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. \texttt{data} = \texttt{read\_3d\_model\_from\_file()};
2. \texttt{pthread\_barrier\_wait()};
3. \texttt{pthread\_barrier\_wait()};
4. \texttt{display\_frame(frame)};

N slave threads:

1. \texttt{pthread\_barrier\_wait()};
2. \texttt{render\_my\_part\_of\_frame(data,frame)};
3. \texttt{pthread\_barrier\_wait()};
Programmers say...

I want C, so I need **sequential** execution.

Hmm, then I need **shared** memory to use threads and pointers.

Then at least supply hardware **cache coherency**.

I can’t reason about state then, give me **atomic** operations.

Hardware architects say...

Can’t do, use **parallel** execution instead.

I’ll give you **distributed** memory.

Ok, but only with a **weak** memory model.

Ok, but that’s extremely **expensive**, so don’t use them.
Programming a multicore architecture without coherency and atomic operations
Programming a multicore architecture \textbf{without} coherency and atomic operations

\ldots by starting from a functional language
Program definition:

1. \texttt{main h = cylinder 2 h}
2. \texttt{cylinder r = \ast (\ast \pi (\text{sqr } r))}
3. \texttt{sqr x = \ast x x}

Evaluation sequence:

1. \texttt{main} \quad \texttt{(* 3 3)}
2. \texttt{cylinder 2} \quad \texttt{(* 3 3)}
3. \texttt{(* \pi (\text{sqr} 2))} \quad \texttt{(* 3 3)}
4. \texttt{(* \pi (* 2 2))} \quad \texttt{(* 3 3)}
5. \texttt{(* \pi (4))} \quad \texttt{(* 3 3)}
6. \texttt{* (12.57\ldots)} \quad \texttt{(* 3 3)}
7. \texttt{* (12.57\ldots)} \quad \texttt{9}
8. \texttt{113.10\ldots}
Dependency-only description

Program definition:

1. \texttt{main h = cylinder 2 h}
2. \texttt{cylinder r = (* \pi (sqr r))}
3. \texttt{sqr x = \* x x}

Evaluation sequence:

1. \texttt{main (* 3 3)}
2. \texttt{cylinder 2 (* 3 3)}
3. \texttt{* (* \pi (sqr 2)) (* 3 3)}
4. \texttt{* (* \pi (* 2 2)) (* 3 3)}
5. \texttt{* (* \pi (4)) (* 3 3)}
6. \texttt{* (12.57\ldots) (* 3 3)}
7. \texttt{* (12.57\ldots) 9}
8. \texttt{113.10\ldots}
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 \( \lambda \)-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 $\lambda$-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 \( \lambda \)-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 $\lambda$-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**; $\text{flush} / \text{fence}$
3. Add to expression
   **Rule 2: pointer write is atomic, in total order**; $(\text{flush})$
   **Rule 3: reads are in total order**
4. Replace with result (indirect)
   (Rule 2 again)
5. Die
   **Rule 4: all operations are completed**; $\text{flush} / \text{fence}$
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; \(\text{flush} / \text{fence}\)

3. Add to expression
   
   **Rule 2:** pointer write is atomic, in total order; (\(\text{flush}\))
   
   **Rule 3:** reads are in total order

4. Replace with result (indirect)
   
   (Rule 2 again)

5. Die
   
   **Rule 4:** all operations are completed; \(\text{flush} / \text{fence}\)

6. Garbage collect, free
\(\lambda\)-calculus in C++

- \(\lambda\)-terms implemented as C++ templates/classes
- \texttt{gcc} (\()\)-operator overloading gives FP-like syntax
- data type: (complex) doubles, large integers (GNU MP)
- one \textbf{worker} thread per core
- Haskell-like \texttt{par} and \texttt{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 \(\lambda\)-term
- \(\approx 5500\) LoC
- GPLv3
- \url{https://sites.google.com/site/jochemrutgers/lambdacpp}
in-order NoC

DDR
In this paper, we showed that these hardware issues can be overcome at a different level. To this extend, although untested, a solution could involve having a (large) scratch.

Table 2. The time is the sum of the time spent in such a phase, presented.

\[ \beta \] is measured. Figure 4 shows the most important states a worker is left as future work.

Finally, the distribution of where time is spent during execution can be done transparently to the application. However, testing such a setup is left as future work.

They are dead at the successive GC. Such a modification to the RTS orders of magnitude higher than that of the global terms.

99.8% of the data does not survive that private nursery stage, so the effects of the memory bottleneck will be reduced significantly.

The table shows that the number of local terms is significantly higher than that of the global terms. If all local terms can be kept local, traffic to main memory and parallel reductions.

orders of magnitude higher than that of the global terms.

