Efficient Parallel Functional Programming with Hierarchical Memory Management

Sam Westrick
Carnegie Mellon University

Joint work with:
Ram Raghunathan, Adrien Guatto, Stefan Muller, Rohan Yadav, Umut Acar, Guy Blelloch, Matthew Fluet
Setting the Stage

- functional programming is good for **expressing** parallelism (no side-effects, no concurrency, no race conditions)

- the point of parallelism is to make things **faster**…
  - absolute efficiency is paramount (speedup w.r.t. fastest sequential solution)

- is parallel functional programming **efficient**?
  - existing implementations achieve good scalability but not absolute efficiency
  - standard challenges: high rate of allocation, heavy reliance upon garbage collection
The Problem

we need
more efficient memory management
for *parallel programs*

(not just functional)
fun msort A = 
  if length A < 2 then A else 
  let 
    val (L, R) = splitMid A 
    val (L', R') = par (fn () => msort L, fn () => msort R) 
    val B = merge L' R' 
  in 
    B 
  end
Example: Mergesort

```plaintext
fun msort A = 
  if length A < 2 then A else
  let
    val (L, R) = splitMid A
    val (L', R') = par (fn () => msort L, fn () => msort R)
    val B = merge L' R'
  in
    B
  end

msort [2,4,3,1]
```
Example: Mergesort

fun msort A =
  if length A < 2 then A else
  let
    val (L, R) = splitMid A
    val (L', R') = par (fn () => msort L, fn () => msort R)
    val B = merge L' R'
  in
    B
  end

par (fn () => msort [2,4], fn () => msort [3,1])
fun msort A = 
  if length A < 2 then A else
  let
    val (L, R) = splitMid A
    val (L', R') = \() => msort L, \() => msort R\)
    val B = merge L' R'
  in
    B
  end
Example: Mergesort

fun msort A = 
  if length A < 2 then A else 
  let 
    val (L, R) = splitMid A 
    val (L', R') = par (fn () => msort L, fn () => msort R) 
    val B = merge L' R' 
  in 
    B 
  end
Example: Mergesort

fun msort A = 
  if length A < 2 then A else 
  let 
    val (L, R) = splitMid A 
    val (L', R') = par (fn () => msort L, fn () => msort R) 
    val B = merge L' R' 
  in 
    B 
  end
fun msort A = 
  if length A < 2 then A else 
  let 
    val (L, R) = splitMid A 
    val (L', R') = par (fn () => msort L, fn () => msort R) 
    val B = merge L' R' 
  in 
    B 
  end
fun msort A =  
  if length A < 2 then A else 
  let 
    val (L, R) = splitMid A 
    val (L', R') = par (fn () => msort L, fn () => msort R) 
    val B = merge L' R' 
  in 
    B 
  end
Example: Mergesort

```ml
fun msort A = 
  if length A < 2 then A else 
  let 
    val (L, R) = splitMid A 
    val (L', R') = par (fn () => msort L, fn () => msort R) 
    val B = merge L' R' 
  in 
  B 
  end 

merge [2,4] [1,3]
```
fun msort A =  
  if length A < 2 then A else  
  let  
    val (L, R) = splitMid A  
    val (L', R') = par (fn () => msort L, fn () => msort R)  
    val B = merge L' R'  
  in  
    B  
  end
Hierarchical Memory Management

fun msort A = 
  if length A < 2 then A else 
  let 
    val (L, R) = splitMid A 
    val (L', R') = \par (fn () => msort L, fn () => msort R) 
    val B = merge L' R' 
  in 
    B 
  end
Hierarchical Memory Management

fun msort A = 
  if length A < 2 then A else 
  let 
    val (L, R) = splitMid A 
    val (L', R') = par (fn () => msort L, fn () => msort R) 
    val B = merge L' R' 
  in 
    B 
  end
Hierarchical Memory Management

fun msort A = 
if length A < 2 then A else 
let 
  val (L, R) = splitMid A 
  val (L', R') = par (fn () => msort L, fn () => msort R) 
  val B = merge L' R'
  in 
    B 
  end
Hierarchical Memory Management

fun msort A = 
  if length A < 2 then A else 
  let 
    val (L, R) = splitMid A 
    val (L', R') = par (fn () => msort L, fn () => msort R) 
    val B = merge L' R' 
  in 
    B 
  end
Hierarchical Memory Management

```haskell
fun msort A = 
if length A < 2 then A else 
let 
  val (L, R) = splitMid A 
  val (L', R') = par (fn () => msort L, fn () => msort R) 
  val B = merge L' R'
in 
  B 
end
```

