Entanglement Detection with Near-Zero Cost

SAM WESTRICK, Carnegie Mellon University, USA
JATIN ARORA, Carnegie Mellon University, USA
UMUT A. ACAR, Carnegie Mellon University, USA

Recent research on parallel functional programming has culminated in a provably efficient (in work and space) parallel memory manager, which has been incorporated into the MPL (MaPLe) compiler for Parallel ML and shown to deliver practical efficiency and scalability. The memory manager exploits a property of parallel programs called disentanglement, which restricts computations from accessing concurrently allocated objects. Disentanglement is closely related to race-freedom, but subtly differs from it. Unlike race-freedom, however, no known techniques exists for ensuring disentanglement, leaving the task entirely to the programmer. This is a challenging task, because it requires reasoning about low-level memory operations (e.g., allocations and accesses), which is especially difficult in functional languages.

In this paper, we present techniques for detecting entanglement dynamically, while the program is running. We first present a dynamic semantics for a functional language with references that checks for entanglement by consulting parallel and sequential dependency relations in the program. Notably, the semantics requires checks for mutable objects only. We prove the soundness of the dynamic semantics and present several techniques for realizing it efficiently, in particular by pruning away a large number of entanglement checks. We also provide bounds on the work and space of our techniques.

We show that the entanglement detection techniques are practical by implementing them in the MPL compiler for Parallel ML. Considering a variety of benchmarks, we present an evaluation and measure time and space overheads of less than 5% on average with up to 72 cores. These results show that entanglement detection has negligible cost and can therefore remain deployed with little or no impact on efficiency, scalability, and space.

CCS Concepts: • **Software and its engineering** → **Garbage collection**; **Parallel programming languages**; **Functional languages**; • **Theory of computation** → Parallel algorithms.

Additional Key Words and Phrases: disentanglement, parallelism, functional, memory management

ACM Reference Format:
Sam Westrick, Jatin Arora, and Umut A. Acar. 2022. Entanglement Detection with Near-Zero Cost. *Proc. ACM Program. Lang.* 6, ICFP, Article 115 (August 2022), 32 pages. https://doi.org/10.1145/3547646

1 INTRODUCTION

With the mainstream availability of multicore computers, parallel programming today is important and relevant but remains to be challenging. A key concern is race conditions, which typically raise serious correctness issues. Because pure functional programs are free of data races, functional programming languages can make parallel programming simpler and safer. As a result, many parallel functional programming languages have been developed going back to the 1980's, including multiLisp [Halstead 1984], Id [Arvind et al. 1989], NESL [Blelloch 1996; Blelloch et al. 1994], several forms of parallel Haskell [Hammond 2011; Li et al. 2007; Marlow and Jones 2011; Peyton Jones

Authors' addresses: Sam Westrick, Carnegie Mellon University, USA, swestric@cs.cmu.edu; Jatin Arora, Carnegie Mellon University, USA, jatina@andrew.cmu.edu; Umut A. Acar, Carnegie Mellon University, USA, umut@cs.cmu.edu.

2475-1421/2022/8-ART115
https://doi.org/10.1145/3547646

et al. 2008], and several forms of parallel ML [Fluet et al. 2011; Guatto et al. 2018; Ohori et al. 2018; Raghunathan et al. 2016; Sivaramakrishnan et al. 2014; Spoonhower 2009; Spoonhower et al. 2009; Westrick et al. 2020; Ziarek et al. 2011]. Some of these languages only support pure or mutation-free functional programs, but others such as Parallel ML [Arora et al. 2021; Guatto et al. 2018; Westrick et al. 2020] also allow for side effects. Despite the decades of work, functional programming languages have not delivered on efficiency and scalability. The primary reason for this is their hunger for memory: functional programs have a high rate of allocation, and this rate increases with parallelism, because multiple cores can allocate memory simultaneously [Appel 1989; Appel and Shao 1996; Auhagen et al. 2011; Doligez and Gonthier 1994; Doligez and Leroy 1993; Gonçalves 1995; Gonçalves and Appel 1995; Marlow and Jones 2011]. This problem has proved challenging [Auhagen et al. 2011; Doligez and Gonthier 1994; Doligez and Leroy 1993; Marlow and Jones 2011; Sivaramakrishnan et al. 2020]: provably work- and space-efficient parallel memory managers remained elusive until recently.

Recent work by Arora, Westrick, and Acar [Arora et al. 2021] presented a provably work- and space-efficient parallel memory manager for parallel functional programs. In addition to presenting theoretical results, they also implement their techniques in the MPL compiler for Parallel ML, which extends the Standard ML language with fork-join parallelism. Experiments show good performance and scalability in practice for a range of parallel benchmarks, including both purely functional ones and those that use effects. This work has taken an important step in bridging the gap between functional programming and performance (efficiency and scalability) by relying upon a memory property called ***disentanglement*** [Guatto et al. 2018; Raghunathan et al. 2016; Westrick et al. 2022a, 2020].

The main goal of disentanglement is to achieve efficiency and scalability by enabling parallel tasks to GC independently in parallel. At a high level, disentanglement requires concurrently executing computations to remain oblivious to each other's memory allocations. More specifically, disentanglement restricts access to memory objects that are allocated in the "past", as determined by sequential dependencies. This restriction prevents ***entanglement***—i.e., cross-references—between objects allocated by concurrent tasks during execution. This, in turn, allows each task to manage its own heap (memory), without needing to synchronize with other tasks.

The memory manager of Arora, Westrick, and Acar [Arora et al. 2021] assumes disentanglement, but they do not enforce or check for it. As a result, if an entangled program is executed, the garbage collector will not behave correctly, which could cause the program to crash, or (worse) return an incorrect result. Specifically, in the presence of entanglement, the garbage collector (which assumes disentanglement) might reclaim an object incorrectly by missing a cross-reference. To avoid this unsafe behavior, disentanglement needs to be enforced.

As a starting point, we note that in purely functional programs, there is no need to explicitly enforce or check for disentanglement because entanglement is impossible. That is, in a strict call-by-value language (e.g. Parallel ML), if mutable data is outlawed (e.g. no **ref**s or **array**s), then disentanglement is guaranteed by construction [Raghunathan et al. 2016; Westrick et al. 2020]. However, there is a problem: purely functional programming forbids in-place updates. We wish to allow for in-place updates to improve efficiency. An interesting motivating example (discussed in Section 2.3) is a parallel graph search which interleaves atomic in-place updates. The resulting algorithm is non-deterministic, and this non-determinism is desirable for improving performance. One advantage of disentanglement is that it allows for in-place updates to be utilized in this manner [Westrick et al. 2020].

Therefore, moving beyond purely functional programming, we seek a solution that allows for in-place updates, as long as the program remains disentangled. To this end, one natural approach could be to use a type system to statically prevent entanglement. However, no such type system is known.

Designing a suitable static test for entanglement is challenging: entanglement is an undecidable property, and it emerges due to subtle interaction between non-determinism and aliasing. It seems possible to devise a conservative type system that ensures disentanglement, but this could outlaw perfectly good, disentangled programs. For example, a type system for race-freedom can ensure disentanglement by implication [Westrick et al. 2020], but such a type system would also outlaw the non-deterministic programs that we wish to allow.

In this paper, we enforce disentanglement with a dynamic approach, which avoids ruling out disentangled programs. In our approach, individual memory accesses are monitored during execution, and if entanglement is detected, then the program is (safely) terminated. This allows for all disentangled programs to run to completion, including those that are effectful and/or non-deterministic. We make these guarantees precise by formulating soundness and completeness properties (Theorems 3.1 and 3.2). Roughly speaking, soundness (a.k.a. "no missed alarms") says that if entanglement is not detected, then the execution is disentangled; similarly, completeness (a.k.a. "no false alarms") says that if execution is disentangled, then entanglement is not detected.

It is important to note that, as a dynamic approach, entanglement detection is execution-dependent. If an execution of a program exhibits entanglement, then we detect the entanglement and terminate. But on the exact same program—even on the same input—it is possible that a different execution might *not* exhibit entanglement, in which case we allow the execution to complete. This is possible because the outcome of a race condition might determine whether or not entanglement occurs during execution. Therefore, although our approach handles entanglement safely, a shortcoming is that we cannot prevent the *possibility* of entanglement. An important problem for future work is to give good diagnostics when entanglement is detected, to facilitate debugging.

Because entanglement detection occurs dynamically and affects runtime performance, it is essential that it can be made efficient and scalable. Our approach takes inspiration from a long line of work on dynamic race detection for parallel programs [Bender et al. 2004; Cheng et al. 1998; Feng and Leiserson 1997; Fineman 2005; Mellor-Crummey 1991; Raman et al. 2010, 2012; Utterback et al. 2016; Xu et al. 2020]. While race detection remains expensive in practice (with overheads exceeding an order of magnitude for sequential runs, e.g., [Utterback et al. 2016; Xu et al. 2020]), we show that entanglement can be detected dynamically on-the-fly with close to zero overhead in practice. This is due to a number of differences between data races and entanglement; for example, unlike typical race detectors, our entanglement detector does not need to maintain an "access history" for each individual memory location. We defer a more detailed discussion of differences between the two techniques to Section 7.2.

Overview. We begin by considering a language with (nested) fork-join parallelism and mutable references, and present a dynamic semantics that checks for entanglement by monitoring accesses to references (mutable objects) only. The dynamic semantics constructs a *computation graph* of the execution that represents the parallel tasks and their dependencies in terms of *fork* and *join* edges. To detect entanglement, the semantics checks that each object accessed is allocated by a task that precedes the current task in the computation graph.

We prove soundness and completeness for the semantics (Theorems 3.1 and 3.2), thereby establishing that to detect entanglement, it suffices to track operations on mutable objects. Notably, the semantics incurs no "overhead" for immutable data, even when such data is reachable through a mutable object. Because the only operations monitored by the semantics are dereferences, we are able to prove that the work overhead of entanglement detection is proportional to the number of dereference operations.

For entanglement checks, the semantics associates each allocated object with a vertex in the computation graph. Naïvely, this would require $O(N)$ additional space for N heap objects, to store one vertex identifier per object. To reduce this space cost, we show how to group allocations by sharing a single vertex identifier amongst many objects allocated by the same task. This reduces the additional space cost from $O(N)$ down to approximately $O(\min(N, M/B))$, where M is the total size of memory and B is a chunking factor. The quantity M/B therefore represents the number of "heap chunks" used to store objects, which is typically much smaller than N in practice.

Our semantics paves the way for an implementation, albeit an inefficient one. The idea is to represent the computation graph using a well-known "series-parallel order maintenance" data structure for checking the precedence relation needed for entanglement checks. Series-parallel order maintenance, or *SP-order maintenance* for short, is well-studied in the race detection literature and many solutions can achieve efficiency and scalability [Bender et al. 2004; Cheng et al. 1998; Feng and Leiserson 1997; Fineman 2005; Mellor-Crummey 1991; Raman et al. 2010, 2012; Utterback et al. 2016; Xu et al. 2020]. In practice, however, the constant factors for precedence queries are significant: we completed such a direct implementation and measured that it can incur as much as 2x overhead, primarily due to the cost of precedence queries.

As a final step, we optimize away many of the SP-order maintanence operations by observing that typically, only a small number of mutable objects can lead to entanglement at any moment. We refer to such objects as *entanglement candidates*. Throughout execution, we explicitly track the set of candidates and only perform graph queries on these objects; all queries on others are pruned away. We prove that this optimization does not lead to an asymptotic impact on our bounds in the worst case, and show empirically that it can dramatically improve efficiency and scalability by eliminating many SP-order maintenance operations.

We present an implementation of the proposed approach for entanglement detection by extending the MPL (MaPLe) compiler for the Parallel ML language. To ensure both efficiency and safe execution, our implementation is integrated closely with the memory management system, including read and write barriers, as well as the garbage collector itself. We evaluate our techniques by considering a suite of 23 parallel benchmarks, many of them ported from state-of-the-art C/C++ benchmark suites. All of these benchmarks are naturally disentangled, which our detector confirms. Our evaluation shows low time overheads in practice, typically around 5% or less on both 1 and 72 processors. In comparison to a sequential baseline (with no parallelism or entanglement detection), execution with entanglement detection achieves between 10 and 63x speedup on 72 processors. Furthermore, the space overhead of entanglement detection is negligible in almost all cases. These experiments collectively demonstrate that entanglement detection has nearly zero cost in terms of time, space, and scalability.

In summary, the contributions of this paper include the following.

- The formal semantics of an entanglement detector that requires entanglement checks only at (mutable) dereference operations and its proof of correctness (Section 3).
- Asymptotic analysis establishing work and space bounds on our entanglement detection algorithm (Section 3.5).
- A technique for pruning a significant number of computation graph queries needed to support entanglement detection, resulting in dramatic performance improvements in practice (Section 4)
- An implementation in Parallel ML that extends the scheduler and memory management system of the MPL (MaPLe) compiler (Section 5).
- A performance evaluation, demonstrating low overhead and good scalability (Section 6).

2 DISENTANGLEMENT

Fork-join programs often exhibit a memory property called ***disentanglement*** which, roughly speaking, is the property that **concurrent tasks remain oblivious to each other's allocations**. In a disentangled program, tasks are not permitted to become ***entangled***: no task is allowed to obtain a pointer to an object that was allocated by a concurrent task.

Disentanglement occurs naturally when communication between concurrent tasks is restricted. In particular, all strict (call-by-value) purely functional programs are disentangled [Raghunathan et al. 2016]. More generally, as long as a program has no determinacy races, it is disentangled [Westrick et al. 2020]. We use the term ***determinacy race*** to refer to any interleaving[1] of atomic accesses to shared-memory which may result in non-determinism. In particular, if two concurrent tasks access the same (mutable) memory location, and at least one task modifies the location, then this constitutes a determinacy race.

Entanglement is always caused by a determinacy race [Westrick et al. 2020]. However, as we discuss in Sections 2.2 and 2.3, some determinacy races are compatible with disentanglement. In particular, when all communication utilizes only pre-allocated memory (i.e. memory allocated by common ancestors in the fork-join task tree), then disentanglement is still guaranteed. This allows determinacy races to be utilized for efficiency in a disentangled manner, which can be useful for efficiency in practice (Section 2.3).

2.1 Memory Management and GC

The lack of entanglement facilitates efficient parallel memory management strategies based on task-local heaps, where parallel tasks can locally garbage-collect their own data independently [Arora et al. 2021; Guatto et al. 2018; Raghunathan et al. 2016; Westrick et al. 2020]. The idea is to organize memory into a dynamic tree structure called the ***heap hierarchy*** which mirrors the dynamic tree of tasks.

In the heap hierarchy, each task is given its own local heap in which to allocate data. Initially, there is a single root heap assigned to the root task of the program. When a task forks, two fresh heaps are created for its children, and the task waits for the two child tasks to complete. When two siblings complete, their heaps are merged with their parent task, and the parent then resumes with a larger heap (consisting of all memory allocated by the parent before it forked, as well as all memory allocated by its completed children). In this way, all computation occurs at ***leaf tasks***, which have no children. Tasks that have at least one child are ***suspended***, and must wait to become a leaf before resuming (by waiting for their children to complete).

In the heap hierarchy, each heap stores objects allocated by the program that may contain pointers to other objects. Any pointer between two objects can be classified as either ***up***, ***down***, ***internal***, or ***cross***, depending on the relative positions of the heaps that contain the two objects in question [Westrick et al. 2020]. For example, if object x points to object y and $heap(x)$ is an ancestor of $heap(y)$, then the pointer from x to y is a *down pointer*. Disentanglement allows for pointers to point up, down, or internal (within a single heap), and disallows cross-pointers.

Disentangled GC. By disallowing cross-pointers, the liveness (reachability) of objects within each heap is only dependent upon its ancestors and descendants. This makes it possible for each heap in the hierarchy to be collected independently through a combination of two forms of collection, which proceed concurrently and in parallel with the various active leaf tasks. Suspended heaps (i.e. those associated with suspended tasks) may be collected without pausing their descendant tasks, using a concurrent non-moving mark-and-sweep algorithm. Leaf heaps—which are local

[1]For simplicity, we assume sequential consistency.

```
1  // initialize strings A[i..j] (exclusive at j)
2  fun init(A: string array, i: int, j: int) =
3    if j − i = 0 then () else
4    if j − i = 1 then A[i] := Int.toString i else
5    let m = ⌊(i + j)/2⌋ in
6    (init(A, i, m) || init(A, m, j)); ()
```

```
7  let A = Array.allocate 3 in
8  let _ = init(A, 0, 3) in
9  let (x, y) = (A[0]^A[2] || A[1]^A[2]) in
10 print (x ^ y)
```

Fig. 1. Example disentangled program.

to a processor—may be collected with a compacting Cheney-style collection, without needing to synchronize with other processors. Only a single task is paused to perform a leaf collection (namely, the task of the leaf being collected).

In the MPL implementation, GCs are supported by lightweight snapshots, remembered sets, and a write barrier [Arora et al. 2021; Westrick et al. 2020]. Like any GC implementation, the details are intricate; for the purposes of this paper, the important detail is that both correctness and efficiency of GC rely heavily on the lack of cross-pointers.

Entanglement can make the GC go wrong. Entanglement can cause incorrect and unpredictable behavior in a memory management system which assumes disentanglement. The problem is that the garbage collector might create a dangling pointer during execution by missing a cross-pointer. For example, if the collector reclaims an object that was reachable by some other task via a cross-pointer, then the cross-pointer will be left dangling. Alternatively, the collector might relocate an object (by making a copy and reclaiming the old version) to compact space and combat fragmentation. In doing so, the collector will update all pointers to point to the new location, but a cross-pointer might be left pointing to the old version. Either way, all bets are off: if the program attempts to read a dangling pointer, it could crash, or (worse) return an incorrect result.

2.2 Disentanglement Example

Figure 1 presents an example of a disentangled program, which we discuss in detail below. Our goal here is to illustrate the nuances of disentanglement, including its interaction with determinacy races, which can be tricky to reason about. Although determinacy races can cause entanglement, not all determinacy races are problematic. In Section 2.3, we present an example where races are utilized for efficiency in parallel programs in a manner that is compatible with disentanglement.

The code in Figure 1 operates on an array of strings, where each string is heap-allocated and immutable. We write $(e_1 \parallel e_2)$ to execute e_1 and e_2 in parallel, wait for both to complete, and return their results as a tuple. The operation `^` denotes string concatenation.

The example defines a function `init` (lines 2-6) which in parallel initializes an array A between two indices i and j by storing a freshly allocated string at each index. On line 8, the example calls `init` on an array of size 3, which results in contents `["0","1","2"]`. It then in parallel concatenates a few elements of the array, resulting in x =`"02"` and y =`"12"` (line 9). Finally, it concatenates x and y and prints out `"0212"` (line 10). As written, the code is free of determinacy races.

Example: disentangled. As presented in Figure 1, this code is disentangled. There are multiple ways we could go about showing this. One way is to observe that the code is determinacy-race-free, which ensures disentanglement [Westrick et al. 2020]. Another approach is to consider all of the allocations that occur in the computation, and where each allocated objects is used. The allocations of this computation include: the array A, the three strings stored in the array (at each $A[i]$), and the two strings allocated in parallel on line 9. The array A is allocated before everything else in the computation, so it is always safe to use. The strings stored in the array are allocated by the calls

```
type graph
type vertex = int // vertices labeled 0 to N−1
val neighbors: graph × vertex → vertex seq = ... // get the out−neighbors of a vertex

// do f(u,v) in parallel for every out−edge (u,v) where u ∈ S; return all verts v where f(u,v) is true
fun edgeMap(G: graph, S: vertex seq, f: vertex × vertex → bool): vertex seq =
  flatten (map (fn u ⇒ filter (fn v ⇒ f(u,v)) (neighbors(G,u))) S)

val P: vertex array = ... // "parents" array, length N, all initially −1 (no parent)

// try to visit v by atomically changing P[v] from −1 to u. Returns true if success, false if already visited
fun tryVisit(u,v): bool = Array.compareAndSwap(P,v,−1,u)

fun bfsRound(G: graph, F: vertex seq): vertex seq = edgeMap(G,F,tryVisit)
```

Fig. 2. BFS in Parallel ML. The code has intentional determinacy races, yet is disentangled.

`init(0, 1)`, `init(1, 2)`, and `init(2, 3)` which are respectively performed by three parallel tasks. These tasks perform no reads on the array and only update disjoint indices, so none of them acquires a cross-reference. Next, on line 9, the three strings within A are used, but this is safe, because all tasks from within the call to `init` on the previous line are guaranteed to complete before line 9. Similarly, line 10 is permitted to use the two strings allocated on line 9 for the same reason.

Example: racy and disentangled. It is possible to change Figure 1 so that the program has a determinacy race but is still disentangled. In particular, consider replacing line 9 with the following.

let $((x, y), \_)$ = $((A[0]$ ^ $A[2]$ || $A[1]$ ^ $A[2])$ || $A[2] := A[0]$)

This code has a race which causes the reads at $A[2]$ to return either `"0"` or `"2"`. Nevertheless, it is disentangled, because both of these strings were allocated by preceding tasks (namely, from within the call to `init`, which completes before line 9 begins).

Example: racy and entangled. It is also possible to change Figure 1 so that the program is entangled due to a determinacy race. Consider replacing line 9 with the following (where the intent is that the string `"!"` is allocated dynamically).

let $((x, y), \_)$ = $((A[0]$ ^ $A[2]$ || $A[1]$ ^ $A[2])$ || $A[2] :=$ “!”)

This introduces a third task which allocates a string `"!"` and writes it at $A[2]$, causing a determinacy race: in some executions x will be `"02"` but in others it will be `"0!"` (and similarly for y). This change makes the program entangled, because the tasks which read the contents of $A[2]$ might obtain a pointer to the string `"!"`, which is allocated by a concurrent task.

2.3 BFS Example: Disentanglement and Non-determinism

Parallel programmers sometimes intentionally employ determinacy races on data in shared memory. That is, a parallel program might interleave atomic accesses and updates in a non-deterministic manner, with different interleavings in different executions (due to differences in scheduling). Nevertheless, these determinacy races can be desirable from the programmer's perspective for improving performance.

Here, we show how disentanglement can allow for determinacy races to be utilized for improved efficiency. We consider a parallel breadth-first-search (BFS) graph traversal which uses atomic compare-and-swap operations. A simplified presentation of this algorithm is shown in Figure 2,

Variables	x, f		
Numbers	n	$\in$	$\mathbb{N}$
Memory Locations	ℓ		
Types	τ	$::=$	$\mathsf{nat} \mid \tau \times \tau \mid \tau \rightarrow \tau \mid \tau\ \mathsf{ref}$
Storable Values	s	$::=$	$n \mid \mathsf{fun}\ f\ x\ \mathsf{is}\ e \mid \langle \ell, \ell \rangle \mid \mathsf{ref}\ \ell$
Expressions	e	$::=$	$\ell \mid s \mid x \mid e\ e \mid \langle e, e \rangle \mid \mathsf{fst}\ e \mid \mathsf{snd}\ e \mid \mathsf{ref}\ e \mid !\,e \mid e := e \mid \langle e \parallel e \rangle$
Memory	μ	$\in$	*Locations* $\rightharpoonup$ *Storable Values*
Allocation Map	α	$\in$	*Locations* $\rightharpoonup$ *Vertices*
Task Tree	T	$::=$	$\mathsf{Leaf}(v) \mid \mathsf{Par}(v, T, T)$
Vertices	u, v, w		
Computation Graphs	G		
Program State	S	$::=$	$(\mu\,;\alpha\,;G\,;T\,;e) \mid \mathsf{ERROR}(\mu\,;\alpha\,;G\,;T\,;e)$

Fig. 3. Syntax

written in Parallel ML.[2] The BFS consists of a series of rounds, where each round visits some of the vertices of the graph. Vertices are visited by setting their "parent" in the array P. In particular, when an edge (u, v) is traversed, we set $P[v]$ to u. Initially, the parent of every vertex is set to -1 (meaning: not yet visited).

The function `bfsRound` implements one round of BFS. It takes as input a graph G and a "frontier" F which contains all vertices visited on the previous round. It then calls `edgeMap`, which is implemented in terms of standard parallel functions on sequences: `map`, `filter`, and `flatten`. The call to `edgeMap` performs many calls to `tryVisit`(u, v) in parallel, one for each edge (u, v) where $u \in F$. The function `tryVisit`(u, v) attempts to visit v by atomically changing the value of $P[v]$ from -1 to u. If this succeeds, `tryVisit` returns **true**; if it fails (because v has already been visited), it returns **false**. The output of the `edgeMap` is the collection of vertices v such that `tryVisit`(u, v) succeeded. In this way, the `edgeMap` simultaneously accomplishes two things: (1) it visits vertices in parallel by setting their parents, and (2) it returns the set of vertices visited on this round (i.e., the next frontier).

This code has a determinacy race: on a single round, there might be two edges (u_1, v) and (u_2, v) that both have the same target vertex v. The two corresponding calls `tryVisit`(u_1, v) and `tryVisit`(u_2, v) will race to visit v, and only one will succeed, resulting in a non-deterministic choice between u_1 and u_2 as the parent of v. This non-determinism is desirable from a performance perspective, because it enables the algorithm to quickly "deduplicate" edges without having to first write all these edges out to an intermediate data structure.

Despite the determinacy race in this code, it is nevertheless disentangled. Parallel ML ensures that the contents (in this case, of type `int`) of the array P are "unboxed", i.e., not heap-allocated. Therefore, reading from P does not return a pointer to heap-allocated data, and thus the determinacy races (between contending `compareAndSwap` operations within `tryVisit`) are safe for disentanglement.

3 ENTANGLEMENT DETECTION SEMANTICS

At a high level, the idea for entanglement detection is to only check dereferences of mutable data (i.e., **ref** cells). We present the detection algorithm by embedding it in the operational semantics of a small ML-like language with (nested) fork-join parallelism, mutable references, and a shared memory. The language maintains a dynamic task tree to manage parallelism, and explicitly constructs a computation graph which is used to check for entanglement.

The syntax of the language is shown in Figure 3, and selected rules of the dynamics are given in Figure 4 (additional details are available in the Appendix). Many aspects are standard (natural

[2]This code is inspired by the Ligra graph framework [Shun and Blelloch 2013].

Execution with Entanglement Detection $\boxed{S \longmapsto S'}$

$$\frac{\ell \notin \mathrm{dom}(\mu)}{(\mu\,;\alpha\,;G\,;\mathsf{Leaf}(v)\,;s) \longmapsto (\mu[\ell \hookrightarrow s]\,;\alpha[\ell \hookrightarrow v]\,;G\,;\mathsf{Leaf}(v)\,;\ell)}\ \textsc{Alloc}$$

$$\frac{\mu(\ell) = \langle \ell_1, \ell_2 \rangle}{(\mu\,;\alpha\,;G\,;\mathsf{Leaf}(v)\,;(\mathsf{fst}\ \ell)) \longmapsto (\mu\,;\alpha\,;G\,;\mathsf{Leaf}(v)\,;\ell_1)}\ \textsc{Fst}$$

$$\frac{\mu(\ell) = \mathsf{ref}\ \ell' \qquad \alpha(\ell') \preccurlyeq_G v}{(\mu\,;\alpha\,;G\,;\mathsf{Leaf}(v)\,;(!\,\ell)) \longmapsto (\mu\,;\alpha\,;G\,;\mathsf{Leaf}(v)\,;\ell')}\ \textsc{Bang-Pass}$$

$$\frac{\mu(\ell) = \mathsf{ref}\ \ell' \qquad \alpha(\ell') \not\preccurlyeq_G v}{(\mu\,;\alpha\,;G\,;\mathsf{Leaf}(v)\,;(!\,\ell)) \longmapsto \mathsf{ERROR}(\mu\,;\alpha\,;G\,;\mathsf{Leaf}(v)\,;\ell')}\ \textsc{Bang-Detect}$$

$$\frac{}{(\mu[\ell_1 \hookrightarrow (\mathsf{ref}\ \_)]\,;\alpha\,;G\,;\mathsf{Leaf}(v)\,;(\ell_1 := \ell_2)) \longmapsto (\mu[\ell_1 \hookrightarrow (\mathsf{ref}\ \ell_2)]\,;\alpha\,;G\,;\mathsf{Leaf}(v)\,;\ell_2)}\ \textsc{Upd}$$

$$\frac{v, w \notin \mathit{vertices}(G) \qquad G' = \mathit{fork}(G, u, v, w)}{(\mu\,;\alpha\,;G\,;\mathsf{Leaf}(u)\,;\langle e_1 \parallel e_2 \rangle) \longmapsto (\mu\,;\alpha\,;G'\,;\mathsf{Par}(u, \mathsf{Leaf}(v), \mathsf{Leaf}(w))\,;\langle e_1 \parallel e_2 \rangle)}\ \textsc{Fork}$$

$$\frac{w \notin \mathit{vertices}(G) \qquad G' = \mathit{join}(G, u, v, w)}{(\mu\,;\alpha\,;G\,;\mathsf{Par}(\_, \mathsf{Leaf}(u), \mathsf{Leaf}(v))\,;\langle \ell_1 \parallel \ell_2 \rangle) \longmapsto (\mu\,;\alpha\,;G'\,;\mathsf{Leaf}(w)\,;\langle \ell_1, \ell_2 \rangle)}\ \textsc{Join}$$

$$\frac{(\mu\,;\alpha\,;G\,;T_1\,;e_1) \longmapsto (\mu'\,;\alpha'\,;G'\,;T_1'\,;e_1')}{(\mu\,;\alpha\,;G\,;\mathsf{Par}(v, T_1, T_2)\,;\langle e_1 \parallel e_2 \rangle) \longmapsto (\mu'\,;\alpha'\,;G'\,;\mathsf{Par}(v, T_1', T_2)\,;\langle e_1' \parallel e_2 \rangle)}\ \textsc{ParL}$$

$$\frac{(\mu\,;\alpha\,;G\,;T_2\,;e_2) \longmapsto (\mu'\,;\alpha'\,;G'\,;T_2'\,;e_2')}{(\mu\,;\alpha\,;G\,;\mathsf{Par}(v, T_1, T_2)\,;\langle e_1 \parallel e_2 \rangle) \longmapsto (\mu'\,;\alpha'\,;G'\,;\mathsf{Par}(v, T_1, T_2')\,;\langle e_1 \parallel e_2' \rangle)}\ \textsc{ParR}$$

Fig. 4. Execution with entanglement detection (selected rules). The entanglement check is highlighted.

numbers, tuples and projections, recursive functions, mutable references). To discuss allocations, the language distinguishes between memory locations ℓ and storable values s. Storable values can be either immutable (numbers, functions, tuples) or mutable (references).

The operational semantics is a small-step semantics of the form $S \longmapsto S'$ where a single program state consists of five components: a memory μ, an allocation map α, a computation graph G, a task tree T, and an expression e. There is also an explicit error state, written $\mathsf{ERROR}(\mu\,;\alpha\,;G\,;T\,;e)$, which indicates when entanglement has been detected (see Section 3.2). Computation graphs and task trees are discussed in Section 3.1. The allocation map α is used for detection, and is discussed in Section 3.2. The memory μ is used to map memory locations to their contents. An explicit allocation step (rule Alloc) extends the memory μ, producing $\mu[\ell \hookrightarrow s]$ where ℓ is a fresh location and s is the contents of that location. In this way, storable values always "take one more step". Memory locations are the only irreducible term of the language.

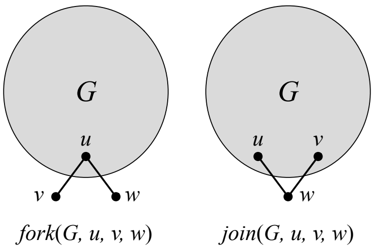

Fig. 5. Functions *fork* and *join* on computation graphs.

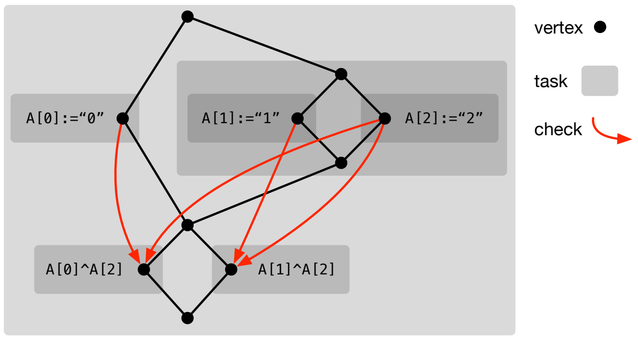

Fig. 6. Example dag and entanglement checks for the disentangled program in Figure 1.

3.1 Parallelism, Task Trees, and Computation Graphs (Dags)

The language supports nested fork-join parallelism based on ***tasks*** organized in a dynamic ***task tree***. Initially, there is a single "root" task. At any moment, any leaf (a task with no children) may ***fork***, which creates two child tasks to run in parallel. While the children are executing, the parent task is suspended. As soon as both children have completed, they ***join*** with the parent, which deletes the children from the tree and resumes execution of the parent task as a leaf. Note therefore that all computation is performed at leaf tasks. We say that two tasks are ***concurrent*** if neither is an ancestor of the other. That is, in the task tree, concurrent tasks could be siblings, cousins, etc.

In the semantics, fork-join parallelism is expressed via the parallel tuple $\langle e_1 \parallel e_2 \rangle$, which runs e_1 and e_2 in parallel, waits for both to complete, and finally evaluates to a tuple of their results. To identify which parallel pairs are currently being evaluated, the semantics maintains a task tree T, which can either be a leaf of the form $\mathsf{Leaf}(v)$, or an internal "par" node denoted $\mathsf{Par}(v, T_1, T_2)$. The elements v stored in the task tree are vertices for the computation graph, described below.

In rule Fork, a parallel tuple begins execution by transitioning from a leaf in the task tree to a par-node, with two leaves as children. Then, steps may occur either on the left or the right non-deterministically via rules ParL and ParR. Eventually, when both child tasks have completed, rule Join transitions back to a leaf in the task tree (resuming the execution of the parent task) and converts the parallel tuple to a standard tuple.

Computation Graphs (Dags). Throughout execution, the semantics maintains a graph which summarizes the execution and is used to check for entanglement. A ***computation graph*** or ***dag*** G is a directed, acyclic graph where vertices represent sequential pieces of computation and edges are ordering constraints.

Dags are extended at each fork and join using a "current vertex" stored in the task tree. In rule Fork where a leaf task currently has vertex u, the function $fork(G, u, v, w)$ extends graph G with two new vertices v and w for the child tasks, and two edges (u, v) and (u, w), indicating that the children began executing after vertex u. Symmetrically, in rule Join where two leaf siblings have currently have vertices u and v, the function $join(G, u, v, w)$ extends graph G with one new vertex w and edges (u, w) and (v, w), indicating that the continuation (at w) began executing after the children completed. This maintenance of graphs is illustrated in Figure 5.

In a dag G, we say that vertex u ***precedes*** vertex v, denoted $u \preccurlyeq_G v$, if there exists a path in the dag from u to v. Note that $\preccurlyeq$ is a partial order.

3.2 Entanglement Detection

Entanglement occurs when a task acquires a memory location that was allocated by a concurrent task. To detect entanglement, the semantics tags each memory location ℓ with a vertex $\alpha(\ell)$, indicating where in the computation the location was allocated. This occurs in rule Alloc, where both the memory μ and the map α are extended with the new location (guaranteeing $\text{dom}(\mu) = \text{dom}(\alpha)$). When dereferencing a mutable reference, we check for entanglement by inspecting the *result* of the read. Specifically, when a read at location ℓ returns some other location ℓ', we compare $\alpha(\ell')$ against the current vertex v. If $\alpha(\ell') \preccurlyeq_G v$, then this access is safe for disentanglement and the execution may proceed with rule Bang-Pass. However, if $\alpha(\ell') \not\preccurlyeq_G v$, then we have detected entanglement (rule Bang-Detect).

In short, entanglement is detected whenever rule Bang-Detect is used during execution, which results in a stuck program state: any state of the form $\text{ERROR}(\mu\,;\alpha\,;G\,;T\,;e)$ cannot step. In this way, the semantics terminates an execution as soon as entanglement is detected. All other steps allow the computation to proceed as normal.

Note that immutable reads are never checked. For example, in rule Fst, we could have included $\alpha(\ell_1) \preccurlyeq_G v$ as one of the premises, but it is intentionally left out. This is because, as we will discuss more carefully in Section 3.4, reads of immutable data are always safe for disentanglement.

3.3 Example Revisited

The dag in Figure 6 summarizes the execution of the example program from Figure 1. Each black circle is a vertex, and the solid black edges (implicitly pointing down) are execution dependencies between vertices. The tasks of the computation are illustrated as shaded gray boxes, such that at any moment throughout execution, the nesting of the shaded gray boxes represents the task tree. We draw red, curved arrows to summarize where entanglement checks occur. A red curved arrow from u to v indicates that a mutable dereference was performed at v, returning some object allocated at u. The intent here is that arrays operate analogously to mutable references: indexing into an array checks the resulting object for entanglement in the same manner as rules Bang-Pass and Bang-Detect of the semantics. For example, reading $A[0]$ and $A[2]$ (Figure 1, line 9) discovers the strings `"0"` and `"2"` which are checked to ensure both were allocated previously in the computation (i.e. not by a concurrent task). The example program is disentangled, and indeed in Figure 6 we see that entanglement is never detected, because for every red curved edge from u to v, there is a path $u \preccurlyeq v$ in the dag.

3.4 Soundness and Completeness

In our approach, entanglement is considered to have been detected whenever rule Bang-Detect is used in an execution. In this setting, soundness can be stated as a preservation property for disentanglement, i.e., that steps taken without detecting entanglement preserve disentanglement. Similarly, completeness is the property that, if disentanglement is preserved by a step, then entanglement is not detected. In other words, soundness is "no missed alarms", and completeness is "no false alarms".

The disentanglement invariant, written S de, is defined in Figure 7. It consists of two components: root disentanglement and memory disentanglement.

The ***root disentanglement*** judgement, written $\alpha\,;G\,;T\,;e$ rootsde, establishes that each task only currently uses locations allocated at or before its associated vertex. We state this formally in terms of $\text{locs}(e)$, the set of locations mentioned directly by an expression, defined in the natural way: for example, $\text{locs}(e_1\ e_2) = \text{locs}(e_1) \cup \text{locs}(e_2)$ and $\text{locs}(\ell) = \{\ell\}$. The judgement $\alpha\,;G\,;T\,;e$ rootsde is then established inductively on both the structure of e and the task tree T, where the task tree is

Disentanglement $\boxed{S \text{ de}}$

$$\frac{\mu ; \alpha ; G \text{ memde} \qquad \alpha ; G ; T ; e \text{ rootsde}}{(\mu ; \alpha ; G ; T ; e) \text{ de}} \qquad \frac{\mu ; \alpha ; G \text{ memde} \qquad \alpha ; G ; T ; e \text{ rootsde}}{\text{ERROR}(\mu ; \alpha' ; G ; T ; e) \text{ de}}$$

Memory Disentanglement $\boxed{\mu ; \alpha ; G \text{ memde}}$

$$\frac{\forall \ell \in \text{imm}(\mu).\ \forall \ell' \in \text{locs}(\mu(\ell)).\ \alpha(\ell') \preccurlyeq_G \alpha(\ell)}{\mu ; \alpha ; G \text{ memde}} \qquad \text{imm}(\mu) \triangleq \{\ell \in \text{dom}(\mu) \mid \mu(\ell) \neq \text{ref}\ \_\}$$

Root Disentanglement $\boxed{\alpha ; G ; T ; e \text{ rootsde}}$

$$\frac{\forall \ell \in \text{locs}(e).\ \alpha(\ell) \preccurlyeq_G v}{\alpha ; G ; \text{Leaf}(v) ; e \text{ rootsde}} \qquad \frac{\alpha ; G ; T_1 ; e_1 \text{ rootsde} \qquad \alpha ; G ; T_2 ; e_2 \text{ rootsde}}{\alpha ; G ; \text{Par}(v, T_1, T_2) ; \langle e_1 \parallel e_2 \rangle \text{ rootsde}}$$

$$\left.\frac{\alpha ; G ; \text{Par}(v, T_1, T_2) ; e \text{ rootsde}}{\alpha ; G ; \text{Par}(v, T_1, T_2) ; (\text{fst } e) \text{ rootsde}}\right\} \text{...}\textit{similarly for } (\text{snd } e),\ (\text{ref } e),\ \textit{and } (!\, e)$$

$$\left.\begin{array}{c}\dfrac{\begin{array}{l}\neg(e_1 \text{ loc}) \\ \alpha ; G ; \text{Par}(v, T_1, T_2) ; e_1 \text{ rootsde} \qquad \forall \ell \in \text{locs}(e_2).\ \alpha(\ell) \preccurlyeq_G v\end{array}}{\alpha ; G ; \text{Par}(v, T_1, T_2) ; (e_1\ e_2) \text{ rootsde}} \\[2ex] \dfrac{\alpha(\ell_1) \preccurlyeq_G v \qquad \alpha ; G ; \text{Par}(v, T_1, T_2) ; e_2 \text{ rootsde}}{\alpha ; G ; \text{Par}(v, T_1, T_2) ; (\ell_1\ e_2) \text{ rootsde}}\end{array}\right\} \text{...}\textit{similarly for } \langle e_1, e_2 \rangle \textit{ and } (e_1 := e_2)$$

Fig. 7. Single-step disentanglement invariant, consisting of memory property for all immutable locations, and disentanglement property for all program "roots".

used to delimit each task. Observe for example that at each leaf task, we have $\forall \ell \in \text{locs}(e).\ \alpha(\ell) \preccurlyeq_G v$, i.e. that every location ℓ which is mentioned by the expression was allocated before the current vertex v in the computation.

The ***memory disentanglement*** judgement, written $\mu ; \alpha ; G$ memde, establishes that each immutable location of the memory only points "backwards" in the computation. For example, if ℓ stores the tuple $\langle \ell_1, \ell_2 \rangle$ then $\alpha(\ell_1) \preccurlyeq_G \alpha(\ell)$ and similarly for ℓ_2. Thus, if it is safe to access ℓ then it is similarly safe to access the contents of ℓ. This formalizes our intuition that immutable data is always safe for disentanglement. In contrast, if ℓ stores a mutable reference ref ℓ', then ℓ' might have been written there by a concurrent task, hence why rule Bang-Pass and rule Bang-Detect need to check this explicitly.

Note that S de is defined for both regular program states as well as ERROR states. This makes it possible to identify any state as disentangled, regardless of whether or not the dynamic semantics claims to have detected entanglement, which is key to the statement of the completeness theorem, below.

To state soundness and completeness, we identify "no-error" program states, which are those in which the dynamic semantics has *not* detected entanglement.

$$\frac{}{(\mu ; \alpha ; G ; T ; e) \text{ noerror}}$$

The soundness and completeness theorems (Theorems 3.1 and 3.2) are then given in terms of single steps. Roughly speaking, soundness says that if entanglement is not detected, then the step is disentangled; similarly, completeness says that if the step is disentangled, then entanglement is not detected. Proofs of both theorems are available in the Appendix.

