Efficient and Scalable Parallel Functional Programming Through Disentanglement

Sam Westrick
samwestrick.com
@shwestrick

March 2022
Parallel Hardware Today

Apple A14: 12 cores

4x Intel Xeon E7: 72 cores

AMD Ryzen Threadripper: 16 cores

AMD Epyc: 64 cores

nVidia GeForce 3090: 10496 (CUDA) cores

Apple S4: 2 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?**
Is there a better way?
In Existing Functional Languages...

- popular “two-level” design [Doligez-Leroy-Gonthier]
  - used by multicore OCaml, GHC Haskell, Manticore, Caml Light, ...
  - minor and major heaps
  - parallel allocation+GC in minor heaps

- invariants:
  - no cross-pointers between minor heaps
  - restrictions between major and minor heaps

- promotions maintain invariants
  - moving (copying) data from minor to major

- problem: shared data must live in major heap
  - scheduler actions trigger promotions
  - high overhead, no provable efficiency (e.g. unbounded space)
Is there a better way?

Disentanglement

“concurrent tasks remain oblivious to each other’s allocations”
MaPLe Compiler

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

\[
\text{val par: (unit -> 'a) * (unit -> 'b) -> 'a * 'b}
\]

- parallel memory management based on disentanglement
- used by 500+ students at CMU each year
- outperforms existing implementations of functional languages
- competitive with state-of-the-art imperative/procedural (including Java, Go, C/C++)

MPL vs multicore OCaml: ~2x average speedup [1]
MPL vs GHC Haskell: ~2x average speedup [1]
MPL vs Manticore: 2-50x speedup [2]

## Sorting Shootout

<table>
<thead>
<tr>
<th></th>
<th>serial (1 proc)</th>
<th>parallel (72 procs)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$T_1$</td>
<td>$T_{72}$</td>
</tr>
<tr>
<td>C++ std::sort</td>
<td>8.8</td>
<td>–</td>
</tr>
<tr>
<td>Cilk samplesort</td>
<td>7.9</td>
<td>0.16</td>
</tr>
<tr>
<td>Cilk mergesort</td>
<td>12.7</td>
<td>0.24</td>
</tr>
<tr>
<td><strong>MPL (Ours) mergesort</strong></td>
<td><strong>18.8</strong></td>
<td><strong>0.37</strong></td>
</tr>
<tr>
<td>Go samplesort</td>
<td>27.2</td>
<td>0.52</td>
</tr>
<tr>
<td>Java mergesort</td>
<td>11.0</td>
<td>0.63</td>
</tr>
<tr>
<td>Haskell/C mergesort</td>
<td>10.6</td>
<td>1.3</td>
</tr>
</tbody>
</table>

- ~24x speedup over C++ std::sort
- 2\textsuperscript{nd} fastest, behind Cilk
- 40% faster than Go
- 70% faster than Java
Parallel ML Benchmarks

- **all disentangled**
- many ported from highly-optimized C/C++
  - PBBS, Ligra, and PAM benchmark suites
- excellent performance
- in general, **within 2-3x of hand-optimized C/C++**
  - e.g. delaunay triangulation, factor 2
- in some cases, **can match C/C++**
  - e.g. linefit: near optimal on our 72-core machine (max read bandwidth)
MPL (72 processors) vs MLton (sequential baseline)

10-63x speedup, often with less space (!)
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 race-freedom [Westrick et al. 2020]
How to utilize disentanglement for improved efficiency and scalability?

idea: organize memory to reflect **structure of parallelism**
Nested Fork/Join Parallelism

classic and popular technique

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

fork (spawn)

join (sync)
Task-Local Heaps

fork (spawn)

merge heaps
into parent

fresh empty heaps

join (sync)
Disentangled Memory Management

- disentanglement: *no cross pointers*
Disentangled Memory Management

- disentanglement: *no cross pointers*
- subtree collection

Diagram:
- reorganize, compact, etc. inside subtree
- naturally parallel
Disentangled Memory Management

- disentanglement: *no cross pointers*
- subtree collection
- internal collections and provable efficiency
  [Arora et al. POPL 21]

Implementation Notes:
- carefully integrated with scheduler
  - new heaps only on steals
- write barrier for down-pointers
- no read barrier
  - *no promotions necessary*

reorganize, compact, etc. inside subtree

naturally parallel
Heap Scheduling

- goal: assign heaps to processors
- each processor manages its own memory
Heap Scheduling

- goal: assign heaps to processors
- each processor manages its own memory
- integrate closely with thread scheduling (work-stealing)
Collection Policy

**algorithm**
- each processor $p$ has local counter $L_p$
- when cumulative size of $p$’s heaps exceeds $k \cdot L_p$:
  - processor $p$ performs GC on its heaps
  - set $L_p$ to amount of memory that survives

**theorem** [Arora et al., POPL 21]
a race-free program with work $W$ and sequential space $R$ requires $O(P \cdot R)$ space and $O(W + P \cdot R)$ work, including costs of memory management

Key idea:
- spines resemble sequential execution
- local counters $L_p$ cannot exceed $R$
fully general

disentangled

race-free

mutation-free
**Theorem** [Westrick et al. POPL 20] all race-free programs are disentangled

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

**Proof Sketch**
- **single-step invariant**: if location $X$ accessible without a race, then $\text{neighbors}(X)$ are in root-to-leaf path
- carry invariant through race-free execution

```
y = malloc()
*x = y
...  
...  
z = *x
```
Writing Disentangled Programs

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

purely functional, parallel, disentangled algorithms

```haskell
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
  - ... 

- Purely functional, parallel, disentangled algorithms:
  - parentheses matching
  - max contiguous subsequence
  - prime sieve
  - sorting
  - order statistics
  - range query
  - graph search
  - connected components
  - shortest paths
  - minimum spanning forest
  - dynamic programming
  - hashing
  - ...

- Fast implementation w/ "local" effects:
  - No need to know about disentanglement!

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

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

fast implementation
w/ “local” effects
...  

mostly purely functional, parallel, disentangled algorithms

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
Writing Disentangled Programs

**Parallel Block-Delayed Sequences**
Sam Westrick, Mike Rainey, Daniel Anderson, and Guy Blelloch
PPoPP’22

- fusion across library calls
  - e.g. only $O(\#\text{processors})$ allocation for `map -> scan -> reduce`
Summary

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

**MaPLe implementation**
- efficient and scalable
- competitive with low-level imperative code

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

github.com/mpillang/mpl