- fork (spawn)
- fresh empty heaps
Hierarchical Memory Management

fun msort A = 
  if length A < 2 then A else 
  let 
    val (L, R) = splitMid A 
    val (L', R') = par (fn () => msort L, fn () => msort R) 
    val B = merge L' R' 
  in 
    B 
  end
fun msort A = 
  if length A < 2 then A else 
  let 
    val (L, R) = splitMid A 
    val (L', R') = par (fn () => msort L, fn () => msort R) 
    val B = merge L' R' 
  in 
    B 
  end
Hierarchical Memory Management

fun msort A = 
  if length A < 2 then A else 
  let 
    val (L, R) = splitMid A 
    val (L', R') = \par (fn () => msort L, fn () => msort R) 
    val B = merge L' R' 
  in 
    B 
  end
Hierarchical Memory Management

fun msort A =  
  if length A < 2 then A else  
  let  
    val (L, R) = splitMid A  
    val (L', R') = par (fn () => msort L, fn () => msort R)  
    val B = merge L' R'  
  in  
    B  
  end
Hierarchical Memory Management

fun msort A = 
  if length A < 2 then A else 
  let 
    val (L, R) = splitMid A 
    val (L', R') = \par (fn () \Rightarrow msort L, fn () \Rightarrow msort R) 
    val B = merge L' R' 
  in 
    B 
  end
Hierarchical Memory Management

- give each **task** its own **heap**
  - tasks allocate new data inside their own heaps
- organize heaps to mirror the **nesting structure** of tasks
  - fork (spawn, async, etc): fresh heaps for children
  - join (sync, finish, etc): merge heaps into parent
Disentanglement:
in strict purely functional programs, all pointers either point up or are internal
[Raghunathan et al, ICFP’16]
Disentanglement:
in _strict_ purely functional programs,
all pointers either point up or are internal

[Raghunathan et al, ICFP’16]
Disentanglement:
in strict purely functional programs, all pointers either point up or are internal
[Raghunathan et al, ICFP’16]
Disentanglement:
in strict purely functional programs, all pointers either point up or are internal
[Raghunathan et al, ICFP’16]
Local Garbage Collection

pick a subtree

reorganize, compact, etc.
inside subtree
Local Garbage Collection

pick a subtree

reorganize, compact, etc.
inside subtree
Local Garbage Collection

Disentanglement is necessary:
Local Garbage Collection

Disentanglement is necessary:

dangling pointer
Local Garbage Collection

- localized within a subtree of heaps
- independent of
  - tasks whose heaps are outside the subtree
  - other local collections (on disjoint subtrees)
- can easily apply any existing GC algorithm
  - just ignore pointers that exit the subtree
In-place Updates

- often crucial for efficiency, especially under the hood
- but, *can* break disentanglement (not always)

```ocaml
let
  val r = ref []
  fun f () = (r := 0 :: !r)
  fun g () = (r := 1 :: !r)
in
  par (f, g)
end
```
In-place Updates

- often crucial for efficiency, especially under the hood
- but, can break disentanglement (not always)

```ml
let
  val r = ref []
  fun f () = (r := 0 :: !r)
  fun g () = (r := 1 :: !r)
in
  par (f, g)
end
```
In-place Updates

- often crucial for efficiency, especially under the hood
- but, can break disentanglement (not always)

```ml
let
  val r = ref []
  fun f () = (r := 0 :: !r)
  fun g () = (r := 1 :: !r)
in
  par (f, g)
end
```
In-place Updates

- often crucial for efficiency, especially under the hood

- but, *can* break disentanglement (not always)

```
let
  val r = ref []
  fun f () = (r := 0 :: !r)
  fun g () = (r := 1 :: !r)
in
  par (f, g)
end
```
In-place Updates

- often crucial for efficiency, especially under the hood
- but, *can* break disentanglement (not always)

```ocaml
let
  val r = ref []
  fun f () = (r := 0 :: !r)
  fun g () = (r := 1 :: !r)
in
  par (f, g)
end
```
In-place Updates

- often crucial for efficiency, especially under the hood
- but, *can* break disentanglement (not always)

options:

- enforce disentanglement dynamically with promotion
  [Guatto et al, PPoPP’18]

- weaken to permit important classes of effects
  [Westrick et al, work in progress]
Implementation

• extend MLton compiler with fork-join library
  \texttt{val par : (unit \rightarrow 'a) \times (unit \rightarrow 'b) \rightarrow 'a \times 'b}

• block-structured heaps
  \begin{itemize}
    \item heaps are lists of blocks:
      \begin{itemize}
        \item merge heaps in O(1) time
      \end{itemize}
  \end{itemize}

• no read barrier. write barrier only on mutable pointer data

• local collections: sequential Cheney-style copying/compacting

• work-stealing scheduler
  \begin{itemize}
    \item GC policy influenced by scheduler decisions
  \end{itemize}
Runtime Overhead

Ours / MLton, 1 core
Speedups
MLton / Ours, 72 cores

fib, tabulate, map, map-in-place, scan, reduce, filter, samplesort, mergesort, dmm, dedup, histogram, barnes-hut, all-nearest