Theorem 3.1 (Soundness). *If S de and $S \longmapsto S'$ and S' noerror, then S' de.*

Theorem 3.2 (Completeness). *If S de and $S \longmapsto S'$ and S' de, then S' noerror.*

The proof of the completeness result (Theorem 3.2) is straightforward; the interesting case is rule Bang-Detect, where the theorem follows directly from a contradiction between disentanglement and the premise $\alpha(\ell') \not\preccurlyeq_G v$. The proof of the soundness result (Theorem 3.1) is more involved, as it amounts to showing that all uses of immutable data are always safe for disentanglement.

An interesting case in the soundness result is rule Alloc, which establishes the connection between the *memory disentanglement* and *root disentanglement* properties discussed above. When allocating a new (immutable) memory location, in order to satisfy the memory disentanglement property, we have to establish that the contents of the new location only point backwards, to other previously allocated locations. Because each of these other locations is a "root" of the current expression, we obtain this property from the root disentanglement invariant. Another interesting case in the proof is an immutable read. For example, in rule Fst, we have to re-establish the root disentanglement property for the result of the read. By appealing to memory disentanglement, which ensures that the contents of the immutable location only point "backwards" in the computation, we are able to do so.

3.5 Cost Analysis

We bound the work and space of our entanglement detection algorithm. Theorems 3.3 and 3.4 follow directly from the observation that entanglement detection only introduces three sources of overhead: maintaining the computation graph, mapping locations to vertices in the graph, and performing graph queries at dereference operations. The work of maintaining the computation graph can be charged to the overall work of the computation, as each step makes at most a constant number of additions to the computation graph. Mapping locations to vertices in the graph can be performed by storing vertex identifiers in memory along with the contents of each location. The remaining costs are isolated to details of how the computation graph is maintained and queried, so for the sake of brevity here, we leave these costs abstract. At a high level, the idea is to use SP-order maintenance, a well-studied problem with many solutions available "off-the-shelf" with low overhead in practice. More details about graph maintenance and queries are provided in Section 5.

Theorem 3.3 (Work). *For a program with work (total number of steps) W, execution with entanglement detection requires $O(W + D \cdot W_q)$ work in total, where D is the number of dereference operations, and W_q is the work required for a graph query.*

Theorem 3.4 (Space). *At any point during execution, entanglement detection requires $O(N + S_g)$ additional space, where N is the current number of heap objects, and S_g is the current space required to maintain the computation graph. A similar bound holds for live (reachable) memory.*

Heap Chunks. A common implementation strategy is to represent heaps as lists of "chunks", where each chunk is a fairly large region of contiguous memory (e.g. one or more pages). When heap chunks are task-local, we can significantly reduce the number of vertex labels stored for entanglement detection. In particular, our implementation (Section 5) guarantees that all objects within a chunk were allocated by the same task (or one of that task's completed subtasks). We can therefore assign one vertex identifier per chunk. For N heap objects using a total of M space, this reduces the amount of additional space needed from N (one vertex identifier per heap object) down to approximately $\min(N, M/B)$ (one vertex identifier per chunk). Because typical memory objects are small, and therefore $N \approx M$, this is a significant improvement. The following theorem formalizes the bound.

Theorem 3.5 (Chunked Space). *Using task-local heap chunks, at any point during execution, entanglement detection requires $O(\min(N, M/B) + T + S_g)$ additional space, where M is the current total heap size, N is the number of heap objects, B is the minimum size of a heap chunk, T is the current number of active tasks, and S_g is the current space required to maintain the computation graph.*

Proof. It suffices to bound the number of heap chunks. Consider a task t and let M_t be the size of t's local heap, split across k_t heap chunks, with N_t heap objects in those chunks. We assume bump-allocation within each chunk, which ensures that the amount of memory allocated in any two consecutive heap chunks is at least B. Hence, we have $k_t \leq 1 + 2M_t/B$. Also, because large objects are given their own chunks, we have $k_t \leq N_t$. Putting these two upper bounds together and summing over all active tasks yields a bound of $O(\min(N, M/B) + T)$ heap chunks. □

4 ENTANGLEMENT CANDIDATES

The entanglement detection semantics in Section 3.2 describes how to check whether an execution is entangled: for every dereference of a **ref** cell, perform a query on the computation graph (Figure 4, rules Bang-Pass and Bang-Detect). Here we show that a significant number of these queries can be pruned away dynamically, resulting in significant performance improvement in practice.

The high-level idea is to annotate each **ref** cell with a bit that indicates whether or not the **ref** requires a graph query when it is dereferenced, to check for entanglement. Any **ref** that is marked is called an ***entanglement candidate*** (or simply *candidate* for short). Throughout execution, **ref** cells are dynamically marked and unmarked; that is, a **ref** might at various points throughout execution be marked as a candidate and later unmarked when it returns to a safe state. On each dereference $!x$, we first check if x is marked as a candidate. If so, then we do a graph query to check for entanglement, consistent with the semantics. But if x is not marked, then we skip the graph query.

Below, we describe how candidates are marked and unmarked throughout execution (Section 4.1) and analyze the cost of this algorithm (Section 4.2). We first focus on **ref** objects, and then generalize our techniques to handle arrays (Section 4.3). In Section 4.5, we connect the idea of candidates with the semantics of Section 3 and argue that this technique is a valid optimization (i.e., it does not affect correctness of detection).

4.1 Marking and Unmarking Candidates

Objects are born safe. When a task first allocates a **ref**, at that moment, the **ref** can only contain data that the task already had access to. Therefore, each **ref** begins its life unmarked, indicating that the ref is safe (i.e., not a candidate). As long as a **ref** is never updated, it remains safe, similar to immutable data.

Marking candidates at updates. Intuitively, when a **ref** is updated, it might become a candidate and need to be marked. Here we leverage an observation about entanglement. Consider a reference x which currently contains a pointer to an object y, and suppose that a task becomes entangled by performing the dereference $!x$ and acquiring a pointer to y. At this moment, the pointer from x to y in memory must be a *down-pointer*, i.e., *heap*(x) must be an ancestor of *heap*(y) in the heap hierarchy (see Section 2.1). **In other words, in order to acquire a cross-pointer, a task must read a down-pointer.** Therefore, any **ref** which contains a down-pointer must be marked as a candidate. We specifically mark candidates at updates: when a task performs an update $x := y$, if this creates a down-pointer from x to y, then the task marks x as a candidate. Note that if the pointer from x to y is not a down-pointer, then the **ref** is not marked as a candidate.

Unmarking candidates in leaf heaps. Any **ref** which does not contain a down-pointer is no longer a candidate and should be unmarked. To unmark a candidate x, we wait until $heap(x)$ is a leaf heap (which occurs naturally due to tasks joining with their parents). At this point, because the heap has no descendants, we know that any pointer from x is no longer a down-pointer, and thus x is safe. Therefore, we unmark all candidates within a heap whenever a heap becomes a leaf. Specifically, whenever two completed tasks join, we unmark all candidates in their parent heap.

Candidate sets. To facilitate unmarking candidates in bulk when a heap becomes a leaf, we give each heap a *candidate set* which is the set of objects within that heap that are currently marked as a candidate. When a candidate is marked, it is added to the corresponding set. To unmark candidates of a heap, we iterate through the candidate set and unmark each object individually. The candidate set is then cleared.

4.2 Cost Analysis of Tracking Candidates

Tracking candidates does not impact the asymptotic costs of execution and has low overhead. The space overhead of candidate sets is $O(1)$ per candidate, because each candidate is only stored in one candidate set. The algorithm takes constant work on each update, to mark a candidate and add it to a candidate set. At joins, it traverses the candidate set of a single heap to clear candidates. This incurs constant work per candidate, and can be charged to the cost of the update that marked the candidate. When an object is dereferenced, we incur constant overhead to first check if the object is a candidate.

Our algorithm for tracking candidates is conservative because the program may also delete a down-pointer (causing an object which is marked as a candidate to no longer have any down-pointers), but the algorithm does not attempt to track this. It would be prohibitively expensive to attempt to track individual down-pointer deletions, especially with candidate arrays (Section 4.3), which may have a large number of down-pointers at any moment. Our algorithm avoids this cost by unmarking candidates “lazily” at joins.

4.3 Candidate Arrays

Generalizing the above algorithm for mutable **array** objects is straightforward. Similar to **ref** cells, we give each array a single bit. If an update $a[i] := x$ creates a down-pointer from a to x, then we mark a as a candidate. Later, when $heap(a)$ becomes a leaf, we unmark a.

In this way, we (conservatively) only track whether or not the whole array is a candidate, rather than attempting to precisely track every index of the array separately, which would be costly in practice. For example, consider an array of pointers and suppose several tasks are adding and deleting down-pointers to it, in parallel. Determining when a particular delete removes the last down pointer from the array would be prohibitively expensive (it would require counting the number of down-pointers and updating this every time the array is mutated). Instead, our algorithm unmarks the array (as a candidate) after all descendant tasks complete. In this way, the algorithm only incurs constant overhead for the entire array.

4.4 Asymptotically Fewer Graph Queries

By tracking candidates, we asymptotically reduce the number of graph queries for many parallel algorithms and primitives. In particular, parallel operations on mutable arrays such as map, filter, and scan (a.k.a., prefix sums) typically reduce from $O(n)$ queries down to $O(1)$.

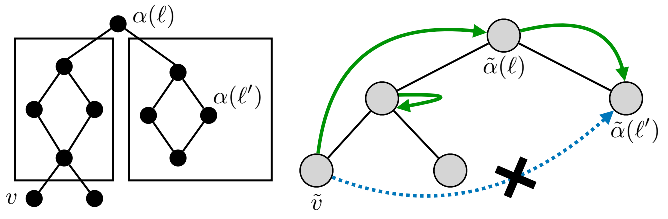

Fig. 8. A partial dag on the left and its corresponding task tree on the right. Each node of the tree corresponds to contracted sub-dags, shown delimited by boxes. In the tree, disentangled pointers may point up, down, or internally to a node. An example entangled pointer is shown in dotted blue.

4.5 Candidates in the Detection Semantics

We now describe how the notion of candidate can be defined in terms of the semantics of Section 3, allowing us to argue that tracking objects with down-pointers suffices for entanglement detection. We formally define a location ℓ as a candidate as follows. The definition says that location ℓ is a candidate if there exists a leaf task that is allowed to access ℓ but who would become entangled if it would dereference ℓ. This is precisely the condition in which entanglement is detected by the semantics.

Definition 1. In a program state $(\mu\,;\alpha\,;G\,;T\,;e)$, consider a location ℓ where $\mu(\ell) = \text{ref }\ell'$. We say that ℓ is an ***entanglement candidate*** if there exists some $\mathsf{Leaf}(v)$ in the task tree T such that $\alpha(\ell) \preccurlyeq_G v$ and $\alpha(\ell') \not\preccurlyeq_G v$.

For example, consider the array A in the example of Figure 6, whose code was shown in Figure 1. Even though the array is mutable, the reads during `A[0]^A[2]` (and during `A[1]^A[2]`) cannot cause entanglement because the contents of the array (`A[0]`, `A[1]`, and `A[2]`) were all allocated *before* ($\preccurlyeq_G$) all leaf tasks. That is, while it is being read from (Figure 1, line 9), the array A is not a candidate and we can elide the graph queries. Note that A was a candidate earlier in the computation—specifically, while it was being initialized during the call to `init` (Figure 1, line 8). It just so happens that the array was never read while it was a candidate. As soon as the tasks within the call to `init` join, the array A is no longer a candidate.

Relating task trees and computation graphs. According to Definition 1, whether or not an object is a candidate depends on the current state of the computation graph G and how it relates to the current task tree, T. We can relate them as follows. Given a partial computation dag G, we can derive the task tree by contracting the dag: label each vertex of the dag so that (i) a fork and its corresponding join get the same label, and (ii) all vertices between them also get that label. If we contract all the vertices with the same label, we get a tree that is isomorphic to the task tree (upto labels); the labels map vertices of the dag to nodes in the task tree.

We use $\tilde{u}$ to refer to the label of vertex u, and define a partial order $\leq$ on the nodes of the task tree: $\tilde{u} \leq \tilde{v}$ if $\tilde{u}$ is an ancestor of $\tilde{v}$. The direction of pointers between objects (up, down, internal, and cross, as described in Section 2.1) can then be expressed in terms of the labels assigned to vertices. In particular, for a location ℓ, let $\alpha(\ell)$ be its allocation vertex in the dag and let $\tilde{\alpha}(\ell)$ be its label in the task tree. Based on their positions in the tree, we can classify a pointer from ℓ to ℓ' as follows: (i) *up pointer* if $\tilde{\alpha}(\ell) > \tilde{\alpha}(\ell')$, (ii) *internal pointer* if $\tilde{\alpha}(\ell) = \tilde{\alpha}(\ell')$, (iii) *down pointer* if $\tilde{\alpha}(\ell) < \tilde{\alpha}(\ell')$, and (iv) *cross pointer* otherwise: when $\tilde{\alpha}(\ell) \not\leq \tilde{\alpha}(\ell')$ and $\tilde{\alpha}(\ell') \not\leq \tilde{\alpha}(\ell)$.

For example, consider the dag and its tree in Figure 8. The dag has two fork-join pairs that correspond to subcomputations that have finished, as shown by the boxes; so every vertex within a box gets the same label. All other vertices are not between a fork-join pair, so they get different labels. After contracting the boxes, we get the tree on the right (with gray nodes). The figure also shows an up-pointer, an internal pointer, and a down-pointer, shown with solid green arrows. An example cross pointer is illustrated as a dotted blue arrow.

Candidates have down-pointers. The following statement formalizes the intuition that candidate objects have down-pointers, thereby establishing that marking candidates when down-pointers are created is sufficient for detecting entanglement (Section 4.1). We give proof details in the Appendix.

$$\ell \text{ is an entanglement candidate} \Leftrightarrow \exists\, \ell' : \ell' \in \mathsf{locs}(\mu(\ell)) \wedge \tilde{\alpha}(\ell) < \tilde{\alpha}(\ell')$$

5 IMPLEMENTATION

We implemented our entanglement detection algorithm by extending the MPL (MaPLe) compiler for Parallel ML [Arora et al. 2021; Westrick et al. 2020]. We call our implementation MPL$^{\Delta}$ ("maple-delta"). Our changes have been incorporated into the main MPL repository (https://github.com/MPLLang/mpl) under version `v0.3`. In this section, we present an overview of several important aspects of the implementation, including integration of entanglement detection with the scheduler and memory management system.

5.1 Scheduling and SP-order Maintenance

MPL uses a standard work-stealing scheduler [Acar et al. 2013; Arora et al. 2001; Blumofe and Leiserson 1998] which (at a high level) maps user-level threads onto OS threads. We modified MPL's user-threads to additionally store their current vertex identifier. At forks and joins, the scheduler reassigns the current vertex identifier of the current thread to match that specified by our algorithm. To implement vertex identifiers and graph queries, we use DePa [Westrick et al. 2022d], an "off-the-shelf" SP-order maintenance algorithm.

5.2 Read and Write Barriers for Detection

Marking entanglement candidates. We modified MPL's existing GC write barrier to additionally mark the mutated object as an entanglement candidate if the write creates a down-pointer. Because MPL already has special handling for down-pointers [Westrick et al. 2020], this modification was straightfoward: in addition to MPL's existing down-pointer management, we simply mark a bit in the object's GC header to indicate that it is an entanglement candidate.

Inserting read barriers. As MPL does not already have a read barrier for GC, we modified the compiler to insert read barriers for reads (dereferences) of mutable objects. In Parallel ML, ref-cells and arrays are the only mutable objects, meaning that the only operations needing a read barrier are the operations which retrieve elements from refs and arrays, including standard dereferences as well as atomic compare-and-swaps:

```
val !: α ref → α                  val refCompareAndSwap: α ref × α × α → α
val sub: α array × int → α        val arrayCompareAndSwap: α array × int × α × α → α
```

In the compiler front-end, we begin by conservatively marking all of these operations as needing a read barrier. However, not all actually result in a read barrier in the generated code. In the compiler backend, MPL eventually chooses concrete, bit-level representations for all data. In this step, we only add read barriers if MPL chooses to represent the contents of the ref (or array) with a pointer.

For example, consider an object x of type `int` **ref** and corresponding dereference `!`x. MPL never chooses to indirect the integer through a pointer, so no read barrier is inserted. But with types such as `(int × int)` **ref**, MPL may choose to either (a) inline the integers into the ref, or (b) allocate the tuple separately and represent the contents of the cell with a pointer. A read barrier is inserted only in the latter case, where MPL chooses to allocate the contents separately.

Read barrier fast and slow path. The read barrier has two behaviors: a fast path for objects that are not entanglement candidates, and a slow path otherwise. The fast path is implemented by codegen in the compiler. For each dereference `!`x, we generate code which first optimistically performs the read, and then checks the header of x to determine if x is a candidate. If x is not a candidate, then the fast path is satisfied, and execution continues as normal (i.e., the graph query is elided). However, if x is marked as a candidate, then we fall back on the slow path to perform a graph query and check for entanglement. The slow path is implemented as a function call into the run-time system (written in C and linked with the generated code).

5.3 Memory Management

Vertex identifiers in heap chunks. MPL represents each heap as a linked list of *heap chunks*, where each chunk is one or more contiguous pages of virtual memory. Two heaps can therefore be merged in constant time by linking two chunk-lists together. To allocate memory, a task bumps an "allocation pointer" within its current chunk, or allocates a new chunk whenever the current chunk becomes full. Large objects—specifically those more than half a page in size—are given their own dedicated chunk.

At the front of each chunk is a *chunk descriptor* containing various GC metadata. We modified the chunk descriptor to additionally store a vertex identifier, which is shared across all objects within the chunk. When a new chunk is allocated, it is assigned the vertex identifier of the task that allocated the chunk. Chunk boundaries are aligned, allowing fast access to chunk descriptors by zeroing-out the low-order bits of an object's address. This makes it possible (in constant time) to look up the vertex of the task which allocated an object. The read barrier therefore can check for entanglement by inspecting the vertex stored in the chunk descriptor of the object in question.

Reassignment of vertices during GC. In MPL, the GC design incorporates a local copying collector which compacts the live objects of each task into a new set of heap chunks. These new chunks must be assigned an appropriate vertex identifier. Here, we take advantage of the fact that after two tasks join, their corresponding graph vertices are indistinguishable with respect to entanglement detection. That is, using the definitions of Section 4, for any two vertices u and v such that $\tilde{u} = \tilde{v}$ and any leaf vertex w, we have $u \preccurlyeq_G w$ if and only if $v \preccurlyeq_G w$. When picking a vertex identifier for a new chunk during GC, it is therefore safe to use any vertex identifier of any chunk in the heap being compacted.

5.4 Chunk Pinning: Handling the Possibility of Entanglement

We now describe how to ensure that execution remains memory-safe, even at the moment when entanglement occurs. This requires care, because there is a race between garbage collection (which compacts task-local memory by copying objects) and entanglement (which allows a task to reach into another task's local memory).

To illustrate the problem, consider two concurrent tasks A and B as shown in Figure 9. Task A begins executing a dereference `!x`, where `x` is shared between the two tasks. By reading from `x`, task A acquires a pointer to an object `y` which was allocated by B. Next, suppose task B performs a garbage collection which copies `y` to a new location `y'`, forwards the down-pointer (from `x`) to point to `y'`, and finally reclaims `y`. When A proceeds with the entanglement check on `y`, it will then read from reclaimed memory, and possibly crash (or worse).

Pinning. To make the entanglement check memory-safe, we have to ensure that it is safe for a concurrent task to access the chunk descriptor of any object which is potentially entangled. Here we take advantage of a key property (discussed in Section 4): entanglement can only occur due to a read of a down-pointer. When MPL creates a down-pointer, it executes a write barrier (which

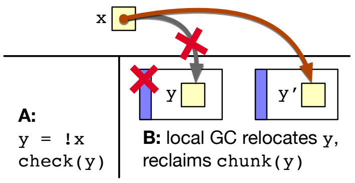

Fig. 9. Example of race between local GC and the entanglement check. Task A first acquires a pointer to y. Meanwhile, B forwards y to y' and reclaims the old memory. Task A then proceeds with the entanglement check on a dangling pointer.

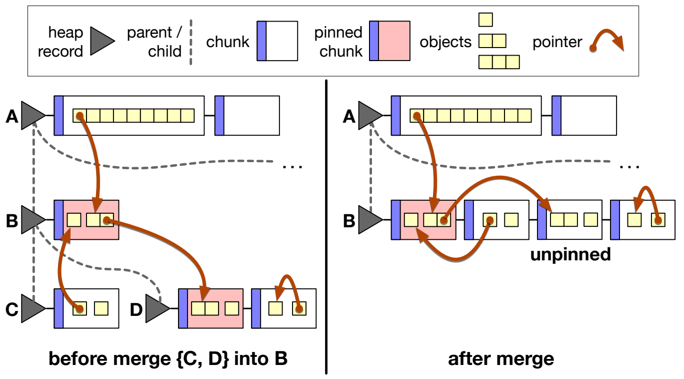

Fig. 10. Example, before and after heaps **C** and **D** merge into **B**. Afterwards, the down-pointer from **B** into **D** has become an internal pointer, and therefore the indicated chunk may be unpinned.

adds the down-pointer to a remembered set for GC). We modified the write barrier to mark the target object of each down-pointer as *pinned*, and modified the GC to not move these objects (i.e. the address of a pinned object must not change during a collection). Any chunk that contains a pinned object is called a *pinned chunk*. By preserving the addresses of pinned objects, the GC implicitly ensures that pinned chunks are not reclaimed. Therefore, it is safe at any moment for a task to access the descriptor of a pinned chunk, to inspect the vertex identifier stored there to detect entanglement.

In the example of Figure 9, the pinning technique fixes the race by ensuring that object y is pinned. The GC performed by task B therefore is required to preserve y (and the chunk containing y), allowing the entanglement check performed by A to proceed safely.

Unpinning. A pinned object may be later unpinned as soon as there are no more down-pointers which point at the object. As heaps merge upwards due to task joins, down-pointers naturally become internal, which enables unpinning. We could unpin objects at heap merges: whenever a heap is merged into its parent, if this causes all down-pointers incident upon a pinned object to become internal, then the object may be safely unpinned. However, unpinning objects at every heap merge would be inefficient, because merges are frequent. We therefore instead perform unpinning in bulk during GC.

The modifications to GC for unpinning were straightforward, because the MPL GC already has a phase which filters down-pointers that have become internal due to prior joins. In particular, down-pointers are stored in a remembered set for each heap, and at local GC, the remembered set is scanned to find down-pointers which have since become internal. During this process, we identify the set of objects which must remain pinned (due to the existence of a down-pointer).

Example. Figure 10 shows an example where pinned chunks become unpinned due to a heap merge. In the figure, there are four heaps shown on the left, labeled **A** through **D**. Each heap consists of a list of chunks, and within each chunk are various objects of different sizes allocated by the program, with pointers to other objects. Pinned chunks are shaded in light red: these chunks contain pinned objects, i.e., objects which are pointed-to by a down-pointer from an ancestor heap. On the left in the figure, there are two pinned chunks: one in heap **B**, and one in heap **D**. The right-side of the figure illustrates the heap structure after **C** and **D** are merged into their parent. As a result of the merge, a down-pointer (from **B** into **D**) becomes an internal pointer within **B**. The corresponding object therefore may be unpinned, as may its containing chunk (which no longer contains any pinned objects).

Cost of chunk pinning. Chunk pinning was the only major change required to safely handle the possibility of entanglement. We measured the difference due to pinning alone and found that overall it offers a slight improvement in time and space, with approximately −3% and −7% difference (respectively) in time and space on average across the benchmarks in our evaluation (Section 6). This slight improvement may seem surprising, given that one disadvantage of pinning is that it introduces additional fragmentation into the system (by preventing the GC from relocating pinned objects to compact memory). However, note that this fragmentation is short-lived: as the program "joins back up", objects naturally become unpinned, as illustrated in Figure 10.

The space loss due to fragmentation appears to be outweighed by other advantages of pinning. For example, the pinning strategy allows for a more space-efficient implementation of remembered sets for down-pointers. In particular, MPL previously used remembered-set entries of the form (x, i, y), indicating that $x[i]$ is a down-pointer to y; this representation allows for the GC to update $x[i]$ to point to the new version of y when y is relocated by collection. Under the pinning strategy, because the addresses of pinned objects are kept fixed (and thus down-pointers do not need to be updated by GC), we do not need to store this additional information in the remembered set. In our MPL$^\Delta$, we instead use remembered-set entries of the form $(d(x), y)$, where $d(x)$ is the depth of the object x which contains a down-pointer to y. The depth is used to determine when it is safe to unpin y. We therefore save space by shrinking the size of each individual remembered-set entry. Furthermore, if there are multiple down-pointers incident upon an object, we only need to keep one entry (the one with minimum depth). These differences can result in significant space savings in the remembered set, especially in programs with a large number of down-pointers.

6 EVALUATION

We present an experimental evaluation of our implementation on a suite of parallel benchmarks. We call our implementation MPL$^\Delta$ ("maple-delta"), as it extends the MPL compiler [Arora et al. 2021; Westrick et al. 2020] for Parallel ML with entanglement detection. Our implementation is publicly available in the main MPL repository[3] under version `v0.3`. Code used in our evaluation is available on Zenodo [Westrick et al. 2022b] and GitHub[4].

At a high level, our evaluation consists of three parts, with the following results.

(1) In Section 6.2, we compare our MPL$^\Delta$ against vanilla MPL (which has no entanglement detection) to measure the overhead of our techniques. We observe close to zero overhead (in terms of both time and space) across the board. For time specifically, we measure approximately 1% overhead on average, and a max of 7%. A majority of benchmarks (18 out of 23) incur less than 2% time overhead.
(2) In Section 6.3, we compare against the MLton [MLton nd] compiler for Sequential (Standard) ML to determine the end-to-end overheads and scalability of our MPL$^\Delta$. We observe sequential overheads typically within a factor of 2, and speedups between 10 and 63 at scale, using 72 processors. This confirms that our techniques are fully parallel and highly scalable.
(3) In Section 6.4, we consider the performance improvement of our entanglement candidates algorithm, which eliminates unnecessary graph queries. We observe that the entanglement candidates algorithm provides improvements up to a factor of 2x, bringing the number of graph queries down to 0 in multiple cases.

[3]https://github.com/MPLLang/mpl
[4]https://github.com/MPLLang/entanglement-detect/tree/icfp22-artifact

Table 1. Times (seconds), max residencies (GB), and percent differences of MPL^{Δ} relative to MPL in parenthesis. The percentage is the overhead of entanglement detection (relative to MPL).

	T_1		T_{72}		R_1		R_{72}	
	MPL	**MPL$^{\Delta}$ (Ours)**	MPL	**MPL$^{\Delta}$ (Ours)**	MPL	**MPL$^{\Delta}$ (Ours)**	MPL	**MPL$^{\Delta}$ (Ours)**
bfs-tree	19.9	20.5 (+3%)	.433	.442 (+2%)	7.8	7.8 (+0%)	68	68 (+0%)
centrality	16.2	16.0 (-1%)	.432	.426 (-1%)	6.9	6.9 (+0%)	5.9	5.9 (+0%)
dedup-strings	5.35	5.52 (+3%)	.114	.122 (+7%)	2.6	2.6 (+0%)	6.4	6.7 (+5%)
delaunay	8.72	9.18 (+5%)	.323	.335 (+4%)	.59	.58 (-2%)	1.3	1.3 (+0%)
dense-matmul	2.78	2.79 (+0%)	.048	.048 (+0%)	.064	.063 (-2%)	.31	.31 (+0%)
game-of-life	1.81	1.73 (-4%)	.067	.067 (+0%)	.061	.061 (+0%)	.35	.35 (+0%)
grep	2.09	2.05 (-2%)	.038	.038 (+0%)	1.1	1.1 (+0%)	1.4	1.4 (+0%)
low-d-decomp	8.16	8.37 (+3%)	.406	.392 (-3%)	6.9	6.9 (+0%)	14	13 (-7%)
msort-strings	2.70	2.77 (+3%)	.060	.058 (-3%)	.62	.62 (+0%)	1.7	1.7 (+0%)
nearest-nbrs	1.54	1.54 (+0%)	.046	.047 (+2%)	.26	.26 (+0%)	1.6	1.6 (+0%)
nqueens	1.60	1.61 (+1%)	.028	.029 (+4%)	.045	.045 (+0%)	.32	.32 (+0%)
palindrome	1.62	1.69 (+4%)	.031	.032 (+3%)	.061	.063 (+3%)	.36	.36 (+0%)
primes	7.42	7.56 (+2%)	.123	.120 (-2%)	.27	.27 (+0%)	.56	.56 (+0%)
quickhull	3.22	3.40 (+6%)	.103	.110 (+7%)	2.4	2.4 (+0%)	5.9	6.1 (+3%)
range-query	14.4	15.4 (+7%)	.250	.248 (-1%)	4.5	4.5 (+0%)	4.5	4.6 (+2%)
raytracer	3.50	3.30 (-6%)	.058	.058 (+0%)	.11	.11 (+0%)	.54	.55 (+2%)
reverb	1.34	1.31 (-2%)	.042	.041 (-2%)	1.5	1.5 (+0%)	2.1	2.1 (+0%)
seam-carve	16.2	16.2 (+0%)	.808	.827 (+2%)	.091	.091 (+0%)	.71	.71 (+0%)
skyline	7.31	7.57 (+4%)	.250	.251 (+0%)	.88	.88 (+0%)	25	25 (+0%)
suffix-array	5.54	5.74 (+4%)	.113	.115 (+2%)	1.2	1.2 (+0%)	1.7	1.6 (-6%)
tinykaboom	2.47	2.48 (+0%)	.042	.042 (+0%)	.0093	.011 (+18%)	.31	.31 (+0%)
tokens	1.96	1.86 (-5%)	.043	.042 (-2%)	1.1	1.1 (+0%)	1.4	1.4 (+0%)
triangle-count	4.74	4.57 (-4%)	.192	.191 (-1%)	2.5	2.5 (+0%)	14	14 (+0%)

Experimental setup. We run all of our experiments on a 72-core Dell PowerEdge R930 consisting of 4×2.4GHz Intel (18-core) E7-8867 v4 Xeon processors and 1TB of memory. To measure run times, we run each benchmark 20 times back-to-back and report the average, excluding initialization (e.g., loading the input), warmup, and teardown. To measure space usage, we measure the average of the maximum resident set size (as reported by Linux) of 20 back-to-back runs of the benchmark. We write T_P for time on P processors, and similarly R_P for the max residency on P processors. For the sequential baseline runs, we write T_s. All run times are in seconds, and all space numbers are in GB.

6.1 Benchmarks

We consider numerous benchmarks from different problem domains, including graphs, text processing, digital audio processing, image analysis and manipulation, numerical algorithms, computational geometry, and others. Many of these are ported to Parallel ML from existing state-of-the-art parallel C/C++ benchmark suites and libraries including PBBS [Anderson et al. 2022; Blelloch et al. 2012; Shun et al. 2012], ParlayLib [Blelloch et al. 2020; Westrick et al. 2022c], Ligra [Shun and Blelloch 2013], and PAM [Sun et al. 2018]. Even though these benchmarks were originally written in C/C++, all of these benchmarks happened to be naturally disentangled. An overview of the benchmarks is available in the Appendix.

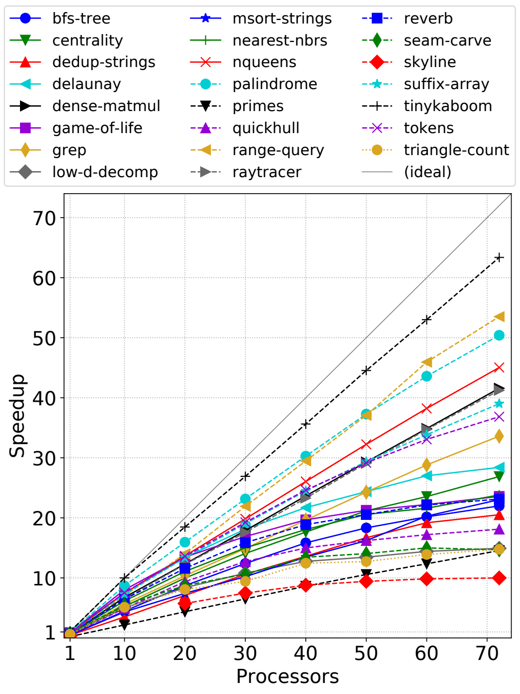

Fig. 11. Speedups of MPL$^\Delta$ in comparison to sequential baseline (which has no parallelism or entanglement detection).

Table 2. Comparison (overheads OV and speedups SU) with sequential baseline. Times are in seconds.

	Base	$P = 1$		$P = 72$	
			OV		SU
	T_s	T_1	T_1/T_s	T_{72}	T_s/T_{72}
bfs-tree	9.71	20.5	2.11	.442	22
centrality	11.5	16.0	1.39	.426	27
dedup-strings	2.50	5.52	2.21	.122	20
delaunay	9.50	9.18	0.97	.335	28
dense-matmul	1.99	2.79	1.40	.048	41
game-of-life	1.57	1.73	1.10	.067	23
grep	1.28	2.05	1.60	.038	34
low-d-decomp	5.85	8.37	1.43	.392	15
msort-strings	1.33	2.77	2.08	.058	23
nearest-nbrs	1.11	1.54	1.39	.047	24
nqueens	1.30	1.61	1.24	.029	45
palindrome	1.61	1.69	1.05	.032	50
primes	1.75	7.56	4.32	.120	15
quickhull	2.00	3.40	1.70	.110	18
range-query	13.3	15.4	1.16	.248	54
raytracer	2.38	3.30	1.39	.058	41
reverb	.955	1.31	1.37	.041	23
seam-carve	12.1	16.2	1.34	.827	15
skyline	2.51	7.57	3.02	.251	10
suffix-array	4.51	5.74	1.27	.115	39
tinykaboom	2.63	2.48	0.94	.042	63
tokens	1.53	1.86	1.22	.042	36
triangle-count	2.82	4.57	1.62	.191	15

6.2 Comparison with MPL

We compare with MPL to confirm our theoretical results, showing little overhead and excellent scalability as the number of cores increase.

Entanglement detection overheads are small. Table 1 shows results on 1 and 72 processors. Columns T_1 and T_{72} show the run-time for MPL and MPL$^\Delta$, which the additional cost of MPL$^\Delta$ relative to MPL shown as percentage for each quantity. Observe that the MPL$^\Delta$ times are usually within ±5% of MPL. 18 out of 23 benchmarks have less than 2% time overhead, and we observe a max time overhead of 7% in only two cases on 72 processors. On average across all benchmarks, the time overhead is approximately 1% on both 1 and 72 processors.

Entanglement detection scales well. On both 1 and 72 processors, we observe similar time overheads across all benchmarks. Entanglement detection therefore has no noticeable impact on scalability.

Space overheads are small. Columns R_1 and R_{72} show the space usage for MPL and MPL$^\Delta$, with the additional cost of MPL$^\Delta$ relative to MPL shown as percentage for each quantity. For both 1-core and 72-core runs, there is almost no noticable space overhead. Only one benchmark (`tinykaboom`) registers above 10% space overhead, but only for sequential runs, where the overall footprint is small: while MPL uses approximately 9 MB, our MPL$^\Delta$ uses 11 MB. Entanglement detection therefore appears to have a small constant space overhead. At scale, with memory on the order of gigabytes, this overhead is negligible.

Difference due to pinning. One of the ways MPL$^\Delta$ differs from the original MPL is that it handles down-pointers via the chunk-pinning technique, described in Section 5. We measured the difference due to pinning alone and found that overall it has little impact, with only about −3% and −7% improvement (respectively) in time and space on average across the benchmarks. The results in Table 1 therefore are with respect to MPL with pinning enabled, to isolate the impact of entanglement detection.

6.3 End-to-End Scalability

Our prior experiments have established that the space and time overheads of entanglement detection are low, both on a single processor and at scale. Here we evaluate the end-to-end scalability of the executables generated by our MPL$^\Delta$ compiler by comparing them against executables generated by the optimizing MLton compiler for Standard ML. As baseline we use the sequential versions of our benchmarks, which are derived from parallel benchmarks by replacing parallel tuples with sequential ones. This approach enables us to measure the cost of entanglement detection by keeping the underlying algorithms the same for both parallel and sequential versions.

Figure 11 shows the speedup of our benchmarks on up to 72 processors with entanglement detection. All benchmarks exhibit scalability, some very well, some less, depending on their specific characteristics. Table 2 shows more concrete measurements for sequential (baseline), uniprocessor runs ($P = 1$) and parallel runs with 72 processors ($P = 72$). The overhead—measured by the ratio of uniprocessor to sequential time—is relatively small, ranging between a factor of 0.94 (6% faster) to 4.32 (332% slower) in comparison to the sequential baseline. These overheads include all overheads of parallelism as well as entanglement detection, which are usually small as shown in Table 1. The largest overhead in comparison to MLton is on the `primes` benchmark (approximately 4x), which has been similarly observed with previous versions of MPL, and appears to be due to differences in compilation rather than overheads in the runtime system [Arora et al. 2021; Westrick et al. 2020].

The speedup—measured by the ratio of the sequential running time to the 72-core running time—shows the benefits of parallelism over the sequential. The results show that the speedups are significant and that they are usually inversely correlated with the overheads, showing that the scalability is overall quite good.

6.4 Improvement due to Entanglement Candidates

We now demonstrate that by eliminating unnecessary graph queries for entanglement detection, we significantly improve performance. Our technique for eliminating unnecessary graph queries is our candidate tracking algorithm (Section 4). To measure the performance improvement due to tracking candidates, we additionally run a version of our compiler that has candidate tracking disabled. We focus here only on the benchmarks which have a large number of graph queries (i.e., a large number of dereferences of mutable pointers to heap-allocated objects). The performance of the omitted benchmarks is not significantly affected by the candidate tracking algorithm (specifically, on average, the performance change is ±1%). That is, by tracking candidates, we improve performance only in the cases where it is needed, and do not harm performance otherwise.

The results of this comparison are presented in Table 3. In the table, the "Improvement Ratio" columns are calculated as $T_{\text{off}}/T_{\text{on}}$, where T_{off} is the time with candidate tracking disabled, and T_{on} is the time with it enabled. Improvement ratios larger than 1 indicate a performance improvement due to the candidate tracking algorithm. We provide improvement ratios for both uniprocessor runs ($P = 1$) as well as parallel runs ($P = 72$). We also collect additional data, including the number of graph queries performed (both with and without candidate tracking), and the number of times a candidate was marked (which is an upper bound on the number of candidate objects).

Table 3. Performance improvement ratio due to tracking candidates, including number of graph queries performed both with and without candidate tracking.

	Improvement Ratio		# Graph Queries		
	$P = 1$	$P = 72$	Naïve	w/ Candidates	# Candidate Marks
bfs-tree	0.99x	0.99x	16M	16M	64
dedup-strings	**1.17x**	**1.12x**	122M	0	0
delaunay	**1.44x**	**1.25x**	237M	0	0
low-d-decomp	0.99x	**1.02x**	1M	0	62
msort-strings	**1.87x**	**1.95x**	247M	0	0
nearest-nbrs	**1.60x**	**1.23x**	73M	0	0
quickhull	**1.62x**	**1.20x**	184M	0	0
skyline	**1.16x**	**1.07x**	174M	1M	1K
triangle-count	**1.02x**	0.95x	3M	0	2

We first observe that by tracking candidates, we are able to improve performance in almost all cases, often by a significant margin. The benchmark with the biggest change is `msort-strings`, with approximately 2x improvement. Other improvements range from 2% up to a factor of 1.6x. Inspecting the number of graph queries performed, we see that our approach successfully elides a significant number of unnecessary graph queries, and in all but 2 benchmarks, the number of graph queries goes down to 0. For example, under the naïve strategy (tracking disabled), the `msort-strings` benchmark performs approximately 250M graph queries, but after tracking candidates, none are performed.

This elimination of unnecessary graph queries is key to the near-zero overhead of entanglement detection. Note that although the graph queries are elided, there is still overhead for the "fast path" of the read barrier on mutable objects, which inspects the header of the dereferenced object to determine whether or not it is a candidate. However, this overhead is not significant enough to be a concern: as evidenced by the results in comparison with vanilla MPL (in Table 1), the cost of the fast path is nearly zero.

There are three interesting benchmarks here which do not see significant improvement: `bfs-tree`, `low-d-decomp`, and `triangle-count`. In the case of `low-d-decomp` and `triangle-count`, although tracking candidates successfully eliminates all graph queries, the number of queries needed in the first place is low relative to the work performed by the benchmark, so the opportunity for improvement is small (e.g. the input for `low-d-decomp` has approximately 200M edges but performs only 1M queries when the candidate tracker is disabled). In contrast, the `bfs-tree` benchmark achieves no performance improvement because no graph queries are eliminated. This is the only benchmark does not see a significant number of queries eliminated due to the candidates algorithm. The reason in this case is that the algorithm uses a shared array to associate state with each element of the input, and negotiates access to this state via a lock-free algorithm. As soon as one piece of this state is updated, the shared array becomes a candidate, and all further accesses to it require full checks.

Finally, it's worth emphasizing that the number of candidate marks (i.e., the number of times any object is marked as a candidate) is small across the board. This is an upper bound on the number of candidate objects, and therefore bounds the space overhead of tracking candidates (i.e., it bounds the size of the candidate set during execution). These results confirm our hypothesis that only a small number of objects typically pose a risk for entanglement.

6.5 Entangled Tests

As mentioned, all benchmarks used in our performance evaluation above are naturally disentangled. To validate our detector, we also developed a set of entanglement test cases, including both synthetic tests as well as variations of some of our benchmarks. Our detector successfully found entanglement in all instances. One interesting case of entanglement we developed was a variation of the `dedup-strings` benchmark. In this benchmark, we were able to introduce entanglement by replacing many task-local hash tables with a single hash table shared across all tasks; this forces tasks to compete non-deterministically for access to the shared hash table, leading to entanglement.

7 RELATED WORK

7.1 Memory Management with Disentanglement

Entanglement detection is motivated by recent results on efficient task parallelism in functional programming languages using disentanglement [Arora et al. 2021; Guatto et al. 2018; Raghunathan et al. 2016; Westrick et al. 2020]. Disentanglement research in turn is motivated by the important role that memory management plays in parallel functional programs and the challenges of ensuring its efficiency and scalability. Although provably space and time efficient memory management is standard for the outdated uniprocessor systems [Jones et al. 2011], it is rare for parallel systems. Other than the recent result of Arora, Westrick, and Acar that relies on disentanglement [Arora et al. 2021], the only other work that comes with guaranteed bounds is by Blelloch and Cheng [Blelloch and Cheng 1999; Cheng and Blelloch 2001] (and its subsequent refinements), which is specialized for real-time garbage collection. Within the world of parallel functional programming, there has been much work on parallel memory managers [Anderson 2010; Auhagen et al. 2011; Doligez and Gonthier 1994; Doligez and Leroy 1993; Domani et al. 2002; Le and Fluet 2015; Marlow and Jones 2011; Sivaramakrishnan et al. 2020]. These approaches typically use processor-local or thread-local heaps combined with a shared global heap that must be collected cooperatively. None of these approaches, however, can guarantee space and work/time bounds.

