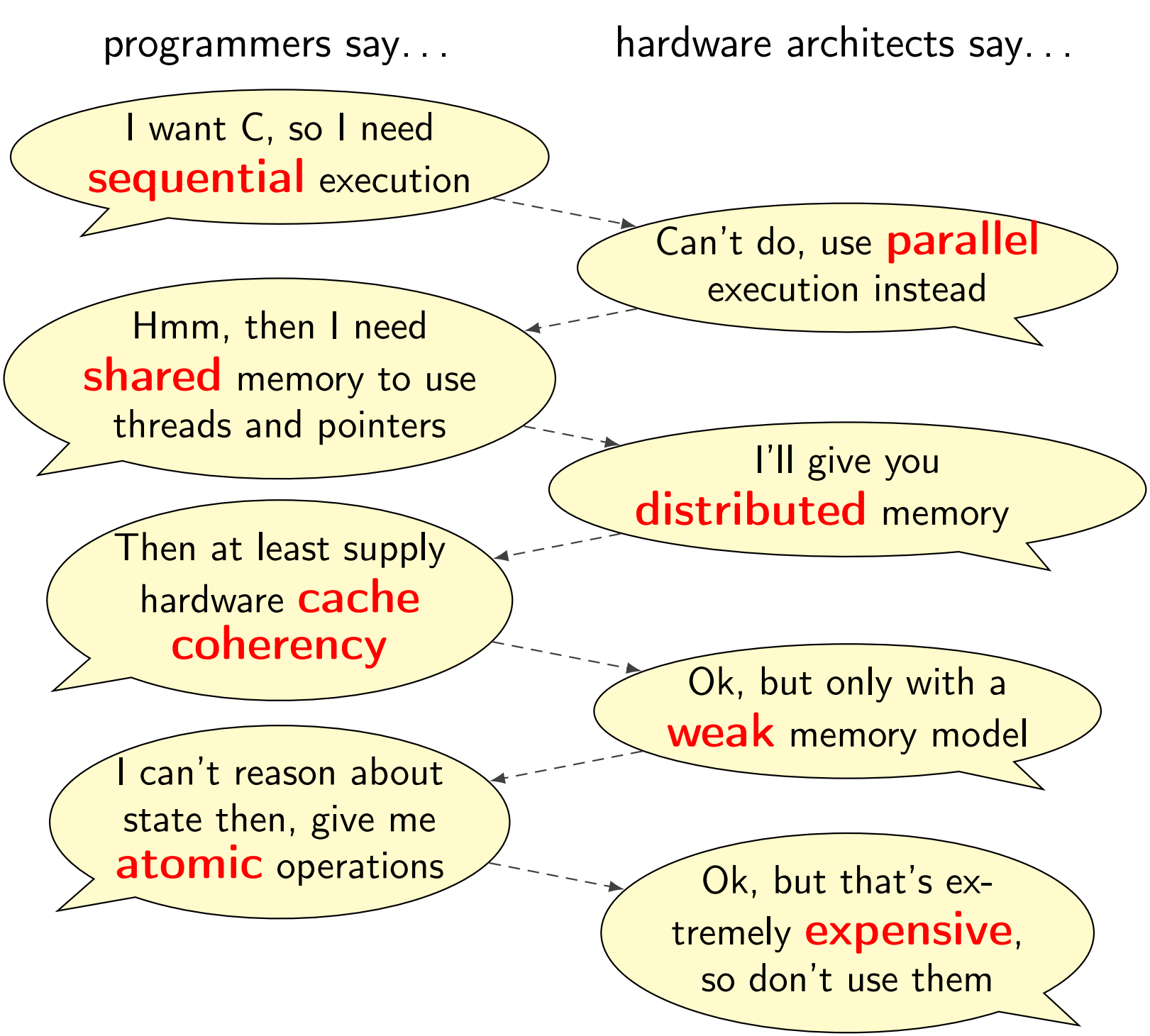
Parallel render Pthread example

One master thread:

```
1 data = read_3d_model_from_file();
2 pthread_barrier_wait();
3 pthread_barrier_wait();
4 display_frame(frame);
```

N slave threads:

```
1 pthread_barrier_wait();
2 render_my_part_of_frame(data,frame);
3 pthread_barrier_wait();
```


programmers say. . .
hardware architects say. . .
I want C, so I need sequential execution
Can't do, use parallel execution instead
Hmm, then I need shared memory to use threads and pointers
I'll give you distributed memory
Then at least supply hardware cache coherency
Ok, but only with a weak memory model
I can't reason about state then, give me atomic operations
Ok, but that's extremely expensive, so don't use them

Programming a multicore architecture

without

coherency and atomic operations

Programming a multicore architecture

without

coherency and atomic operations

. . . by starting from a functional language

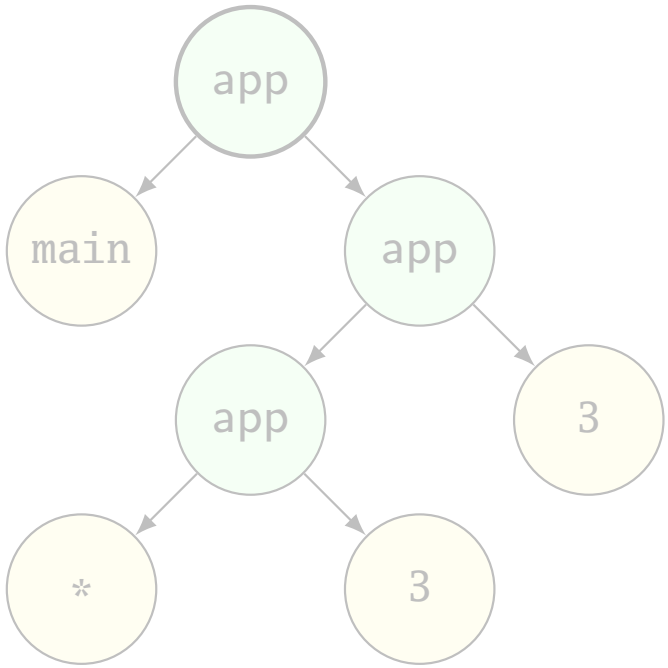
Dependency-only description

Program definition:

```
1  main h      = cylinder 2 h
2  cylinder r = * (* π (sqr r))
3  sqr x       = * x x
```

