THE UNIVERSITY OF AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

SCIENCE
SCHOOL OF COMPUTER SCIENCE

24 November 2020

An ideal match?

Investigating how well-suited Concurrent ML is to implementing Belief Propagation for Stereo Matching

James Cooper
jcoo092@aucklanduni.ac.nz

THE UNIVERSITY OF AUCKLAND Te Whare Wānanga o Tāmaki Makaurau NEW ZEALAND | SCIENCE SCHOOL OF COMPUTER SCIENCE

Outline I

Outline

Outline

Stereo Matching Generally

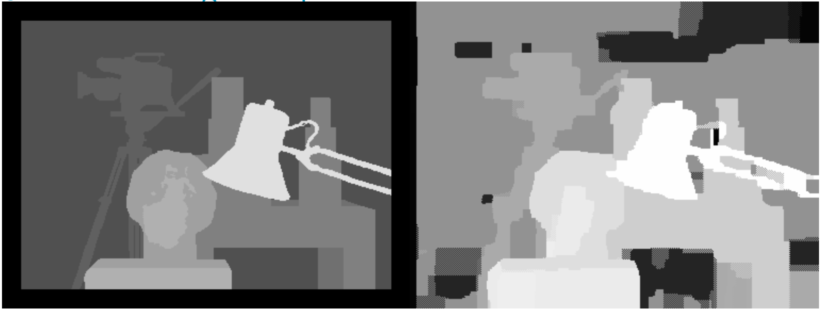
- SM is finding correspondences between stereo images
- Images are of the same scene
- Captured simultaneously
- Correspondences (‘disparity’) are used to estimate depth
- SM is an ill-posed problem – can only make best guess
- Impossible to perform ‘perfectly’ in general case

Stereo Matching Example I

(a) Left camera's image (b) Right camera's image

Figure 1: The popular 'Tsukuba' example stereo matching images, so called because they were created by researchers at the University of Tsukuba, Japan. They are probably the most widely-used benchmark images in stereo matching.

Stereo Matching Example II



(a) Ground truth disparity map

(b) Disparity map generated using a simple Belief Propagation Stereo Matching implementation

Figure 2: The ground truth disparity map for the Tsukuba images, and an example of a possible real disparity map produced by using Belief Propagation Stereo Matching. The ground truth represents what would be expected if stereo matching could be carried out 'perfectly'.

Stereo Matching Example III

Figure 2b was generated using the program at `https://github.com/jcoo092/stereo-matching-practice/tree/master/rust` while the other three are from the Middlebury benchmarks image sets [23] (see `https://vision.middlebury.edu/stereo/data/`).

Stereo Matching Visualisation

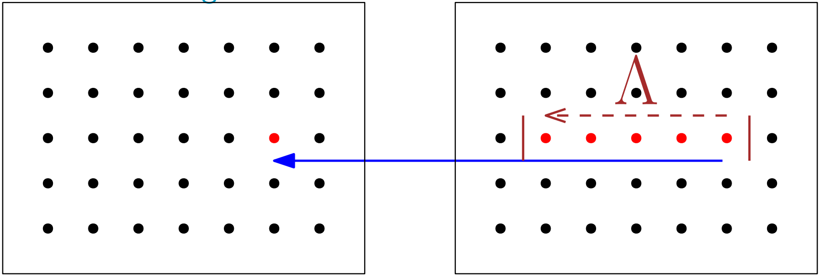


Figure 3: The concept of basic stereo matching. A specific point in one image is compared to points in the corresponding line in the other image, to decide which points match to each other. The shift in position along the line is called the 'disparity'. This disparity, when combined with some information about the cameras, can be used to estimate the distance from the cameras to photographed objects. An example simple comparison would be taking the absolute difference in pixel colours, where the smallest difference implies the most likely match.