7.2 Race Detection

Because data races often cause incorrect behavior [Adve 2010; Boehm 2011], there has been much work on detecting races in parallel and concurrent programs. Entanglement detection is similar to race detection in the sense that it concerns a property of memory accesses. Prior research on race detection can broadly be divided into two lines of work: (i) those that target task-parallel programs, and (ii) those that target general concurrency. The first line of work assumes, as we do here, task parallelism, where a program may create many (e.g., millions of) fine-grained threads, which synchronize in a structured fashion. The second line of work assumes a general concurrency setting, where programs contain a small number of coarse-grained threads that may synchronize by using locks, synchronization variables, etc. Techniques from this second line of work do not scale to task-parallel programs because of their coarse-grained threads assumption (e.g,. [Raman et al. 2012]), and are less directly relevant to this paper. We therefore limit our discussion to race detection techniques for task parallelism, and refer the interested reader to prior work ([Flanagan and Freund 2009; Kini et al. 2017; O'Callahan and Choi 2003; Savage et al. 1997; Smaragdakis et al. 2012; Yu et al. 2005]).

Race detection for task-parallel programs. Many algorithms for race detection in task-parallel programs, such as fork-join programs, have been proposed [Bender et al. 2004; Cheng et al. 1998; Feng and Leiserson 1997; Fineman 2005; Mellor-Crummey 1991; Raman et al. 2010, 2012; Utterback et al. 2016; Xu et al. 2020]. These algorithms all revolve around an ordering data structure, sometimes called a *series-parallel order* or *SP-order* data structure, which keeps track of whether two instructions are sequentially dependent or can be executed in parallel. Experiments with state-of-the-art race detection show over an order of magnitude overhead in sequential runs, and parallel runs with race detection typically run slower than the sequential baseline even with over a dozen cores [Utterback et al. 2016]. All of the above work considers nested parallelism with fork-join and async-finish constructs, which result in similar dependency structures. More recent work considers race detection for futures and establishes worst-case bounds, though the overheads are no longer constant [Xu et al. 2020].

Our entanglement detection techniques share the same basic structure as these race detectors, in the sense that for entanglement detection we similarly record the structure of the computation as a graph and query the graph with the help of an order maintenance data structure. However, entanglement detection is more efficient than race detection. This is due to a few differences between the two techniques.

One of the most significant overheads of race detection is incurred by the maintenance of "access histories", which track the history of reads and writes on each individual memory location [Mellor-Crummey 1991; Raman et al. 2012; Utterback et al. 2016]. For example, an array of size N requires N access histories for race detection (one for each index). In contrast, entanglement detection requires only a single annotation (one vertex identifier) for the whole array. Furthermore, as we show in Section 3.5, the space overhead of the annotations for entanglement detection can be further reduced by grouping together the allocations of individual threads, resulting in essentially negligible space overhead. Maintenance of access histories for race detection also incurs a time penalty, particularly on reads, where the thread identifier of the reader is logged at each access.

Finally, we note that entanglement detection benefits from compiler optimizations which perform data inlining (also called "flattening" or "unboxing"). For example, when operating on an array of unboxed integers, entanglement detection requires neither a read barrier nor a write barrier, and therefore incurs no overhead. In contrast, race detection still needs to monitor these operations.

7.3 Parallel Programming Languages

In this paper, we propose a technique for entanglement detection and implemented it as part of the MPL compiler for Parallel ML, which is a functional languages that extends Standard ML with parallelism. As with the Standard ML language, Parallel ML supports references and destructive updates, and allows writing both purely functional and impure (imperative) programs.

Parallel ML builds on a rich history of research or parallel programming languages, including both procedural and functional ones. Programming languages such as Cilk/Cilk++ [Blumofe et al. 1995; Frigo et al. 2009; Intel Corporation 2009a], Cilk-F/L [Singer et al. 2020a, 2019], I-Cilk [Muller et al. 2020], and Intel TBB [Intel Corporation 2009b] extend C/C++ with task parallelism but they all require manual memory management, which is especially challenging for parallel programs. The Rust language offers a type-safe option for systems-level programming [Rust Team 2019] and can ensure memory safety under certain assumptions.

Extensions of the Java language to support parallelism include Fork-Join Java [Lea 2000], and Habanero Java [Imam and Sarkar 2014], all of which support automatic memory management. The X10 [Charles et al. 2005] language is designed with concurrency and parallelism from the beginning and supports both imperative an object-oriented features. Even though these languages simplify writing parallel programs by managing memory automatically, avoiding concurrency bugs can still be challenging, because of the lax control over side effects that these languages offer. Motivated partly by this concern, research on Deterministic Parallel Java [Bocchino, Jr. et al. 2009] develops type systems to guarantee determinism.

Modern type-safe functional programming languages offer substantial control over side effects [Gifford and Lucassen 1986; Kuper and Newton 2013; Kuper et al. 2014; Launchbury and Peyton Jones 1994; Lucassen and Gifford 1988; Park et al. 2008; Peyton Jones and Wadler 1993; Reynolds 1978; Steele 1994; Terauchi and Aiken 2008], which help programmers avoid race conditions, which can harm correctness. Recent projects include Manticore [Fluet et al. 2008, 2011], MultiMLton [Sivaramakrishnan et al. 2014; Ziarek et al. 2011], SML# [Ohori et al. 2018], multicore OCaml [Sivaramakrishnan et al. 2020], and prior work on disentanglement and MPL [Acar et al. 2015; Arora et al. 2021; Guatto et al. 2018; Raghunathan et al. 2016; Westrick et al. 2020]. There has also been significant progress on parallel and concurrent Haskell [Chakravarty et al. 2007; Keller et al. 2010], including work on memory management techniques [Marlow and Jones 2011]. Our work differs from prior work in its emphasis on theoretical guarantees on efficiency and implementations that can match the bounds in practice.

The work presented in this paper and most of the work on parallel programming languages consider compute-intensive jobs, where the main performance metric is throughput. There has recently been interest in "responsive parallelism" which include a broader class of jobs that mix compute-intensive and interactive tasks. The goal in responsive parallelism is to maximize throughput while also minimizing latency for the interactive tasks [Muller et al. 2020, 2017, 2018, 2019]. Responsive parallelism requires new scheduling algorithms that mix competitive and cooperative scheduling [Muller and Acar 2016; Singer et al. 2020b].

We note that even though disentanglement and entanglement have so far been applied to functional programming languages, they are fundamentally a language-agnostic property, and thus could be applied to procedural languages.

8 CONCLUSION

Recent work opened a new angle of attack on the problem of efficiency and scalability for parallel functional programs by utilizing a memory property called disentanglement. There exists, however, no known automated techniques for checking disentanglement, which leaves the burden of this task entirely to the programmer. In this paper, we present techniques for detecting *entanglement* (i.e., violations of disentanglement) dynamically, with low overhead, both in theory and practice. We formalize the approach by presenting a semantics, proving its soundness and completeness, and by presenting techniques for realizing it efficiently. We validate the practicality of the approach by extending the MPL compiler for Parallel ML and confirming empirically that the techniques perform and scale well. In contrast to the related problem of race detection for fine-grained parallel programs, we show that entanglement detection can be performed with nearly zero overhead (both space and time).

Our experience with developing a parallel benchmark suite shows that many parallel programs are naturally disentangled. Perhaps surprisingly, we have found that this holds even for programs that were originally written in low level languages such as C/C++. But, as we consider a broader set of parallel programs and include concurrent programs, which use communication between concurrently executing tasks (or threads), entanglement will arise naturally. In future work, we plan to use entanglement detection to bring the benefits of disentanglement-based memory management to entangled programs by detecting entanglement at run-time and managing it automatically. The result should be a fully general parallel functional programming language that supports effects and thus communication.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their thorough and insightful comments and suggestions. This work was supported by the National Science Foundation under grants CCF-1901381, CCF-2107241, CCF-2119352, and CCF-2115104.

REFERENCES

Umut A. Acar, Guy Blelloch, Matthew Fluet, Stefan K. Muller, and Ram Raghunathan. 2015. Coupling Memory and Computation for Locality Management. In *Summit on Advances in Programming Languages (SNAPL).*

Umut A. Acar, Arthur Charguéraud, and Mike Rainey. 2013. Scheduling Parallel Programs by Work Stealing with Private Deques. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '13).*

Sarita V. Adve. 2010. Data races are evil with no exceptions: technical perspective. *Commun. ACM* 53, 11 (2010), 84.

Daniel Anderson, Guy E. Blelloch, Laxman Dhulipala, Magdalen Dobson, and Yihan Sun. 2022. The problem-based benchmark suite (PBBS), V2. In *PPoPP '22: 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Seoul, Republic of Korea, April 2 - 6, 2022*, Jaejin Lee, Kunal Agrawal, and Michael F. Spear (Eds.). ACM, 445–447. https://doi.org/10.1145/3503221.3508422

Todd A. Anderson. 2010. Optimizations in a private nursery-based garbage collector. In *Proceedings of the 9th International Symposium on Memory Management, ISMM 2010, Toronto, Ontario, Canada, June 5-6, 2010.* 21–30.

Andrew W. Appel. 1989. Simple Generational Garbage Collection and Fast Allocation. *Software Prac. Experience* 19, 2 (1989), 171–183. http://www.cs.princeton.edu/fac/~appel/papers/143.ps

Andrew W. Appel and Zhong Shao. 1996. Empirical and analytic study of stack versus heap cost for languages with closures. *Journal of Functional Programming* 6, 1 (Jan. 1996), 47–74. ftp://daffy.cs.yale.edu/pub/papers/shao/stack.ps

Jatin Arora, Sam Westrick, and Umut A. Acar. 2021. Provably Space Efficient Parallel Functional Programming. In *Proceedings of the 48th Annual ACM Symposium on Principles of Programming Languages (POPL)".*

Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. 2001. Thread Scheduling for Multiprogrammed Multiprocessors. *Theory of Computing Systems* 34, 2 (2001), 115–144.

Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. 1989. I-structures: Data Structures for Parallel Computing. *ACM Trans. Program. Lang. Syst.* 11, 4 (Oct. 1989), 598–632.

Sven Auhagen, Lars Bergstrom, Matthew Fluet, and John H. Reppy. 2011. Garbage collection for multicore NUMA machines. In *Proceedings of the 2011 ACM SIGPLAN workshop on Memory Systems Performance and Correctness (MSPC).* 51–57.

Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Charles E. Leiserson. 2004. On-the-Fly Maintenance of Series-Parallel Relationships in Fork-Join Multithreaded Programs. In *16th Annual ACM Symposium on Parallel Algorithms and Architectures.* 133–144.

Guy E. Blelloch. 1996. Programming Parallel Algorithms. *Commun. ACM* 39, 3 (1996), 85–97.

Guy E. Blelloch, Daniel Anderson, and Laxman Dhulipala. 2020. ParlayLib - A Toolkit for Parallel Algorithms on Shared-Memory Multicore Machines. In *SPAA '20: 32nd ACM Symposium on Parallelism in Algorithms and Architectures, Virtual Event, USA, July 15-17, 2020*, Christian Scheideler and Michael Spear (Eds.). ACM, 507–509. https://doi.org/10.1145/3350755.3400254

Guy E. Blelloch and Perry Cheng. 1999. On Bounding Time and Space for Multiprocessor Garbage Collection. In *Proceedings of SIGPLAN'99 Conference on Programming Languages Design and Implementation (ACM SIGPLAN Notices).* ACM Press, Atlanta, 104–117.

Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. 2012. Internally deterministic parallel algorithms can be fast. In *PPoPP '12* (New Orleans, Louisiana, USA). 181–192.

Guy E. Blelloch, Jonathan C. Hardwick, Jay Sipelstein, Marco Zagha, and Siddhartha Chatterjee. 1994. Implementation of a Portable Nested Data-Parallel Language. *J. Parallel Distrib. Comput.* 21, 1 (1994), 4–14.

Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1995. Cilk: An Efficient Multithreaded Runtime System. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* Santa Barbara, California, 207–216.

Robert D. Blumofe and Charles E. Leiserson. 1998. Space-Efficient Scheduling of Multithreaded Computations. *SIAM J. Comput.* 27, 1 (1998), 202–229.

Robert L Bocchino, Jr., Vikram S. Adve, Sarita V. Adve, and Marc Snir. 2009. Parallel programming must be deterministic by default. In *First USENIX Conference on Hot Topics in Parallelism.*

Hans-Juergen Boehm. 2011. How to Miscompile Programs with "Benign" Data Races. In *3rd USENIX Workshop on Hot Topics in Parallelism, HotPar'11, Berkeley, CA, USA, May 26-27, 2011.*

Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon L. Peyton Jones, Gabriele Keller, and Simon Marlow. 2007. Data parallel Haskell: a status report. In *Proceedings of the POPL 2007 Workshop on Declarative Aspects of Multicore Programming, DAMP 2007, Nice, France, January 16, 2007.* 10–18.

Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (San Diego, CA, USA) *(OOPSLA '05).* ACM, 519–538.

Guang-Ien Cheng, Mingdong Feng, Charles E. Leiserson, Keith H. Randall, and Andrew F. Stark. 1998. Detecting data races in Cilk programs that use locks. In *Proceedings of the 10th ACM Symposium on Parallel Algorithms and Architectures (SPAA '98)*.

Perry Cheng and Guy Blelloch. 2001. A Parallel, Real-Time Garbage Collector. In *Proceedings of SIGPLAN 2001 Conference on Programming Languages Design and Implementation (ACM SIGPLAN Notices)*. ACM Press, Snowbird, Utah, 125–136.

Damien Doligez and Georges Gonthier. 1994. Portable, Unobtrusive Garbage Collection for Multiprocessor Systems. In *Conference Record of the Twenty-first Annual ACM Symposium on Principles of Programming Languages (ACM SIGPLAN Notices)*. ACM Press, Portland, OR. ftp://ftp.inria.fr/INRIA/Projects/para/doligez/DoligezGonthier94.ps.gz

Damien Doligez and Xavier Leroy. 1993. A Concurrent Generational Garbage Collector for a Multi-Threaded Implementation of ML. In *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages (ACM SIGPLAN Notices)*. ACM Press, 113–123. file://ftp.inria.fr/INRIA/Projects/cristal/Xavier.Leroy/publications/concurrent-gc.ps.gz

Tamar Domani, Elliot K. Kolodner, Ethan Lewis, Erez Petrank, and Dafna Sheinwald. 2002. Thread-Local Heaps for Java. In *ISMM'02 Proceedings of the Third International Symposium on Memory Management (ACM SIGPLAN Notices)*, David Detlefs (Ed.). ACM Press, Berlin, 76–87. http://www.cs.technion.ac.il/~erez/publications.html

Mingdong Feng and Charles E. Leiserson. 1997. Efficient Detection of Determinacy Races in Cilk Programs. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*. 1–11.