Evaluation sequence:

```
1  main                (* 3 3)
2  cylinder 2          (* 3 3)
3  * (* π (sqr 2))     (* 3 3)
4  * (* π (* 2 2))     (* 3 3)
5  * (* π (4))         (* 3 3)
6  * (12.57...)        (* 3 3)
7  * (12.57...)        9
8  113.10...
```



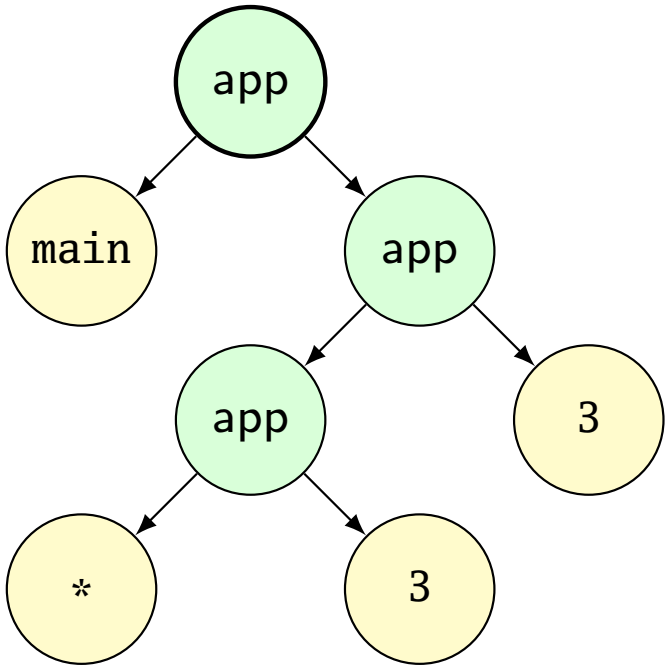
Dependency-only description

Program definition:

```
1  main h      = cylinder 2 h
2  cylinder r = * (* π (sqr r))
3  sqr x       = * x x
```

Evaluation sequence:

```
1  main                (* 3 3)
2  cylinder 2          (* 3 3)
3  * (* π (sqr 2))     (* 3 3)
4  * (* π (* 2 2))     (* 3 3)
5  * (* π (4))         (* 3 3)
6  * (12.57...)        (* 3 3)
7  * (12.57...)        9
8  113.10...
```



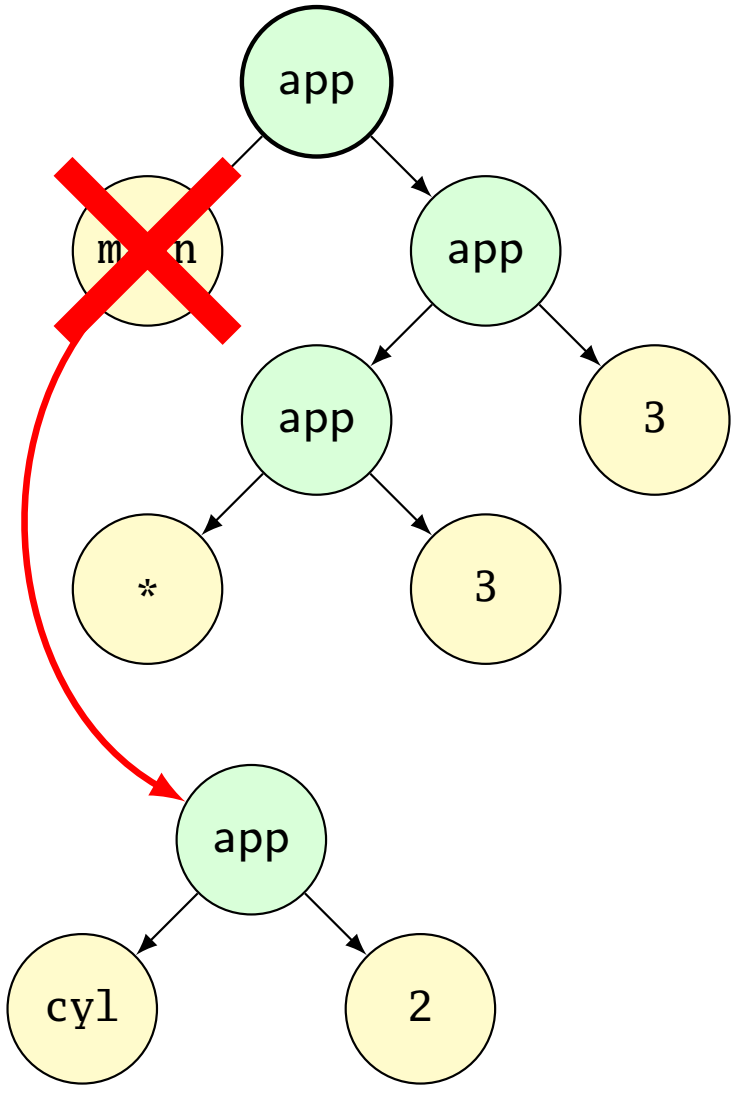
Dependency-only description

Program definition:

```
1  main h      = cylinder 2 h
2  cylinder r = * (* π (sqr r))
3  sqr x       = * x x
```

Evaluation sequence:

```
1  main                 (* 3 3)
2  cylinder 2           (* 3 3)
3  * (* π (sqr 2))      (* 3 3)
4  * (* π (* 2 2))      (* 3 3)
5  * (* π (4))          (* 3 3)
6  * (12.57...)         (* 3 3)
7  * (12.57...)         9
8  113.10...
```

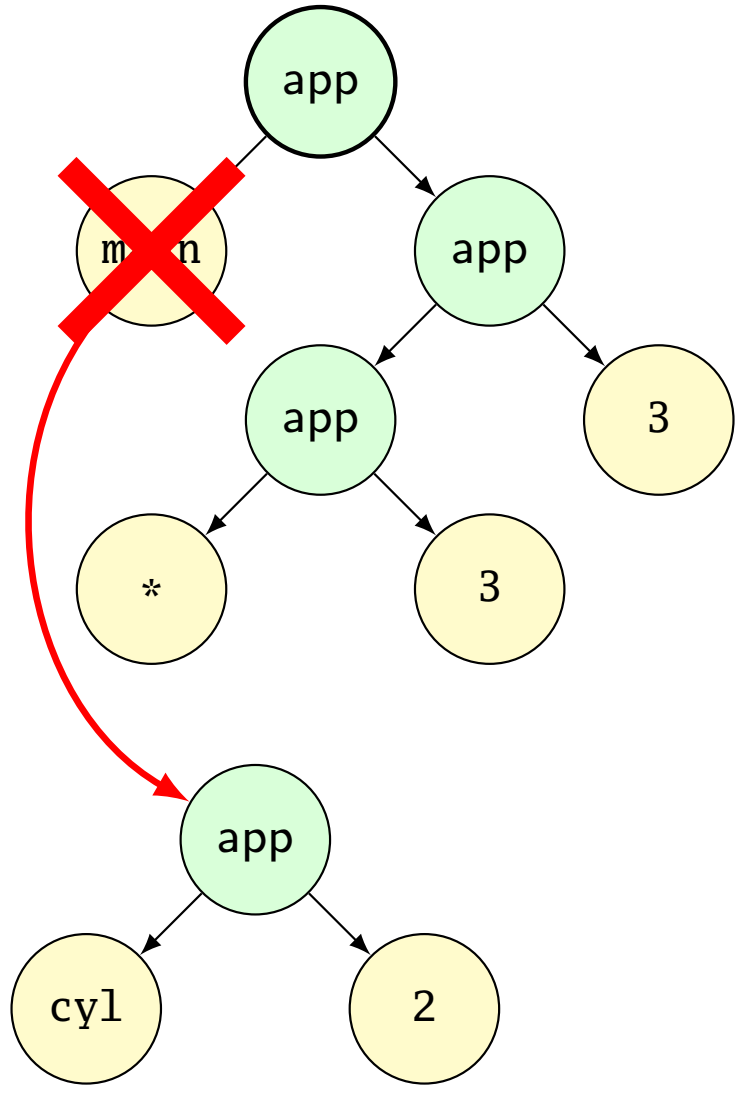


Dependency-only description

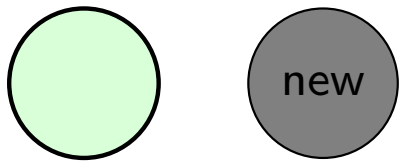
- Terms are constant
- Duplicates are identical
- No order in execution
- No memory/state
- No implicit behavior

. . . therefore. . .

- Parallel description
- Shortcuts in synchronization
- Lossy work distribution
- Only atomic pointer writes
- **atomic free**

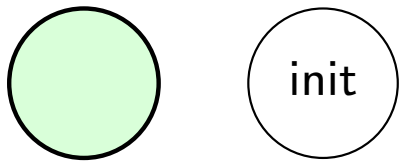


A λ-term's life



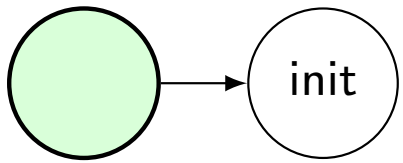
1. Memory allocation
2. Memory initialization (construction)
3. Add to expression
4. Replace with result (indirect)
5. Die
6. Garbage collect, free

A λ-term's life



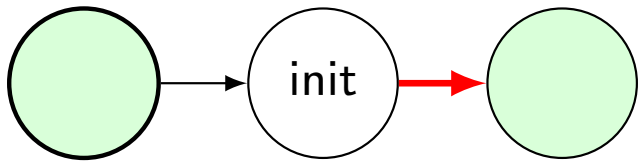
1. Memory allocation
2. Memory initialization (construction)
3. Add to expression
4. Replace with result (indirect)
5. Die
6. Garbage collect, free

A λ-term's life



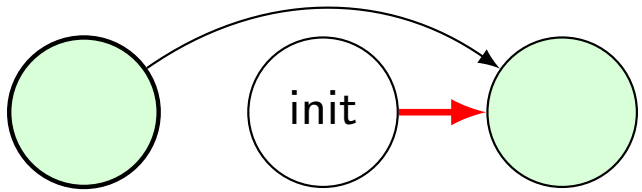
1. Memory allocation
2. Memory initialization (construction)
3. Add to expression
4. Replace with result (indirect)
5. Die
6. Garbage collect, free

A λ-term's life



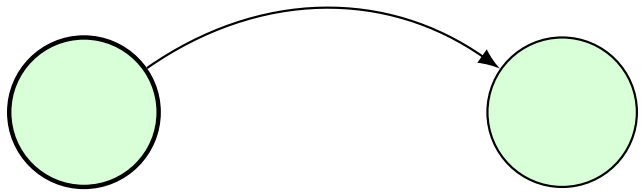
1. Memory allocation
2. Memory initialization (construction)
3. Add to expression
4. Replace with result (indirect)
5. Die
6. Garbage collect, free

A λ-term's life



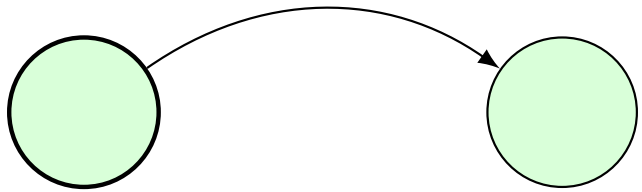
1. Memory allocation
2. Memory initialization (construction)
3. Add to expression
4. Replace with result (indirect)
5. Die
6. Garbage collect, free

A λ-term's life



1. Memory allocation
2. Memory initialization (construction)
3. Add to expression
4. Replace with result (indirect)
5. Die
6. Garbage collect, free

A λ-term's life



1. Memory allocation	private
2. Memory initialization (construction)	r/w access, private
3. Add to expression	read-only, shared
4. Replace with result (indirect)	pointer write, shared
5. Die	private
6. Garbage collect, free	private

From phases to rules

1. Memory allocation
2. Memory initialization (construction)

   Rule 1: construction must be completed; *flush / fence*
3. Add to expression

   Rule 2: pointer write is atomic, in total order; *(flush)*

   Rule 3: reads are in total order
4. Replace with result (indirect)

   (Rule 2 again)
5. Die

   Rule 4: all operations are completed; *flush / fence*
6. Garbage collect, free

From phases to rules to requirements

1. Memory allocation
2. Memory initialization (construction)

   Rule 1: construction must be completed; *flush* / *fence*
3. Add to expression

   Rule 2: pointer write is atomic, in total order; *(flush)*

   Rule 3: reads are in total order
4. Replace with result (indirect)

   (Rule 2 again)
5. Die

   Rule 4: all operations are completed; *flush / fence*
6. Garbage collect, free

From phases to rules to requirements

1. Memory allocation
2. Memory initialization (construction)

   Rule 1: construction must be completed; *flush* / *fence*
3. Add to expression

   Rule 2: pointer write is atomic, in total order; *(flush)*

   Rule 3: reads are in total order
4. Replace with result (indirect)

   (Rule 2 again)
5. Die

   Rule 4: all operations are completed; *flush / fence*
6. Garbage collect, free

From phases to rules to requirements

1. Memory allocation
2. Memory initialization (construction)

   Rule 1: construction must be completed; *flush* / *fence*
3. Add to expression

   Rule 2: pointer write is atomic, in total order; *(flush)*

   Rule 3: reads are in total order
4. Replace with result (indirect)

   (Rule 2 again)
5. Die

   Rule 4: all operations are completed; *flush / fence*
6. Garbage collect, free

From phases to rules to requirements

1. Memory allocation
2. Memory initialization (construction)

   Rule 1: construction must be completed; *flush* / *fence*
3. Add to expression

   Rule 2: pointer write is atomic, in total order; *(flush)*

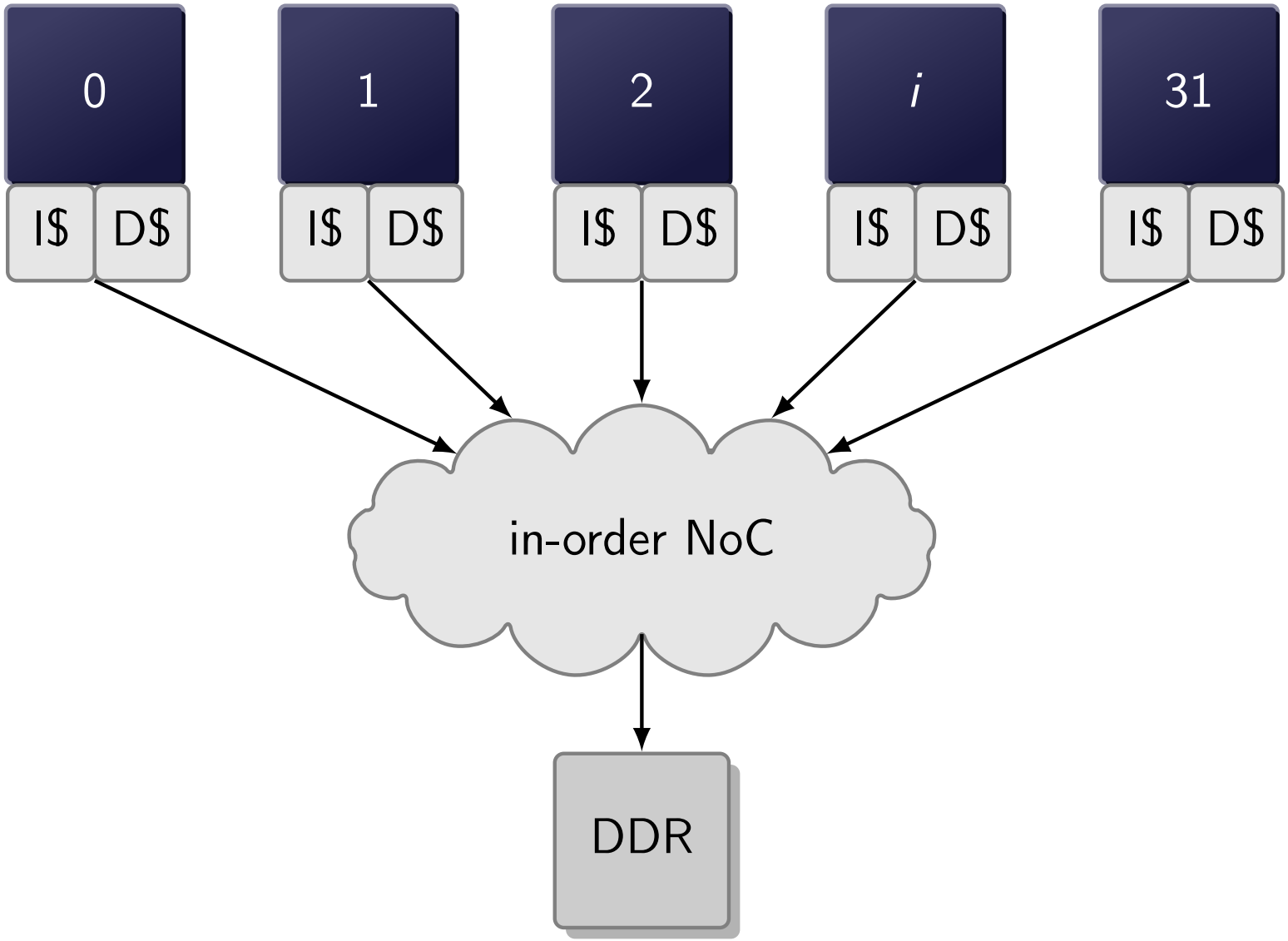
   Rule 3: reads are in total order
4. Replace with result (indirect)

