SCIENCE
SCHOOL OF COMPUTER SCIENCE

30 April 2021

Desperately Seeking CML

A search for a highly-efficient implementation of Concurrent ML

James Cooper

Outline

SCIENCE
SCHOOL OF COMPUTER SCIENCE

Outline

Motivation

- Looking at Belief Propagation in Stereo Matching
- Belief Propagation is described in terms of message passing
- Implementations are usually nothing like that, though
- Seemed like a perfect fit with Concurrent ML
- No evidence combination of the two ever investigated
- (NB: BP no longer state-of-the-art in Stereo Matching)

Challenge

- Stereo Matching involves images
- Output is a new image – many pixels
- With BP, each pixel can be a messenger
- Umpteen messages to exchange
- Therefore, need *extremely* efficient CML
- Thus, assess the options out there

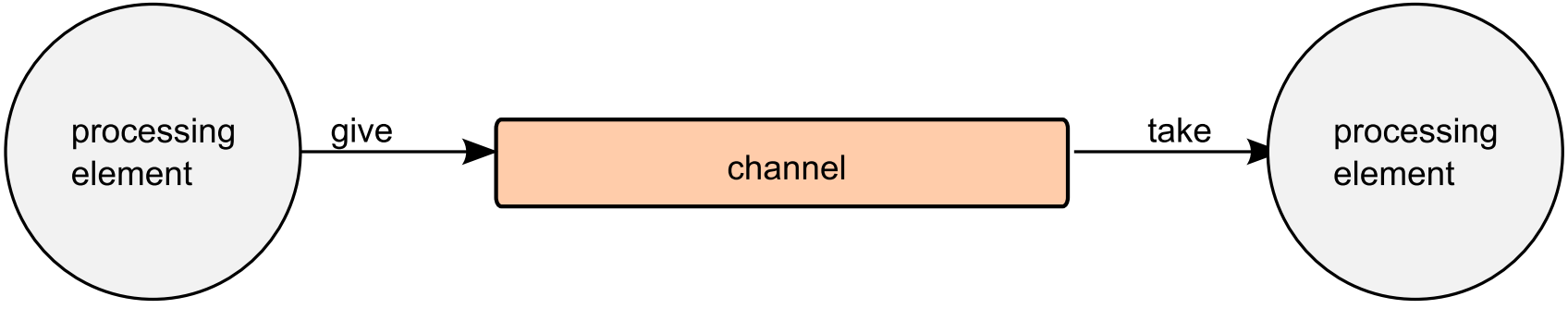
Outline

Concurrent 'Meta Language'

- Concurrency via lightweight communicating threads
- Shared-memory only
- First implemented in Standard ML of New Jersey
- Initially for concurrent but not parallel programming
- Separate processing elements communicate via channels
- Sadly, mostly forgotten now[1]

[1] But see "Concurrent ML - The One That Got Away" talk by Michael Sperber at Code Mesh 2017 (on YouTube)

Concurrent ML Diagramatically



Outline

Selected CML/host implementation requirements

- Parallel CML – aiming for 'real-time' stereo matching
- Ahead-of-time compilation/fast standalone executables
- Available for common Linux distributions
- Pre-existing support for image file IO
- Well documented
- Still maintained

Possibilities

The ready-to-use options which meet all criteria:

Possibilities

The ready-to-use options which meet all criteria:

Photo by Andrea Piacquadio from Pexels

https://www.pexels.com/photo/woman-in-white-shirt-showing-frustration-3807738/

Possibilities

- *No* language with CML support that meets all criteria

Possibilities

- *No* language with CML support that meets all criteria
- SML/NJ seems to be single-core only
- A number of languages come close, including: Go; Clojure with Core.Async library; Rust with Crossbeam crate (maybe others); Crystal; Julia; Kotlin; C++ with various old libraries; .NET with Tasks; could go on...
- All of these seem to stop at a less advanced approach
- Not necessarily parallel, either
- Many also fail to meet at least one other criterion

Possibilities with CML support I

If one relaxes criteria:

- F# with Hopac library (`https://github.com/Hopac/Hopac`)
- Haskell with Control.Concurrent.CML [2] or Transactional Events [4]
- Guile Scheme (`https://www.gnu.org/software/guile/`) with Fibers library (`https://github.com/wingo/fibers`)
- Manticore [7]
- MLton [12]

Possibilities with CML support II

- OCaml with Events module
  (`https://caml.inria.fr/pub/docs/manual-ocaml/libref/Event.html`)
  (see also [5])
- Racket [6]

Hopac

- Tried out Hopac earlier
- Compared CML with simple imperative approach for median filter [3]
- Results were underwhelming, at best
- 20+ times slower than naïve imperative version
- Turns out, *maybe* a memory leak (issues #192 & #201 at `https://github.com/Hopac/Hopac`)
- Not really maintained anymore, anyway

Elimination

- Hopac has issues
- Haskell's libraries are old and unmaintained; research prototypes only
- Guile doesn't do standalone executables
- Manticore seems to be similar to MLton, but has drawbacks
- MLton – selected
- OCaml – single-core-only
- Racket – selected

MLton

- Somewhat-maintained, largely-stable SML variant
- Includes a near-complete port of CML
- Haphazardly documented
- Some rough edges
- Few extra libraries

Racket

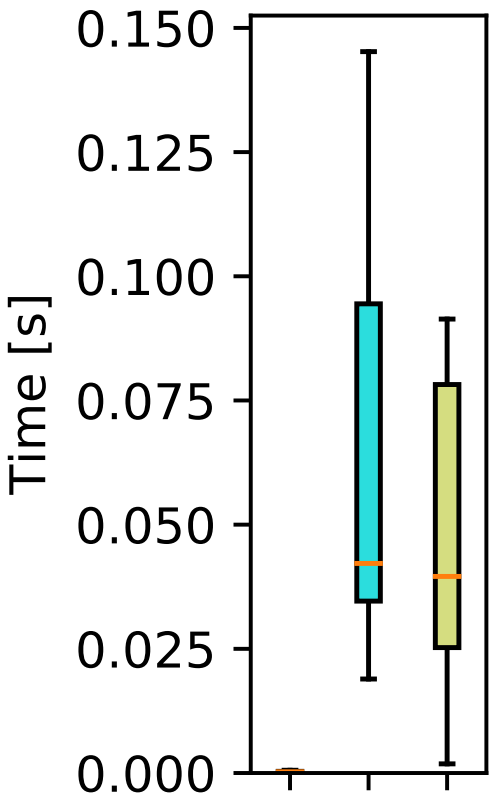
- Scheme LISP implementation – untyped & typed
- *Mostly* re-implements CML in its `sync` library
- Generally well-documented
- Active and friendly community
- Wide array of libraries available
- Doesn't *really* do standalone executables

Outline

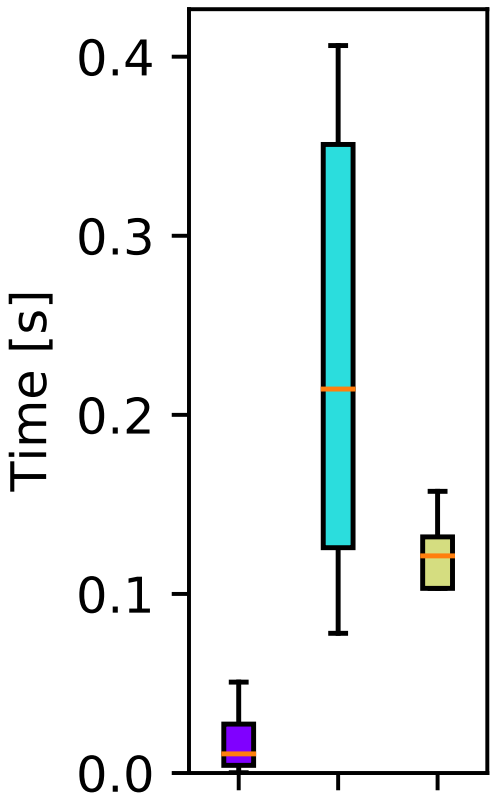
Comparative Benchmarks

- Assess running times, determine ‘fastest’ ‘language’
- Tested MLton and both Typed & Untyped Racket
- Implemented six exemplar test programs (See `https://github.com/jcoo092/CML_benchmarks`):
  - Linear Algebra
  - Monte Carlo Pi
  - Communications Time
  - Selection Time
  - Spawn
  - Whispers