Jeremy T. Fineman. 2005. *Provably Good Race Detection That Runs in Parallel*. Master's thesis. Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, Cambridge, MA.

Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: efficient and precise dynamic race detection. *SIGPLAN Not.* 44, 6 (June 2009), 121–133. https://doi.org/10.1145/1543135.1542490

Matthew Fluet, Mike Rainey, and John Reppy. 2008. A scheduling framework for general-purpose parallel languages. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*.

Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. 2011. Implicitly threaded parallelism in Manticore. *Journal of Functional Programming* 20, 5-6 (2011), 1–40.

Matteo Frigo, Pablo Halpern, Charles E. Leiserson, and Stephen Lewin-Berlin. 2009. Reducers and Other Cilk++ Hyperobjects. In *21st Annual ACM Symposium on Parallelism in Algorithms and Architectures*. 79–90.

David K. Gifford and John M. Lucassen. 1986. Integrating Functional and Imperative Programming. In *Proceedings of the ACM Symposium on Lisp and Functional Programming (LFP)*. ACM Press, 22–38.

Marcelo J. R. Gonçalves. 1995. *Cache Performance of Programs with Intensive Heap Allocation and Generational Garbage Collection*. Ph. D. Dissertation. Department of Computer Science, Princeton University.

Marcelo J. R. Gonçalves and Andrew W. Appel. 1995. Cache Performance of Fast-Allocating Programs. In *Record of the 1995 Conference on Functional Programming and Computer Architecture*.

Adrien Guatto, Sam Westrick, Ram Raghunathan, Umut A. Acar, and Matthew Fluet. 2018. Hierarchical memory management for mutable state. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2018, Vienna, Austria, February 24-28, 2018*. 81–93.

Robert H. Halstead, Jr. 1984. Implementation of Multilisp: Lisp on a Multiprocessor. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming* (Austin, Texas, United States) *(LFP '84)*. ACM, 9–17.

Kevin Hammond. 2011. Why Parallel Functional Programming Matters: Panel Statement. In *Reliable Software Technologies - Ada-Europe 2011 - 16th Ada-Europe International Conference on Reliable Software Technologies, Edinburgh, UK, June 20-24, 2011. Proceedings*. 201–205.

Shams Mahmood Imam and Vivek Sarkar. 2014. Habanero-Java library: a Java 8 framework for multicore programming. In *2014 International Conference on Principles and Practices of Programming on the Java Platform Virtual Machines, Languages and Tools, PPPJ '14*. 75–86.

Intel Corporation 2009a. *Intel Cilk++ SDK Programmer's Guide*. Intel Corporation. Document Number: 322581-001US.

Intel Corporation 2009b. *Intel(R) Threading Building Blocks*. Intel Corporation. Available from http://www.threadingbuildingblocks.org/documentation.php.

Richard Jones, Antony Hosking, and Eliot Moss. 2011. *The garbage collection handbook: the art of automatic memory management*. Chapman & Hall/CRC.

Gabriele Keller, Manuel M.T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Ben Lippmeier. 2010. Regular, shape-polymorphic, parallel arrays in Haskell. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming* (Baltimore, Maryland, USA) *(ICFP '10)*. 261–272.

Dileep Kini, Umang Mathur, and Mahesh Viswanathan. 2017. Dynamic race prediction in linear time. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 157–170.

Lindsey Kuper and Ryan R Newton. 2013. LVars: lattice-based data structures for deterministic parallelism. In *Proceedings of the 2nd ACM SIGPLAN workshop on Functional high-performance computing*. ACM, 71–84.

Lindsey Kuper, Aaron Todd, Sam Tobin-Hochstadt, and Ryan R. Newton. 2014. Taming the Parallel Effect Zoo: Extensible Deterministic Parallelism with LVish. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. ACM, New York, NY, USA, 2–14. https://doi.org/10.1145/2594291.2594312

John Launchbury and Simon L. Peyton Jones. 1994. Lazy Functional State Threads. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI), Orlando, Florida, USA, June 20-24, 1994*. 24–35.

Matthew Le and Matthew Fluet. 2015. Partial Aborts for Transactions via First-class Continuations. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming* (Vancouver, BC, Canada) *(ICFP 2015)*. 230–242.

Doug Lea. 2000. A Java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande* (San Francisco, California, USA) *(JAVA '00)*. 36–43.

Peng Li, Simon Marlow, Simon L. Peyton Jones, and Andrew P. Tolmach. 2007. Lightweight concurrency primitives for GHC. In *Proceedings of the ACM SIGPLAN Workshop on Haskell, Haskell 2007, Freiburg, Germany, September 30, 2007*. 107–118.

J. M. Lucassen and D. K. Gifford. 1988. Polymorphic Effect Systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '88)*. ACM, New York, NY, USA, 47–57.

Simon Marlow and Simon L. Peyton Jones. 2011. Multicore garbage collection with local heaps. In *Proceedings of the 10th International Symposium on Memory Management, ISMM 2011, San Jose, CA, USA, June 04 - 05, 2011*, Hans-Juergen Boehm and David F. Bacon (Eds.). ACM, 21–32.

John Mellor-Crummey. 1991. On-the-fly Detection of Data Races for Programs with Nested Fork-Join Parallelism. In *Proceedings of Supercomputing'91*. 24–33.

MLton n.d.. MLton web site. http://www.mlton.org.

Stefan Muller, Kyle Singer, Noah Goldstein, Umut A. Acar, Kunal Agrawal, and I-Ting Angelina Lee. 2020. Responsive Parallelism with Futures and State. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*.

Stefan K. Muller and Umut A. Acar. 2016. Latency-Hiding Work Stealing: Scheduling Interacting Parallel Computations with Work Stealing. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2016, Asilomar State Beach/Pacific Grove, CA, USA, July 11-13, 2016*. 71–82.

Stefan K. Muller, Umut A. Acar, and Robert Harper. 2017. Responsive Parallel Computation: Bridging Competitive and Cooperative Threading. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. ACM, New York, NY, USA, 677–692.

Stefan K. Muller, Umut A. Acar, and Robert Harper. 2018. Competitive Parallelism: Getting Your Priorities Right. *Proc. ACM Program. Lang.* 2, ICFP, Article 95 (July 2018), 30 pages.

Stefan K. Muller, Sam Westrick, and Umut A. Acar. 2019. Fairness in Responsive Parallelism. In *Proceedings of the 24th ACM SIGPLAN International Conference on Functional Programming (ICFP 2019)*.

Robert O'Callahan and Jong-Deok Choi. 2003. Hybrid dynamic data race detection. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2003, June 11-13, 2003, San Diego, CA, USA*, Rudolf Eigenmann and Martin C. Rinard (Eds.). ACM, 167–178.

Atsushi Ohori, Kenjiro Taura, and Katsuhiro Ueno. 2018. Making SML# a General-purpose High-performance Language. Unpublished Manuscript.

Sungwoo Park, Frank Pfenning, and Sebastian Thrun. 2008. A Probabilistic Language Based on Sampling Functions. *ACM Trans. Program. Lang. Syst.* 31, 1, Article 4 (Dec. 2008), 46 pages.

Simon L. Peyton Jones, Roman Leshchinskiy, Gabriele Keller, and Manuel M. T. Chakravarty. 2008. Harnessing the Multicores: Nested Data Parallelism in Haskell. In *FSTTCS*. 383–414.

Simon L. Peyton Jones and Philip Wadler. 1993. Imperative Functional Programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Charleston, South Carolina, USA) *(POPL '93)*. 71–84.

Ram Raghunathan, Stefan K. Muller, Umut A. Acar, and Guy Blelloch. 2016. Hierarchical Memory Management for Parallel Programs. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming* (Nara, Japan) *(ICFP 2016)*. ACM, New York, NY, USA, 392–406.

Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. 2010. Efficient Data Race Detection for Async-Finish Parallelism. In *Runtime Verification*, Howard Barringer, Ylies Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon Pace, Grigore Rosu, Oleg Sokolsky, and Nikolai Tillmann (Eds.). Lecture Notes in Computer Science, Vol. 6418. Springer Berlin / Heidelberg, 368–383.

Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin Vechev, and Eran Yahav. 2012. Scalable and Precise Dynamic Datarace Detection for Structured Parallelism. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. 531–542.

John C. Reynolds. 1978. Syntactic Control of Interference. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Tucson, Arizona) *(POPL '78)*. ACM, New York, NY, USA, 39–46.

Rust Team. 2019. Rust Language. https://www.rust-lang.org/

Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: A Dynamic Race Detector for Multi-Threaded Programs. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP)*.

Julian Shun and Guy E. Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *PPOPP '13*. ACM, New York, NY, USA, 135–146.

Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. 2012. Brief Announcement: The Problem Based Benchmark Suite. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures* (Pittsburgh, Pennsylvania, USA) *(SPAA '12)*. 68–70.

Kyle Singer, Kunal Agrawal, and I-Ting Angelina Lee. 2020a. Scheduling I/O Latency-Hiding Futures in Task-Parallel Platforms. In *1st Symposium on Algorithmic Principles of Computer Systems, APOCS 2020, Salt Lake City, UT, USA, January 8, 2020*, Bruce M. Maggs (Ed.). SIAM, 147–161. https://doi.org/10.1137/1.9781611976021.11

Kyle Singer, Noah Goldstein, Stefan K. Muller, Kunal Agrawal, I-Ting Angelina Lee, and Umut A. Acar. 2020b. Priority Scheduling for Interactive Applications. In *SPAA '20: 32nd ACM Symposium on Parallelism in Algorithms and Architectures, Virtual Event, USA, July 15-17, 2020*, Christian Scheideler and Michael Spear (Eds.). 465–477.

Kyle Singer, Yifan Xu, and I-Ting Angelina Lee. 2019. Proactive Work Stealing for Futures. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming* (Washington, District of Columbia) *(PPoPP '19)*. ACM, New York, NY, USA, 257–271. https://doi.org/10.1145/3293883.3295735

K. C. Sivaramakrishnan, Stephen Dolan, Leo White, Sadiq Jaffer, Tom Kelly, Anmol Sahoo, Sudha Parimala, Atul Dhiman, and Anil Madhavapeddy. 2020. Retrofitting parallelism onto OCaml. *Proc. ACM Program. Lang.* 4, ICFP (2020), 113:1–113:30.

K. C. Sivaramakrishnan, Lukasz Ziarek, and Suresh Jagannathan. 2014. MultiMLton: A multicore-aware runtime for standard ML. *Journal of Functional Programming* FirstView (6 2014), 1–62.

Yannis Smaragdakis, Jacob Evans, Caitlin Sadowski, Jaeheon Yi, and Cormac Flanagan. 2012. Sound predictive race detection in polynomial time. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, John Field and Michael Hicks (Eds.). ACM, 387–400.

Daniel Spoonhower. 2009. *Scheduling Deterministic Parallel Programs*. Ph. D. Dissertation. Carnegie Mellon University. https://www.cs.cmu.edu/~rwh/theses/spoonhower.pdf

Daniel Spoonhower, Guy E. Blelloch, Phillip B. Gibbons, and Robert Harper. 2009. Beyond Nested Parallelism: Tight Bounds on Work-stealing Overheads for Parallel Futures. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures* (Calgary, AB, Canada) *(SPAA '09)*. ACM, New York, NY, USA, 91–100.

Guy L. Steele, Jr. 1994. Building Interpreters by Composing Monads. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon, USA) *(POPL '94)*. ACM, New York, NY, USA, 472–492.

Yihan Sun, Daniel Ferizovic, and Guy E. Blelloch. 2018. PAM: parallel augmented maps. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2018, Vienna, Austria, February 24-28, 2018*, Andreas Krall and Thomas R. Gross (Eds.). ACM, 290–304. https://doi.org/10.1145/3178487.3178509

Tachio Terauchi and Alex Aiken. 2008. Witnessing Side Effects. *ACM Trans. Program. Lang. Syst.* 30, 3, Article 15 (May 2008), 42 pages.

Robert Utterback, Kunal Agrawal, Jeremy T. Fineman, and I-Ting Angelina Lee. 2016. Provably Good and Practically Efficient Parallel Race Detection for Fork-Join Programs. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2016, Asilomar State Beach/Pacific Grove, CA, USA, July 11-13, 2016*. 83–94.

Sam Westrick, Jatin Arora, and Umut Acar. 2022a. *Provably Space-Efficient Parallel Functional Programming*. https://blog.sigplan.org/2022/01/13/provably-space-efficient-parallel-functional-programming/

Sam Westrick, Jatin Arora, and Umut A. Acar. 2022b. *Entanglement Detection With Near-Zero Cost: Artifact*. https://doi.org/10.5281/zenodo.6671887

Sam Westrick, Mike Rainey, Daniel Anderson, and Guy E. Blelloch. 2022c. Parallel block-delayed sequences. In *PPoPP '22: 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Seoul, Republic of Korea, April 2 - 6, 2022*, Jaejin Lee, Kunal Agrawal, and Michael F. Spear (Eds.). ACM, 61–75. https://doi.org/10.1145/3503221.3508434

Sam Westrick, Larry Wang, and Umut A. Acar. 2022d. DePa: Simple, Provably Efficient, and Practical Order Maintenance for Task Parallelism. *CoRR* abs/2204.14168 (2022). https://doi.org/10.48550/arXiv.2204.14168 arXiv:2204.14168

Sam Westrick, Rohan Yadav, Matthew Fluet, and Umut A. Acar. 2020. Disentanglement in Nested-Parallel Programs. In *Proceedings of the 47th Annual ACM Symposium on Principles of Programming Languages (POPL)"*.

Yifan Xu, Kyle Singer, and I-Ting Angelina Lee. 2020. Parallel determinacy race detection for futures. In *PPoPP '20: 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, California, USA, February 22-26,* *2020*, Rajiv Gupta and Xipeng Shen (Eds.). ACM, 217–231. https://doi.org/10.1145/3332466.3374536

Yuan Yu, Tom Rodeheffer, and Wei Chen. 2005. RaceTrack: efficient detection of data race conditions via adaptive tracking. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles 2005, SOSP 2005, Brighton, UK, October 23-26, 2005*, Andrew Herbert and Kenneth P. Birman (Eds.). ACM, 221–234.

Lukasz Ziarek, K. C. Sivaramakrishnan, and Suresh Jagannathan. 2011. Composable asynchronous events. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*. 628–639.