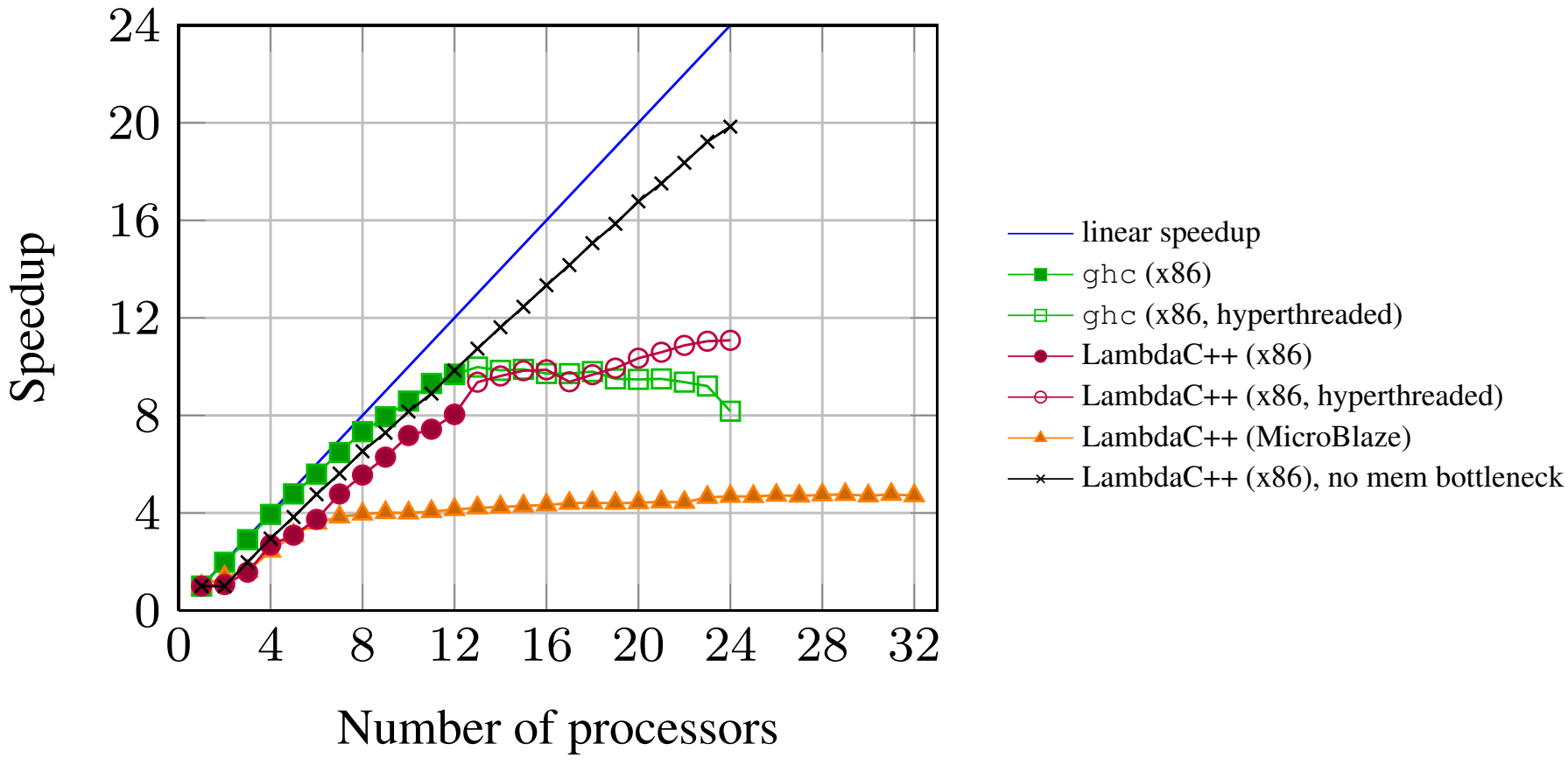
   (Rule 2 again)
5. Die

   Rule 4: all operations are completed; *flush / fence*
6. Garbage collect, free

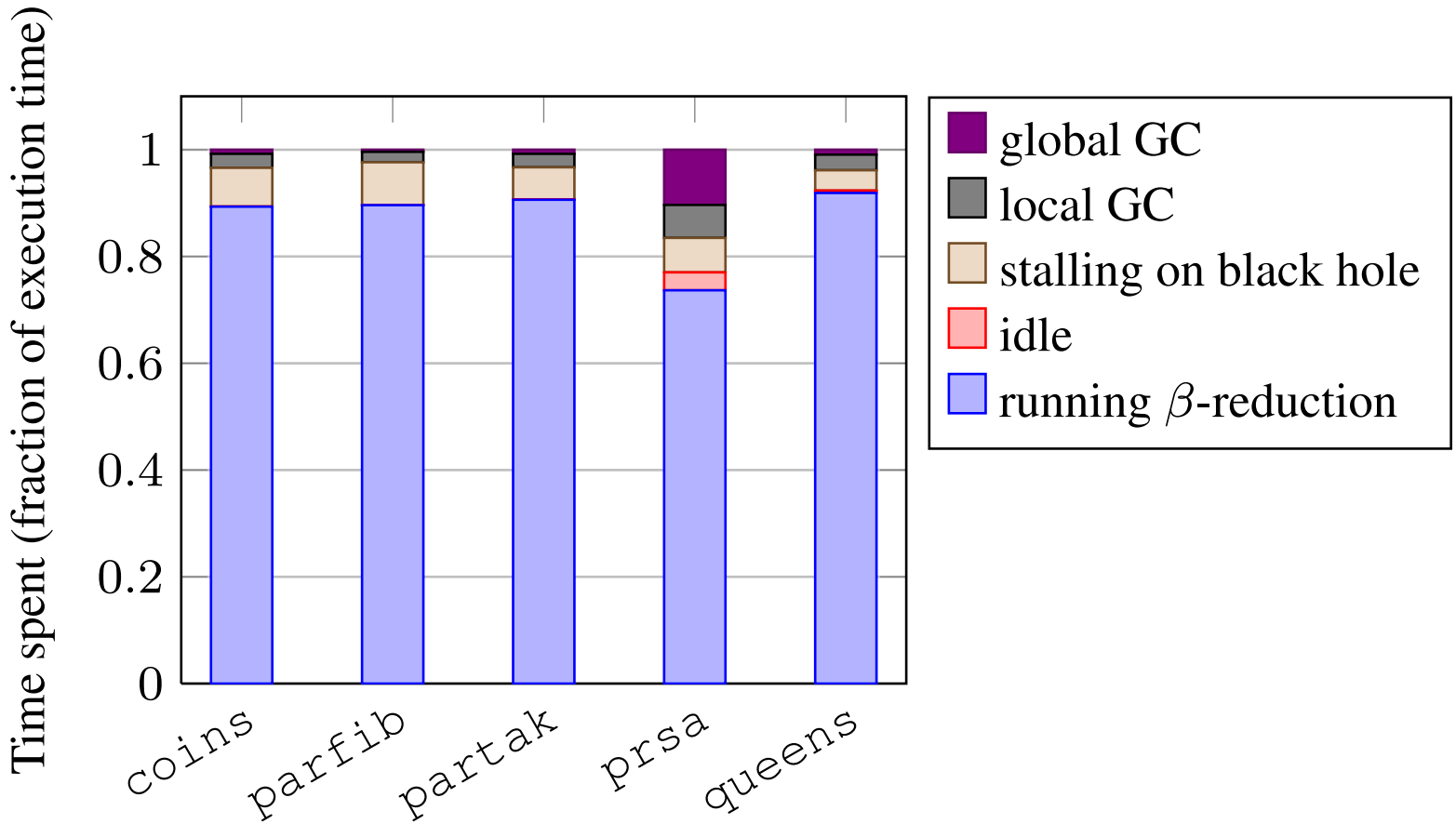
λ-calculus in C++

- λ-terms implemented as C++ templates/**classes**
- **gcc**; ()-operator overloading gives FP-like syntax
- data type: (complex) doubles, large integers (GNU MP)
- one **worker** thread per core
- Haskell-like `par` and `pseq`
- local vs. global data and garbage collection
- mark–sweep GC (global GC is stop-the-world)
- $\approx$400 instructions in run-time per created λ-term
- $\approx$5500 LoC
- GPLv3
- `https://sites.google.com/site/jochemrutgers/lambdacpp`

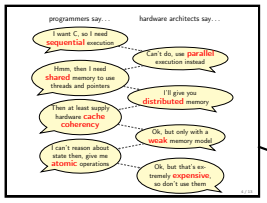

0
1
2
i
31
I$
D$
I$
D$
I$
D$
I$
D$
I$
D$
in-order NoC
DDR

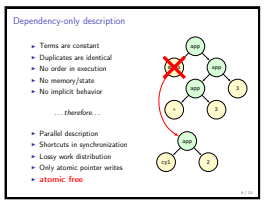

Speedup
24
20
16
12
8
4
0
0
4
8
12
16
20
24
28
32
Number of processors
linear speedup
ghc (x86)
ghc (x86, hyperthreaded)
LambdaC++ (x86)
LambdaC++ (x86, hyperthreaded)
LambdaC++ (MicroBlaze)
LambdaC++ (x86), no mem bottleneck

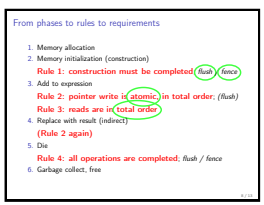
(b) `parfib`

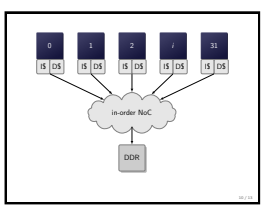

Time spent (fraction of execution time)
1
0.8
0.6
0.4
0.2
0
coins
parfib
partak
prsa
queens
global GC
local GC
stalling on black hole
idle
running β-reduction

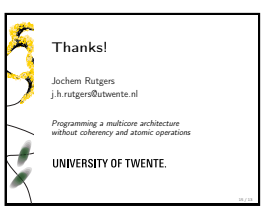
Highlights

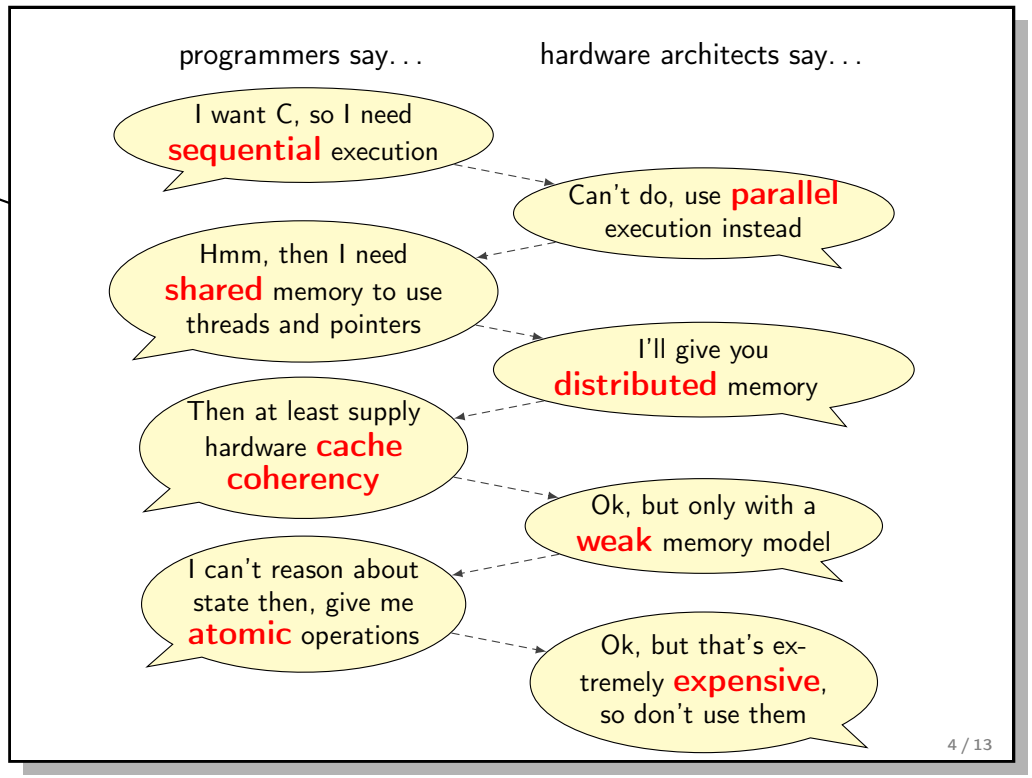






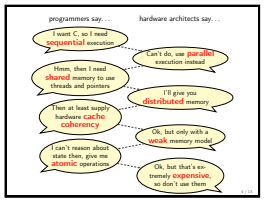


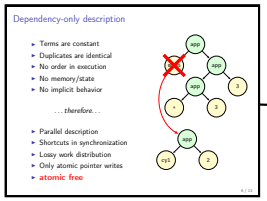


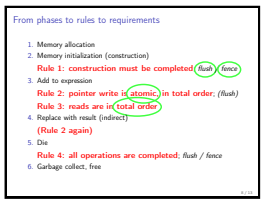


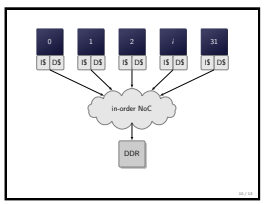
- Accept the **hardware** trends
- Another programming **model** might be more suitable
- Extreme example: **FP** is hardware-friendly...
- ...cache coherency and **atomics** are avoided