Outline

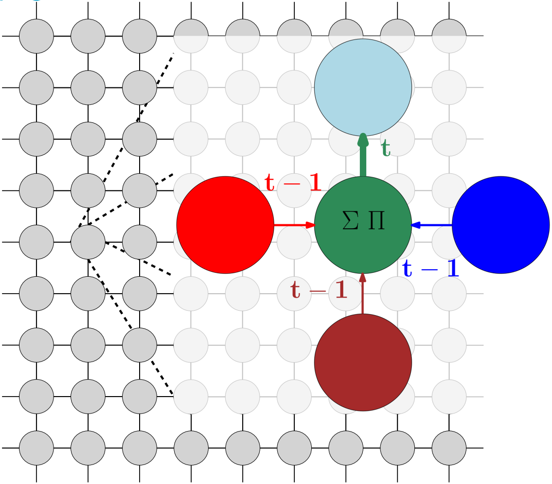
Belief Propagation

- BP is an approach to estimating marginal probabilities over Markov Random Fields & similar [12]
- Introduced by Pearl for inference over factor graphs [20]
- Explicitly based around message passing
- Nodes in graph 'tell' their neighbours their belief of the likelihood of each state
- Family of algorithms these days [3]

Belief Propagation for Stereo Matching

- Adds 'smoothness' costs to 'data' costs
- Early successful example of 'global' SM [23]
- The hidden states of the MRF are the possible disparity values
- Convergence in BP only guaranteed for trees
- SM over images is on a grid – no guarantee of convergence
- SM uses 'Loopy Belief Propagation' [26]
- Stop after 'convergence', or arbitrary # of iterations (*probably* converged?)

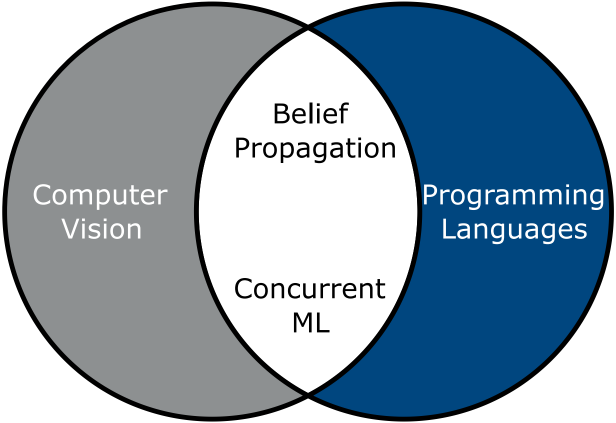
Belief Propagation Visualisation



So What?

- Belief Propagation is explicitly based on the idea of individual processing elements exchanging messages
- Researchers' implementations, however, do not resemble that
- Usually, some nested `for` loops [11]
- How about a message-passing-based programming approach?
- Concurrent ML appears to be an almost-exact theoretical fit

Overlap


Computer Vision
Belief Propagation
Concurrent ML
Programming Languages

Research Question

Can switching to a message-passing programming style like Concurrent ML improve the implementations of Belief Propagation and related algorithms in some way, relative to the typical imperative approach?

Hypothesised Potential Improvements

- Time/memory efficiency
- Correspondence between theory and code
- Code 'quality'
- Use of all of available hardware resources
- Algorithm scheduling
- Scaling across hardware 'sizes'
- 'Future-proof'

Outline

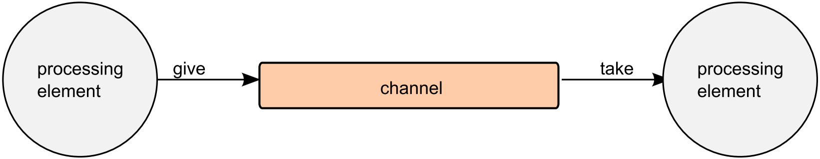
Outline

Concurrent ML

- Originally created by Reppy for his PhD dissertation [22]
- First implemented in Standard ML of New Jersey
- Shared-memory only
- Initially for concurrent but not parallel programming
- Separate processing elements communicate via channels
- Used successfully to create the eXene windowing system
- 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



Message Passing I

- Different from Actors [1] – CML is synchronous communication by anonymous logical processing elements via independent channels
- CML builds upon Communicating Sequential Processes [18]
- (In actuality, CML, Go and many others are arguably closer to Pi Calculus [19] than CSP, but CML predates Pi Calculus, and CSP is still the theoretical model that many languages refer to as their inspiration for concurrency.)
- Goes beyond CSP (and Go), however, with its 'events'

Message Passing II

- "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` ∘ `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 [28] – Join Calculus-based
- Rendezvous (e.g. Ada, Eiffel's SCOOP) – not channel-based message passing (maybe splitting hairs)
- All good future work targets (probably)

Outline

Strict CML implementation requirements

- Parallel CML – aiming for 'real-time' stereo matching
- ‘Green threads’ or similar - distributes work automatically
- Still maintained
- Tail-call-optimisation/last-call-optimisation
- Available for common Linux distributions
- Supports AMD64 and ARM architectures

Nice-to-have CML implementation requirements

- Ahead-of-time compilation/fast 'standalone' executables
- Support for image file IO
- Support for common image processing routines
- Pre-existing stereo matching implementations available
- Support for SIMD/data-parallel/GPU programming
- C interop/Foreign Function Interface
- Well-documented

Possibilities I

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

Photo by Andrea Piacquadio from Pexels

Possibilities II

- *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; could go on...
- All of these seem to stop at CSP (at best)
- Not necessarily parallel, either
- Many also fail to meet at least one other criterion

Possibilities with CML support I

If one accepts not meeting some criteria:

- F# with Hopac library (`https://github.com/Hopac/Hopac`)
- Haskell with Control.Concurrent.CML [6] or Transactional Events [8]
- Guile Scheme (`https://www.gnu.org/software/guile/`) with Fibers library (`https://github.com/wingo/fibers`)
- Manticore [13]
- MLton [30]

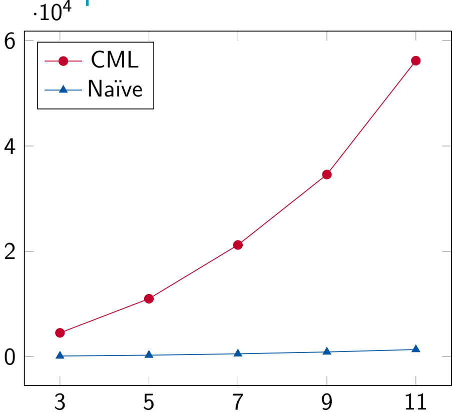
Possibilities with CML support II

- OCaml with Events module (`https://caml.inria.fr/pub/docs/manual-ocaml/libref/Event.html`) (see also [9])
- Racket [10]

Hopac

- Tried out Hopac earlier
- Compared CML with simple imperative nested-`for`-loops approach for median filter [7]
- 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

Hopac Comparative Results



	Naïve	CML
3	131	4,527
5	288	11,006
7	549	21,210
9	899	34,587
11	1,361	56,196

Comparison of mean running time results in milliseconds between simple nested-`for`-loop approach and Hopac for median filter implementations with varying window sizes, on a ˜ 1 megapixel image

Elimination I

- Hopac has issues
- Haskell's libraries are old and long unmaintained; research prototypes only
- Guile doesn't do AOT compilation & standalone executables (it looks like you might be able to write a program in Guile as a library, and embed that in a dummy C program, but it would probably still be relatively slow, and very awkward)
- Manticore seems to be similar to MLton, but AMD64-only
- MLton – selected

Elimination II

- OCaml with Events library – single-core-only. Not clear if it will work with Multicore OCaml, but if it does, would be worth revisiting when MC-OCaml is officially released
- Racket – selected

MLton

- Somewhat-maintained, largely-stable SML variant
- Includes a near-complete port of CML
- Emits C, targets GCC backend, supports many architectures
- Haphazardly documented
- A handful of rough edges
- Few extra libraries

Racket

- Scheme LISP implementation – untyped & typed
- *Mostly* re-implements CML in its 'sync' library
- Reasonably well-documented
- Active and (generally) friendly community
- Wide array of libraries available
- Doesn't *really* do standalone executables (simply bundles core runtime into executable – very large Hello World)

Racket's `Places`

- Racket's favoured approach to parallelism differs from usual task-stealing workpool approach
- Racket uses `Places` [27]
- Each place (roughly) corresponds to an OS thread
- Spawn green threads to run atop a place
- No automatic multiplexing across CPU cores
- Sending messages to specific green threads on other `Places` becomes a two-level process

Outline

Comparative Benchmarks I

- Assess running times, determine ‘fastest’ ‘language’
- 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 [5] (see also `https://github.com/kevin-chalmers/cpa-lang-shootout`) and [21]
- Communications Time measures speed at enumerating natural numbers via four threads in a specific arrangement
- Linear algebra tests matrix/vector addition and multiplication – Linear Algebra is used heavily in Computer Vision.
- Monte Carlo Pi tests parallelism/multithreading effectiveness

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

- 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
- Racket's `Places` make this awkward to implement

Whispers II

- 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
- The other two were nevertheless implemented in MLton

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 analysis

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

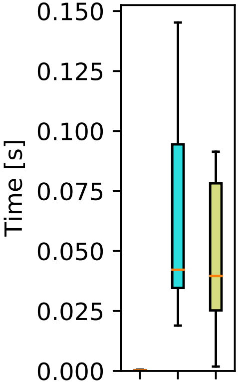
Running Time Data Processing I

- *At least* 10 runs per test instance
- Removed detected outliers
- Focused on average, *probably* reflects ‘real’ running times for an ongoing process
- Used minimum value for each test & language as a baseline
- In theory, baseline represents the best estimate of the actual unavoidable program start-up/finish overhead, meaning further running time is actual relevant work

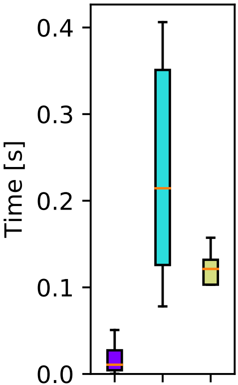
Running Time Data Processing II

- Baseline subtracted from all other test results
- Should (hopefully) help to avoid bias unrelated to specific test

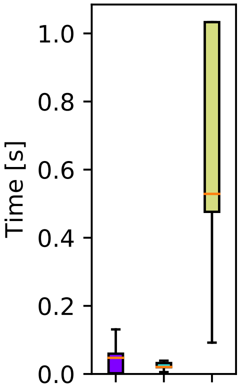
Results Box & Whisker Plots 1



Commstime

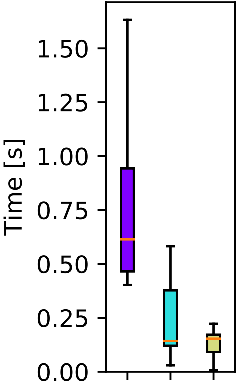


Linear Algebra (Matrix)

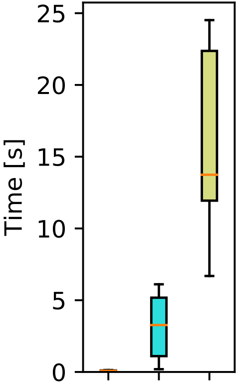


Monte Carlo Pi

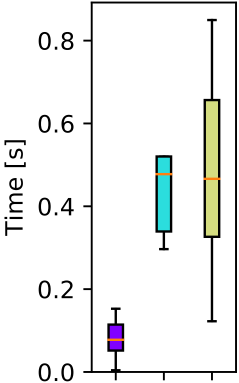
Results Box & Whisker Plots 2



Select Time



Spawn



Whispers (Ring)

Threats to Validity I

- 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 (?)

Threats to Validity II

- (Even though three warm up runs were used at the beginning of each test)
- Linear Algebra wasn't necessarily optimal – Racket has a third-party wrapper over BLAS. Not used to avoid biasing the results against MLton.

There's a Catch...

- MLton outperforms Racket (both typed and untyped) on all but two benchmarks: Selection and Monte Carlo Pi
- MCP tests parallelism capabilities of the language
- MCP tests show that 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. Confirmed by monitoring CPU use in `htop` and via MLton mailing list
- MultiMLton [24] – now dead and buried

Here's Another One...

- Racket's `Places` make a 'direct' BP approach awkward
- `Places` were retrofitted onto a sequential language
- Rumours (unconfirmed) in Racket community that `Places` scale poorly beyond 8 or so cores
- Message passing in `Places` is also relatively slow [27] – for sharing memory, not so much for communications-heavy programs
- Racket perhaps less-than-ideal for testing CML BP

Where to Now?

- MLton is a fairly good implementation of SML
- Lack of multithreading makes 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, also created by Reppy & co in late 2000s
- Standard ML-esque (non-compliant with SML standard)
- Explicitly for testing parallelism designs
- Includes a parallel implementation of CML
- Also, parallel tuples and parallel arrays + array comprehensions
- Reppy & co implemented just enough for research goals

Challenges with Manticore I

- Porting from MLton proving 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 – relying on reading source, plus some exemplar benchmark programs and trial & error
- Monte Carlo Pi is parallel, but large fixed time jump
- Benchmarks still work-in-progress currently

Outline

CML & BP: A good fit?

- Haven't actually been able to investigate this yet...
- Progress much slower than hoped/anticipated
- Lost considerable time to the various language issues
- Also been working on other things (not presented here)
- Earlier Hopac work gives possible indication
- Will need to complete project in Racket and/or Manticore before making final judgement

Refined Hypotheses

- CML approach will be less time/memory efficient with respect to heavily numerical computation, as in Computer Vision
- Much closer to the theory, however
- Maybe better code quality (?)
- Also, better scalability over multiple cores
- I.e. More future-proof than traditional implementations of Belief Propagation – thread multiplexing implies good scaling over cores
- Lack of global synchronisation keeps system running

Sliding Scale I

- Issue of granularity
- One thread per disparity map pixel?
- Traditional approach is arguably 1 thread per image
- Continuum of # of pixels per thread
- Where is the 'sweet spot'?
- One thread per pixel, or one thread per map, arguably at extremes of continuum
- Intra- and inter-thread communications

Sliding Scale II

- How would a `parallel for` loop (as likely the simplest approach to introduce parallelism to an existing implementation) fit with this?
- Scope for task parallelism *and* data parallelism
- Optimal number of pixels per thread almost certainly will be at least enough to fill registers for vector/SIMD instructions

Outline

Outline

Summary I

- Stereo matching: correspondences between two or more images of the same scene
- Using correspondences/'disparity' information about cameras used, can estimate depth to objects in the scene
- Belief Propagation based on message-passing on a graph – in SM, the nodes are (essentially) pixels in an output image
- Concurrent ML seems like a clear theoretical fit to BP
- Investigated CML options

Summary II

- Turns out there are few usable ones out there now
- No ‘ideal’ candidate
- Had tried F# + Hopac earlier – poor performance, possible memory leak
- Decided to test out MLton and Racket
- MLton much faster than Racket, but actually not parallel
- Changed focus to Manticore
- Language issues have slowed work

Summary III

- Haven't yet tested BP + CML
- Suspect it will be slower than usual implementations
- Code will be closer to theory, though
- Quite possibly will scale to manycore better too
- Optimum probably between extremes
- Results may vary heavily between implementations

Outline

Future Directions

- Refine design of benchmark programs
- Improve experiment design
- More CML implementations & host languages [2, 25, 31]
- Other message-passing/rendezvous models [4, 29]
- Other base algorithms, e.g. Semi-global Matching [15, 17], Concurrent Propagation [14]
- How to 'fake' message passing in shared-memory? [16]
- Other hardware, e.g. GPUs, Intel CPUs' TGX instructions

Acknowledgements

Thanks must firstly be given to my supervisors, Dr Radu Nicolescu and Assoc-Prof 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; 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 (my apologies!).

Thank you to all of you!

Outline

References I

[1] Gul A Agha. *ACTORS : a model of concurrent computation in distributed systems*. MIT Press series in artificial intelligence. Cambridge, Mass.: MIT Press, 1986, p. 190.

[2] V. Allombert, F. Gava and J. Tesson. 'Multi-ML: Programming Multi-BSP Algorithms in ML'. In: *International Journal of Parallel Programming* 45.2 (2017), pp. 340–361. DOI: 10.1007/s10766-016-0417-6.

[3] Andrew Blake, Pushmeet Kohli and Carsten Rother, eds. *Markov Random Fields for Vision and Image Processing*. Cambridge, Mass.: The MIT Press, 2011. DOI: 10.7551/mitpress/8579.001.0001.

[4] Frank De Boer et al. 'A Survey of Active Object Languages'. In: *ACM Computing Surveys* 50.5 (Oct. 2017), pp. 1–39. DOI: 10.1145/3122848.

[5] 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.

References II

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

[7] 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.

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

[9] 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.

[10] 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.

References III

[11] Pedro F. Felzenszwalb and Daniel P. Huttenlocher. 'Efficient Belief Propagation for Early Vision'. In: *International Journal of Computer Vision* 70.1 (Oct. 2006), pp. 41–54. DOI: 10.1007/s11263-006-7899-4.

[12] Pedro F. Felzenszwalb and Ramin Zabih. 'Dynamic Programming and Graph Algorithms in Computer Vision'. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 33.4 (Apr. 2011), pp. 721–740. DOI: 10.1109/TPAMI.2010.135.

[13] 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.

[14] Georgy Gimel'Farb et al. 'Concurrent propagation for solving ill-posed problems of global discrete optimisation'. In: *Proceedings - International Conference on Pattern Recognition*. Icpr. Tsukuba, Japan: IEEE, 2012, pp. 1864–1867.

References IV

[15] Rui Gong, Georgy Gimel'farb and Patrice Delmas. 'Semi-global stereo matching under large and spatially variant perceptive deviations'. In: *2015 International Conference on Image and Vision Computing New Zealand (IVCNZ).* Vol. 2016-Novem. 1. IEEE, Nov. 2015, pp. 1–6. DOI: 10.1109/IVCNZ.2015.7761553.

[16] Raphael Hiesgen, Dominik Charousset and Thomas C. Schmidt. 'OpenCL Actors – Adding Data Parallelism to Actor-Based Programming with CAF'. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics).* Ed. by Alessandro Ricci and Philipp Haller. Vol. 10789. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 59–93. DOI: 10.1007/978-3-030-00302-9.

[17] Heiko Hirschmüller. 'Stereo processing by semiglobal matching and mutual information'. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 30.2 (Feb. 2008), pp. 328–341. DOI: 10.1109/TPAMI.2007.1166. arXiv: 1607.08085.

References V

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

[19] Robin Milner. 'Elements of Interaction: Turing Award Lecture'. In: *Communications of the ACM* 36.1 (Jan. 1993), pp. 78–89. DOI: 10.1145/151233.151240.

[20] Judea Pearl. 'Reverend Bayes on inference engines: A distributed hierarchical approach'. In: *Proceedings of the AAAI National Conference on AI* (1982), pp. 133–136.

[21] 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.

[22] John H. Reppy. *Concurrent Programming in ML*. New York, New York, USA: Cambridge University Press, 2007, p. 308.

References VI

[23] Daniel Scharstein and Richard Szeliski. 'A Taxonomy and Evaluation of Dense Two-Frame Stereo Correspondence Algorithms'. In: *International Journal of Computer Vision* 47.1 (Apr. 2002), pp. 7–42. DOI: 10.1023/A:1014573219977.

[24] 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.

[25] KC Sivaramakrishnan et al. 'Lightweight asynchrony using parasitic threads'. In: *Proceedings of the 5th ACM SIGPLAN workshop on Declarative aspects of multicore programming - DAMP '10*. New York, New York, USA: ACM Press, 2010, p. 63. DOI: 10.1145/1708046.1708059.

[26] Jian Sun, Nan Ning Zheng and Heung Yeung Shum. 'Stereo matching using belief propagation'. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 25.7 (July 2003), pp. 787–800. DOI: 10.1109/TPAMI.2003.1206509.

References VII

[27] Kevin Tew et al. 'Places: Adding Message-Passing Parallelism to Racket'. In: *ACM SIGPLAN Notices* 47.2 (Mar. 2012), pp. 85–96. DOI: 10.1145/2168696.2047860.

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

[29] Carlos A Varela. *Programming Distributed Computing Systems : A Foundational Approach*. Cambridge, Mass.: MIT Press, 2013, p. 296.

[30] 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.

[31] Lukasz Ziarek and Suresh Jagannathan. *Featherweight Threads for Communication Featherweight Threads for Communication*. Tech. rep. Purdue University, Oct. 2011.