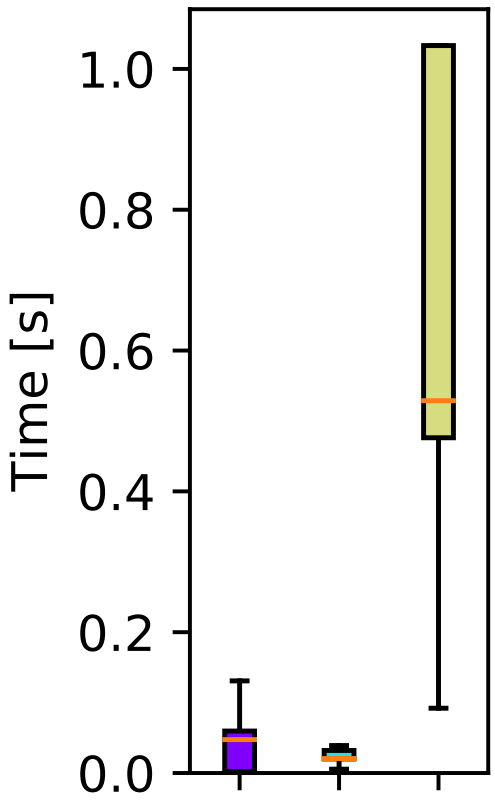
Results: Box & Whisker Plots of Running Times 1



Commstime

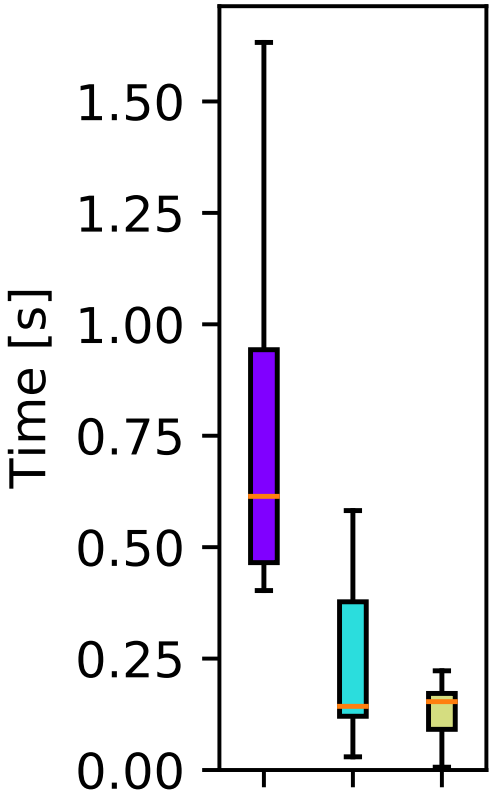


Linear Algebra (Matrix)

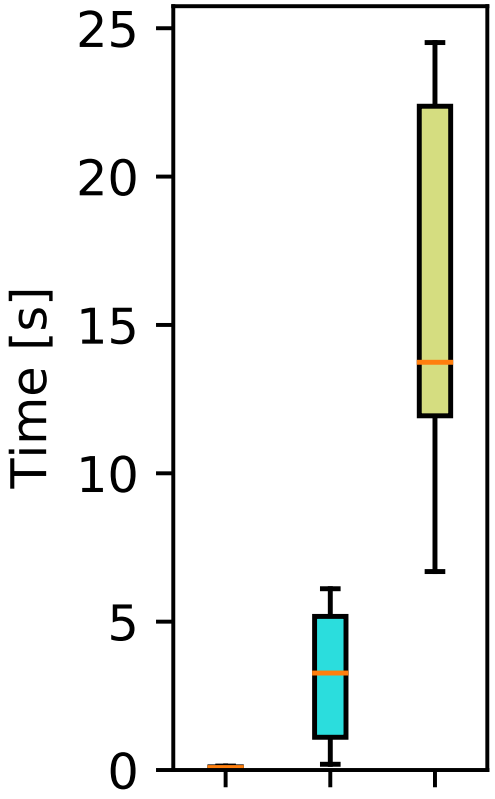


Monte Carlo Pi

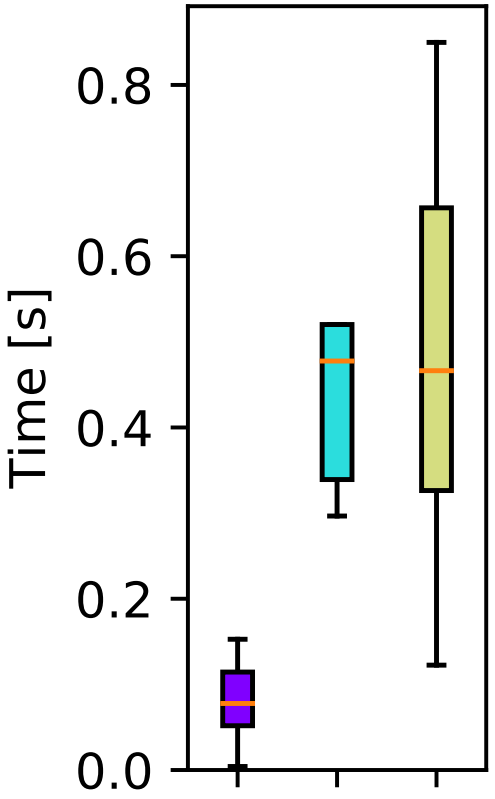
Results: Box & Whisker Plots of Running Times 2



Selection Time



Spawn



Whispers (Ring)

There's a Catch...

- MLton outperforms Racket (both typed and untyped) on all but two benchmarks: Selection Time and Monte Carlo Pi

There's a Catch...

- MLton outperforms Racket (both typed and untyped) on all but two benchmarks: Selection Time and Monte Carlo Pi
- MCP tests parallelism capabilities of the language
- MCP tests showed the MLton program is very fast, but gets slower the more threads are used in the program – even with 2

There's a Catch...

- MLton outperforms Racket (both typed and untyped) on all but two benchmarks: Selection Time and Monte Carlo Pi
- MCP tests parallelism capabilities of the language
- MCP tests showed the MLton program is very fast, but gets slower the more threads are used in the program – even with 2
- Turns out MLton is single-core-only.
- MultiMLton [10] – now dead and buried

Outline

Where to Now?

- MLton is a fairly good implementation of SML
- Lack of multithreading made it out-of-scope for this work
- Manticore is (roughly) another SML implementation
- Manticore comes with parallel CML built-in
- Obvious next choice
- *Probably* not even too hard to port MLton to Manticore

Manticore

- Research language created in late 2000s
- Standard ML-esque
- Explicitly for testing parallelism designs
- Creators implemented just enough for research goals

Sting in Manticore's Tail

- Porting from MLton proved more difficult than anticipated
- No ready-to-run installer, build from source instead
- Needed to adapt a Dockerfile from one of the researchers
- Missing parts of SML Basis (aka standard) library
- Almost no documentation – relied on reading source, plus some exemplar benchmark programs and trial & error

Sting in Manticore's Tail

- Porting from MLton proved more difficult than anticipated
- No ready-to-run installer, build from source instead
- Needed to adapt a Dockerfile from one of the researchers
- Missing parts of SML Basis (aka standard) library
- Almost no documentation – relied on reading source, plus some exemplar benchmark programs and trial & error
- Turned out to be lacking core parts of CML!