Highlights







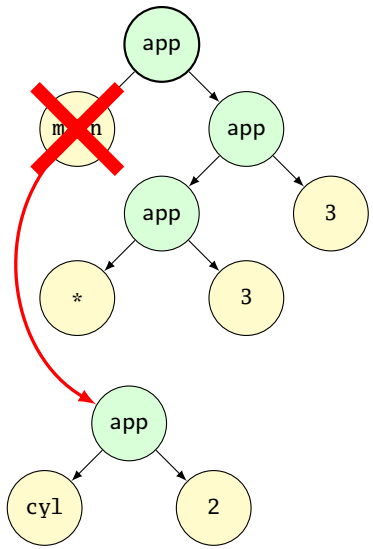


Dependency-only description

- Terms are constant
- Duplicates are identical
- No order in execution
- No memory/state
- No implicit behavior

. . . therefore. . .

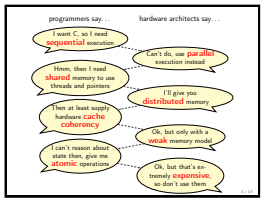
- Parallel description
- Shortcuts in synchronization
- Lossy work distribution
- Only atomic pointer writes
- atomic free

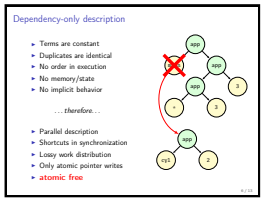


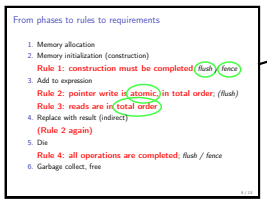
6 / 13

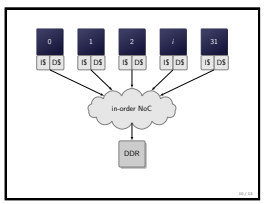
- Accept the **hardware** trends
- Another programming **model** might be more suitable
- Extreme example: **FP** is hardware-friendly. . .
- . . . cache coherency and **atomics** are avoided

Highlights









From phases to rules to requirements

1. Memory allocation
2. Memory initialization (construction)

   Rule 1: construction must be completed; *flush* / *fence*
3. Add to expression

   Rule 2: pointer write is atomic, in total order; *(flush)*

   Rule 3: reads are in total order
4. Replace with result (indirect)

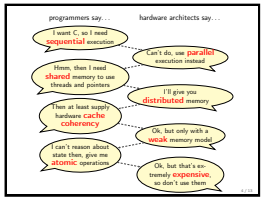
   (Rule 2 again)
5. Die

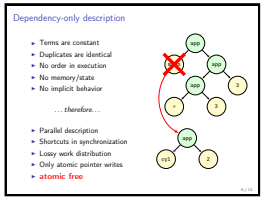
   Rule 4: all operations are completed; *flush / fence*
6. Garbage collect, free

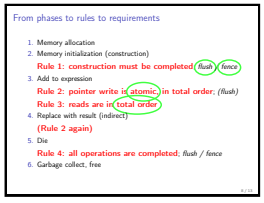
8 / 13

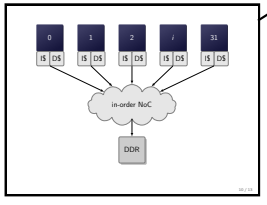
- Accept the **hardware** trends
- Another programming **model** might be more suitable
- Extreme example: **FP** is hardware-friendly...
- ...cache coherency and **atomics** are avoided