Time spent during execution (LambdaC++, x86, 12 cores) is used. The table shows that the number of local terms is

\[ 10 \cdot 10^{-97} \]

\[ 10 \cdot 47 \cdot 44 \cdot 36 \]

\[ 10 \cdot 10^{-3} \cdot 3 \cdot 4 \cdot 4 \cdot 4 \cdot 4 \]

\[ 10 \cdot 10^{-5} \cdot 3 \cdot 4 \cdot 4 \cdot 4 \cdot 4 \]

\[ 10 \cdot 10^{-5} \cdot 3 \cdot 4 \cdot 4 \cdot 4 \cdot 4 \]

\[ 10 \cdot 10^{-5} \cdot 3 \cdot 4 \cdot 4 \cdot 4 \cdot 4 \]

\[ 10 \cdot 10^{-5} \cdot 3 \cdot 4 \cdot 4 \cdot 4 \cdot 4 \]

\[ 10 \cdot 10^{-5} \cdot 3 \cdot 4 \cdot 4 \cdot 4 \cdot 4 \]

\[ 10 \cdot 10^{-5} \cdot 3 \cdot 4 \cdot 4 \cdot 4 \cdot 4 \]

\[ 10 \cdot 10^{-5} \cdot 3 \cdot 4 \cdot 4 \cdot 4 \cdot 4 \]

\[ 10 \cdot 10^{-5} \cdot 3 \cdot 4 \cdot 4 \cdot 4 \cdot 4 \]

\[ 10 \cdot 10^{-5} \cdot 3 \cdot 4 \cdot 4 \cdot 4 \cdot 4 \]

\[ 10 \cdot 10^{-5} \cdot 3 \cdot 4 \cdot 4 \cdot 4 \cdot 4 \]

\[ 10 \cdot 10^{-5} \cdot 3 \cdot 4 \cdot 4 \cdot 4 \cdot 4 \]

\[ 10 \cdot 10^{-5} \cdot 3 \cdot 4 \cdot 4 \cdot 4 \cdot 4 \]

\[ 10 \cdot 10^{-5} \cdot 3 \cdot 4 \cdot 4 \cdot 4 \cdot 4 \]

\[ 10 \cdot 10^{-5} \cdot 3 \cdot 4 \cdot 4 \cdot 4 \cdot 4 \]

\[ 10 \cdot 10^{-5} \cdot 3 \cdot 4 \cdot 4 \cdot 4 \cdot 4 \]

\[ 10 \cdot 10^{-5} \cdot 3 \cdot 4 \cdot 4 \cdot 4 \cdot 4 \]
In this paper, we showed that these hardware issues can be overcome at a different level. To this extent,

![Bar chart showing time spent during execution (LambdaC++, x86, 12 cores)](chart)

- **coins**: 
  - Global GC: 0.005
  - Local GC: 0.002
  - Stalling on black hole: 0.001
  - Idle: 0.003
  - Running β-reduction: 0.991

- **parfib**: 
  - Global GC: 0.005
  - Local GC: 0.002
  - Stalling on black hole: 0.001
  - Idle: 0.003
  - Running β-reduction: 0.991

- **partak**: 
  - Global GC: 0.005
  - Local GC: 0.002
  - Stalling on black hole: 0.001
  - Idle: 0.003
  - Running β-reduction: 0.991

- **prsa**: 
  - Global GC: 0.005
  - Local GC: 0.002
  - Stalling on black hole: 0.001
  - Idle: 0.003
  - Running β-reduction: 0.991

- **queens**: 
  - Global GC: 0.005
  - Local GC: 0.002
  - Stalling on black hole: 0.001
  - Idle: 0.003
  - Running β-reduction: 0.991

6. Conclusion

One of the hardware design issues of a multiprocessor platform is atomic global communication between cores, such as cache coherency and synchronization. In this paper, we showed that these hardware issues can be overcome at a different level. To this extend,
Accept the **hardware** trends

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

Another programming **model** might be more suitable

Extreme example: **FP** is hardware-friendly...

... cache coherency and **atomics** are avoided
From phases to rules to requirements

1. Memory allocation
2. Memory initialization (construction)
   - **Rule 1**: construction must be completed \(\text{flush} / \text{fence}\)
3. Add to expression
   - **Rule 2**: pointer write is atomic, in total order; \(\text{flush}\)
   - **Rule 3**: reads are in total order
   - (Rule 2 again)
4. Replace with result (indirect)
5. Die
   - **Rule 4**: all operations are completed; \(\text{flush} / \text{fence}\)
6. Garbage collect, free

- Accept the **hardware** trends
- Another programming **model** might be more suitable
- Extreme example: **FP** is hardware-friendly...
- \ldots cache coherency and **atomics** are avoided
- Accept the **hardware** trends
- Another programming **model** might be more suitable
- Extreme example: **FP** is hardware-friendly...
- \ldots cache coherency and **atomics** are avoided
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.
Table 2. Generated terms during evaluation (LambdaC++, x86, 12 cores)