Outline

Conclusion I

- Belief Propagation based on message-passing concepts
- Concurrent ML seems like a clear theoretical fit to BP
- Investigated CML options
- Turns out there are few usable ones out there now
- No ‘ideal’ candidate
- Had tried F# + Hopac earlier – poor performance
- Tested out MLton and Racket

Conclusion II

- MLton much faster than Racket, but actually not parallel
- Changed focus to Manticore
- Manticore now missing CML bits...
- Too many problems, redirected research focus
- Hope to re-visit in future

Acknowledgements

Thanks must firstly be given to my supervisors, Dr. Radu Nicolescu and Associate-Professor Patrice Delmas of the School of Computer Science at the University of Auckland. Beyond them, other people who have been notably helpful to me (mostly through the MLton and Manticore mailing lists, and the Racket Slack workspace) at some point through the course of this work include: Lars Bergstrom; José Calderón Trilla; Kevin Chalmers; Kavon Farvadin; Matthew Flatt; Matthew Fluet; Suresh Jagannathan; Laurent Orseau; Alex Potanin; Ivan Raikov; Yawar Raza; John Reppy; Bhargav Shivkumar; KC Sivaramakrishnan; Jens Axel Søgaard; Sam Tobin-Hochstadt; Lukasz Ziarek; and undoubtedly others who I have missed.
Thank you to all of you!

References I

[1] Kevin Chalmers. 'What are Communicating Process Architectures? Towards a Framework for Evaluating Message-passing Concurrency Languages'. In: *Communicating Process Architectures 2017*. Ed. by J. B. Pedersen et al. IOS Press, 2017, pp. 225–252.

[2] Avik Chaudhuri. 'A concurrent ML library in concurrent Haskell'. In: *ACM SIGPLAN Notices* 44.9 (2009), pp. 269–280. DOI: 10.1145/1596550.1596589.

[3] James Cooper. 'Concurrent ML as an Alternative Parallel Programming Style for Image Processing'. In: *2018 International Conference on Image and Vision Computing New Zealand (IVCNZ)*. Vol. 2018-Novem. Auckland, New Zealand: IEEE, Nov. 2018, pp. 1–6. DOI: 10.1109/IVCNZ.2018.8634712.

[4] KEVIN DONNELLY and MATTHEW FLUET. 'Transactional Events'. In: *Journal of Functional Programming* 18.5-6 (2008), pp. 649–706. DOI: 10.1017/s0956796808006916.

[5] Laura Effinger-Dean, Matthew Kehrt and Dan Grossman. 'Transactional events for ML'. In: *ACM SIGPLAN Notices* 43.9 (2008), pp. 103–114. DOI: 10.1145/1411204.1411222.

References II

[6] Matthias Felleisen et al. 'The racket manifesto'. In: *Leibniz International Proceedings in Informatics, LIPIcs*. Vol. 32. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015, pp. 113–128. DOI: 10.4230/LIPIcs.SNAPL.2015.113.

[7] Matthew Fluet. 'The manticore project'. In: *Proceedings of the 2nd ACM SIGPLAN workshop on Functional high-performance computing - FHPC '13*. New York, New York, USA: ACM Press, 2013, p. 1. DOI: 10.1145/2502323.2508150.

[8] Charles Antony Richard Hoare. *Communicating sequential processes*. Prentice-Hall international series in computer science. Englewood Cliffs, N.J.: Prentice/Hall International, 1985, p. 256.

[9] John Reppy, Claudio V. Russo and Yingqi Xiao. 'Parallel concurrent ML'. In: *ACM SIGPLAN Notices* 44.9 (Aug. 2009), pp. 257–268. DOI: 10.1145/1631687.1596588.

References III

[10] K. C. SIVARAMAKRISHNAN, LUKASZ ZIAREK and SURESH JAGANNATHAN. 'MultiMLton: A multicore-aware runtime for standard ML'. In: *Journal of Functional Programming* 24.6 (Nov. 2014), pp. 613–674. DOI: 10.1017/S0956796814000161.

[11] Aaron Turon. 'Reagents'. In: *ACM SIGPLAN Notices* 47.6 (Aug. 2012), pp. 157–168. DOI: 10.1145/2345156.2254084.

[12] Stephen Weeks. 'Whole-program compilation in MLton'. In: *Proceedings of the 2006 workshop on ML - ML '06*. New York, New York, USA: ACM Press, 2006, pp. 1–1. DOI: 10.1145/1159876.1159877.

Message Passing

- CML builds upon Communicating Sequential Processes [8]
- Goes beyond CSP (and Go) with its 'events'
- "Higher-order concurrent programming" (per Reppy)
- Events make synchronisation a first-class value
- Event combinators permit specification of abstracted protocols

Beyond Go?

- Go has channels, and selection over channels
- Channels fixed at compile time (modulo reflection)
- CML has selection over dynamic list of channels
- Go only has (blocking) send and receive
- Send and receive events represent communication *in potentia*
- CML is arguably a ‘superset’ of CSP

Some CML Types

Blocking:

```
send: ('a chan * 'a) -> unit
recv: 'a chan -> 'a
```

Non-Blocking:

```
sendEvt: ('a chan * 'a) -> unit event
recvEvt: 'a chan -> 'a event
```

Blocking:

```
sync: 'a event -> 'a
```

Event Combinators

- `wrap`: Resolve input event, and then execute a supplied function on the result
- `guard`: Immediately prior to input event's resolution, run another supplied function and use result for resolving the event
- `withNack`: Provide a second function for cancellation/if an event is *not* selected
- `choose`: Create a new event that represents the first event from a list to become available
- `select`: `sync` $\circ$ `choose`

Why not X?

- More message passing models/libs/langs than just CML
- CSP-derived – usually limited vs. CML
- Actors – asynchronous, unbounded max memory
- Reagents [11] – Join Calculus-based
- Rendezvous (e.g. Ada, Eiffel's SCOOP) – not channel-based message passing (maybe splitting hairs)
- All good future work targets (probably)

Comparative Benchmarks I

- Assess running times, determine ‘fastest’ ‘language’
- Tested MLton and both Typed & Untyped Racket
- Implemented six exemplar test programs (See `https://github.com/jcoo092/CML_benchmarks`):
  - Communications Time
  - Linear Algebra
  - Monte Carlo Pi
  - Selection Time
  - Spawn
  - Whispers

Comparative Benchmarks II

- Inspiration for some of them was taken from [1] (see also `https://github.com/kevin-chalmers/cpa-lang-shootout`) and [9]
- Linear algebra tests matrix/vector addition and multiplication – Linear Algebra is used heavily in Computer Vision.
- Monte Carlo Pi tests parallelism/multithreading effectiveness
- Other four test aspects of the CML implementation [1]
- Communications Time measures speed at enumerating natural numbers via four threads in a specific arrangement

Comparative Benchmarks III

- Selection Time measures the time taken to 'select' over a list of channels, when one side blocks awaiting communication before the other offers on a randomly-selected channel
- Spawn measures the time taken to create CML threads
- Whispers measures message passing speed sans other computation, using different communication topologies

Whispers I

- Measures message passing speed using different communication topologies
- Independently conceived, but done in the past
- Intent was for three styles:
  - Ring – all threads arranged such that they receive from one thread, and send to another, forming a logical ring
  - Grid – Threads are arranged in a logical grid, and exchange messages with their 'neighbours' above, below and to the left and right
  - K_n (aka all-to-all) – all threads send and receive to every other thread, as on a complete graph

Whispers II

- Racket's `Places` make this awkward to implement
- Only implemented Ring in Racket, so only tested Ring on both Racket and MLton
- Other two are certainly possible, but were expected at the time to be overly lengthy to program

Experimental Process

- Assumption of long-running program. E.g. robot running stereo matching continuously as part of vision system
- Run in a virtual machine
- Automated with a Makefile
- Input & output on command line
- Timings collected using `hyperfine`[2]
- Vary number of iterations to perform, and problem size
- Hyperfine stores results in files for later analysis

[2] `https://github.com/sharkdp/hyperfine`

Threats to Validity

- Take results with pinch of salt
- Unfamiliar with tested languages before start of this work
- Not necessarily representative of the capabilities of each language for an expert
- Does provide a test of how easy it is to get fast programs from each language when previously unfamiliar with it
- First execution usually slower than others (?)
- (Even though three warm up runs were used at the beginning of each test)