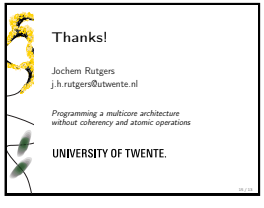
Highlights

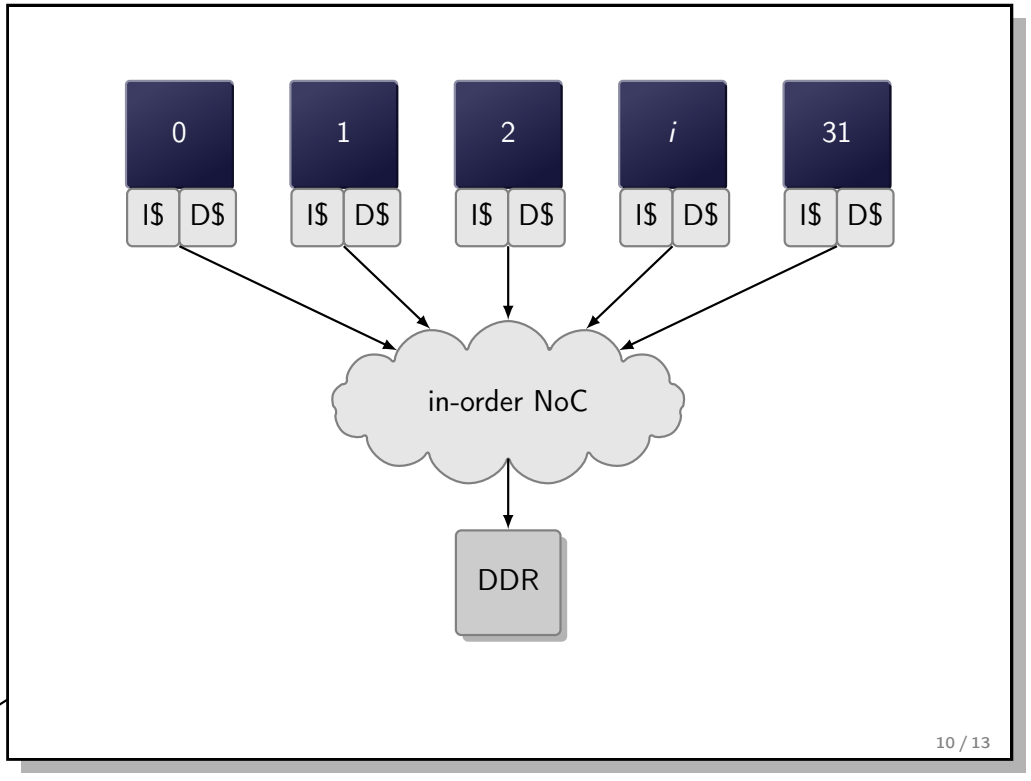






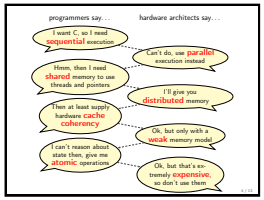


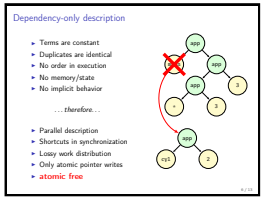


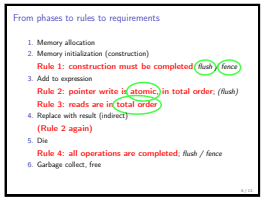


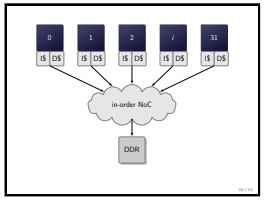
- Accept the **hardware** trends
- Another programming **model** might be more suitable
- Extreme example: **FP** is hardware-friendly...
- ...cache coherency and **atomics** are avoided

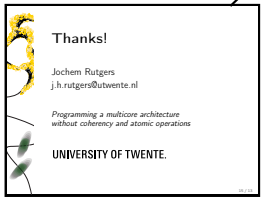
Highlights

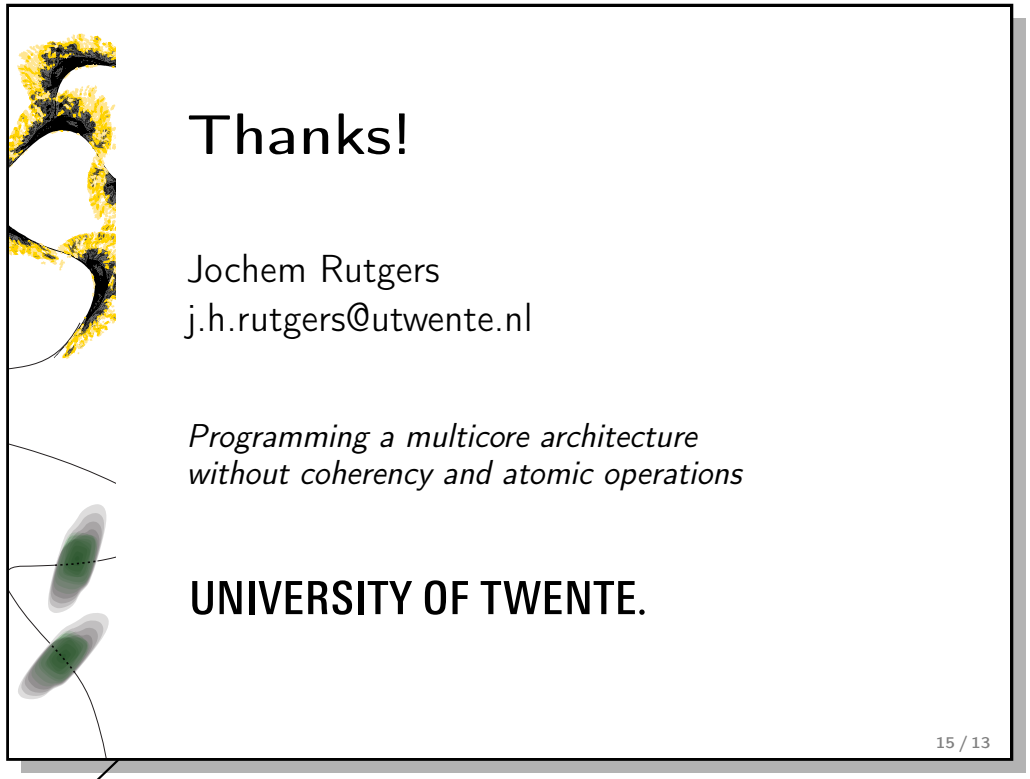








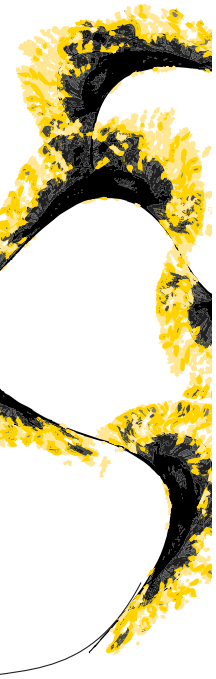




- Accept the **hardware** trends
- Another programming **model** might be more suitable
- Extreme example: **FP** is hardware-friendly...
- ...cache coherency and **atomics** are avoided

Part II

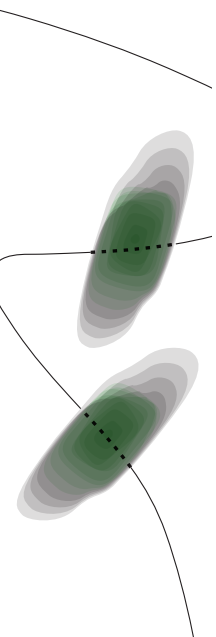
Appendix



Thanks!

Jochem Rutgers
j.h.rutgers@utwente.nl

Programming a multicore architecture without coherency and atomic operations



UNIVERSITY OF TWENTE.

benchmark	local applications[a]	local constants[a]	globals[a]
`coins`	0.418	0.582	$1.36 \cdot 10^{-4}$
`parfib`	0.379	0.621	$1.44 \cdot 10^{-4}$
`partak`	0.351	0.648	$5.47 \cdot 10^{-4}$
`prsa`	0.412	0.583	$4.97 \cdot 10^{-3}$
`queens`	0.445	0.555	$9.10 \cdot 10^{-5}$

[a] Fraction of sum of all global and local terms

Table 2. Generated terms during evaluation (LambdaC++, x86, 12 cores)