Efficient Parallel Functional Programming with Effects

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

Although functional programming languages simplify writing safe parallel programs by helping programmers to avoid data races, they have traditionally delivered poor performance. Recent work improved performance by using a hierarchical memory architecture that allows processors to allocate and reclaim memory independently without any synchronization, solving thus the key performance challenge afflicting functional programs. The approach, however, restricts mutation, or memory effects, so as to ensure "disentanglement", a low-level memory property that guarantees independence between different heaps in the hierarchy.

This paper proposes techniques for supporting entanglement and for allowing functional programs to use mutation at will. Our techniques manage entanglement by distinguishing between disentangled and entangled objects and shielding disentangled objects from the cost of entanglement management. We present a semantics that formalizes entanglement as a property at the granularity of memory objects, and define several cost metrics to reason about and bound the time and space cost of entanglement. We present an implementation of the techniques by extending the MPL compiler for Parallel ML. The extended compiler supports all features of the Parallel ML language, including unrestricted effects. Our experiments using a variety of benchmarks show that MPL incurs a small time and space overhead compared to sequential runs, scales well, and is competitive with languages such as C++, Go, Java, OCaml. These results show that our techniques can marry the safety benefits of functional programming with performance.

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

Additional Key Words and Phrases: functional programming, memory management, parallel, concurrent

ACM Reference Format:
Jatin Arora, Sam Westrick, and Umut A. Acar. 2023. Efficient Parallel Functional Programming with Effects. *Proc. ACM Program. Lang.* 7, PLDI, Article 170 (June 2023), 26 pages. https://doi.org/10.1145/3591284

1 INTRODUCTION

Architectural advances of the past decade have motivated interest in parallel programming languages that can utilize modern multicore chip. To solve the challenges of parallel programming, researchers have proposed functional languages because they can control and even outlaw pernicious data races [Adve 2010; Boehm 2011] and can naturally express bulk-parallel operations. Parallel functional programming has, however, traditionally failed to deliver scalable performance, because of its hunger for memory, which only increases with parallelism as multiple processor allocate simultaneously. Tackling this problem proved difficult and even after decades of research going back to 1980s performance, until recently, remained poor, especially compared to other managed languages [Arvind et al. 1989; Blelloch 1996; Blelloch et al. 1994; Collins et al. 2014; Fluet et al. 2011; Guatto et al. 2018; Halstead 1984; Hammond 2011; Li et al. 2007; Marlow and Jones 2011;

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

2475-1421/2023/6-ART170
https://doi.org/10.1145/3591284

Ohori et al. 2018; Peyton Jones et al. 2008; Raghunathan et al. 2016; Sivaramakrishnan et al. 2014; Spoonhower 2009; Ziarek et al. 2011].

Recent work develops promising techniques to manage memory in parallel functional programs by exploiting a property called disentanglement [Arora et al. 2021; Westrick et al. 2020]. Westrick et al define disentanglement as a program property which guarantees that concurrent threads of a program are oblivious to each other's memory allocations. They show that determinacy race free programs are disentangled and propose a hierarchical heap architecture for managing disentangled programs. In followup work, Arora et al present a "heap scheduler", which determines the time and place (processor) for garbage collection of heaps in the hierarchy and guarantees space and work efficiency [Arora et al. 2021]. Arora et al's experimental results show important promise: assuming disentanglement, parallel functional programs can deliver efficiency and scalability.

These results on disentanglement take a significant step towards bridging performance and safety in parallel functional programs. But the work assumes disentanglement. This is a significant restriction because it is difficult for a programmer to predict and eliminate all entanglement. In addition, many interesting parallel programs use mutation in ways that create entanglement, by for example, using various forms of concurrency to communicate between concurrent tasks [Acar et al. 2017; Feldman et al. 2020; Fraser 2003; Gramoli 2015; Harris 2001; