Efficient and Scalable Parallel Functional Programming Through Disentanglement

Sam Westrick
Carnegie Mellon University

ML Workshop
Ljubljana, Slovenia
September 2022
Parallel Hardware Today

Apple A14: 12 cores

Apple S4: 2 cores

nVidia GeForce 3090: 10496 (CUDA) cores

AMD Epyc: 64 cores

AMD Ryzen Threadripper: 16 cores

4x Intel Xeon E7: 72 cores
Parallel Programming

imperative

- mutability (in-place updates)
- manual memory management
- race conditions

functional

- immutability
- automatic memory management
- deterministic by default

fast

can parallel functional programming be fast and scalable?

slow?
Parallel Programming

imperative
- mutability (in-place updates)
- manual memory management
- race conditions

functional
- immutability
- automatic memory management
- deterministic by default
- high rate of allocation
- heavy reliance on GC

- fast
- can parallel functional programming be fast and scalable
- slow?
Sequential

Parallel
Is there a better way?
"concurrent tasks remain oblivious to each other’s allocations"
MaPLe Compiler

- based on MLton, **full Standard ML language**, extended with

  ```
  val par: (unit -> 'a) * (unit -> 'b) -> 'a * 'b
  ```

- parallel memory management based on disentanglement

- used by 500+ students at CMU each year

github.com/mpllang/mpl
Parallel ML Benchmark Suite

- over 30 state-of-the-art parallel algorithms
  - ported from highly-optimized C++ benchmark suites (PBBS, GBBS, Ligra, PAM, ...)
- all disentangled
- MPL has excellent parallel time and space performance
- same memory footprint as C++ (on average)
- generally within 2x time of hand-optimized C++
  - e.g. linefit (±5%), sparse matrix-vector mult (±10%),
    mergesort (1.3x), nearest-neighbors (1.7x),
    tokenization (1.7x), delaunay triangulation (2.3x)
**Speedup** (higher is better)

- tinykaboom
- raytracer
- range-query
- mergesort
- triangle-count
- dense matmul
- tokenization
- grep
- max-indep-set
- palindrome
- nearest nbrs
- centrality
- low-d-decomp
- suffix-array
- bfs
- reverb
- dedup
- quickhull
- delaunay
- seam-carve

**Space Blowup** (lower is better)

- MPL (72 processors)
- MLton (sequential baseline)

14-60x speedup, often with less space (average: -30%)
MPL vs Java and Go (on 72 processors)

average vs Go:
2x faster
30% less space

average vs Java:
3x faster
4x less space
can parallel functional programming be fast and scalable?

YES:

• MPL can outperform existing implementations of parallel languages
• MPL can compete with low-level optimized C++ code
Disentanglement
Disentanglement

- observed in efficient parallel code:
  concurrent tasks are oblivious to each other’s allocations

- in computation graph:
  allocation precedes use

- arbitrary? no:
  guaranteed by
determinacy-race-freedom
  [Westrick et al. 2020]

<table>
<thead>
<tr>
<th>graphs</th>
<th>betweenness centrality, breadth-first search, minimum spanning tree, maximum independent set, low-diameter decomposition, triangle counting</th>
</tr>
</thead>
<tbody>
<tr>
<td>geometry</td>
<td>delaunay triangulation, nearest neighbors, quickhull, 2D range query</td>
</tr>
<tr>
<td>images</td>
<td>seam carving, raytracing, tinylaboom, GIF encode+decode</td>
</tr>
<tr>
<td>audio</td>
<td>WAV encode+decode, reverb</td>
</tr>
<tr>
<td>text</td>
<td>tokenization, deduplication, grep, word-count, longest palindrome, suffix array</td>
</tr>
<tr>
<td>numeric</td>
<td>dense+sparse matrix mult, integration, linear regression</td>
</tr>
</tbody>
</table>
Disentanglement

How to utilize disentanglement for improved efficiency and scalability?

idea: organize memory to reflect structure of parallelism: concurrent execution ⇔ memory separation
Nested Fork/Join Parallelism

classic and popular (as programming model and/or execution model):

- Cilk, ParlayLib, Intel TBB, Microsoft TPL, OpenMP, Legion, Rayon, Fork/Join Java, Habanero Java, X10, multiLisp, Id, NESL, parallel Haskell, Manticore, Futhark, SML#, etc.
Task-Local Heaps

fork (spawn)

join (sync)
Task-Local Heaps

fork (spawn) → fresh empty heaps → join (sync) → merge heaps into parent
Disentangled Memory Management

- disentanglement: *no cross-pointers*  
  (up-pointers are down-pointers are allowed)
Disentangled Memory Management

- disentanglement: *no cross-pointers*
  (up-pointers are down-pointers are allowed)
- subtree collection

reorganize, compact, etc. inside subtree

naturally parallel
Disentangled Memory Management

- disentanglement: no cross-pointers
  (up-pointers are down-pointers are allowed)
- subtree collection
- internal concurrent collections

Concurrent non-moving GC
(no descendant pauses)

Reorganize, compact, etc.
inside subtree

Naturally parallel
**LGC**
- heaps local to one processor
- compactifying (copying) GC

**CGC**
- heaps with at least 2 active descendants
- concurrent non-moving mark-sweep
- snapshot-at-the-beginning (SATB)

**Notes:**
- write barrier for remembered sets (for SATB, and down-pointers)
- never stops the world
- no promotions necessary
- LGC+CGC $\rightarrow$ provable efficiency
  [Arora et al. 2021]
Ensuring Disentanglement
**Theorem**  [Westrick et al. POPL 20]

determinacy-race-free programs are disentangled

Intuition

- if entangled, must be a **read/write** race
- **write**: creates down-pointer
- **read**: discovers data across

Proof idea

- **single-step invariant**: if location $X$ accessible without a race, then $neighbors(X)$ are in root-to-leaf path
- carry invariant through race-free execution
fully general

disentangled

effectful and race-free

mutation-free (e.g. purely functional)
Entanglement Detection

Algorithm
- build computation graph during execution
- annotate allocated locations with current vertex
- check results of memory reads
  - disentangled: result allocated before current vertex
  - otherwise, entanglement detected

sound (no missed alarms) and complete (no false alarms)
provably efficient (work, span, and space)
[Westrick et al. ICFP 22]

Implementation and Evaluation:
- nearly zero overhead (±5%) for both time and space
- read-barrier on mutable pointers only
- SP-order maintenance
Writing Disentangled Programs
Writing Disentangled Programs

pure library interface
- tabulate
- filter
- map
- flatten
- reduce
- merge
- scan
- ...

purely functional, parallel, disentangled algorithms

fun mergesort(X) =
  if length(X) <= granularity then
    quicksort(X)
  else
    let
      val (L,R) = split(X)
      val (sL,sR) = par(fn _ => mergesort(L),
        fn _ => mergesort(R))
    in
      merge(sL,sR)
    end

no need to know about disentanglement!

fast implementation w/ “local” effects
- ...

only 10% more time+memory than hand-optimized
Writing Disentangled Programs

- Pure library interface
  - tabulate
  - filter
  - map
  - flatten
  - reduce
  - merge
  - scan
  - ... 

- Fast implementation w/ “local” effects
  - ...

- Purely functional, parallel, disentangled algorithms

- Parenthood matching
- Max contiguous subsequence
- Prime sieve
- Sorting
- Order statistics
- Range query
- Graph search
- Connected components
- Shortest paths
- Minimum spanning forest
- Dynamic programming
- Hashing
- ... 

15-210 (Undergrad Course)
Parallel and Sequential Data Structures and Algorithms

No need to know about disentanglement!
Writing Disentangled Programs

- **pure library interface**
  - tabulate
  - filter
  - map
  - flatten
  - reduce
  - merge
  - scan
  - ...  

- **fast implementation w/ “local” effects**
  - ...  

- **mostly purely functional, parallel, disentangled algorithms**

```plaintext
fun forwardBFS(G,s) = 
  let
    fun outEdges(u) = map(fn v => (u,v), neighbors(G,u))
    val parents = tabulate(numVertices(G), fn v => -1)
    fun tryVisit(u,v) = 
      if compareAndSwap(parents,v,-1,u) then SOME(v) else NONE
    fun search(F) = 
      if length(F) = 0 then ()
      else search(filterOp(tryVisit, flatten(map(outEdges, F))))
    in
      tryVisit(s,s);
      search(singleton(s));
      parents
    end
```
Summary

disentanglement
- “concurrent tasks remain oblivious to each other’s allocations”
- common property, guaranteed by race-freedom, functional programming
- enables fully parallel memory management and GC

MaPLe implementation
- fast, scalable, and space-efficient
- competitive with low-level imperative code

Future / Ongoing work
- static enforcement of disentanglement (e.g. type system)
- dynamic “entanglement management”
- distributed computing

github.com/mpillang/mpl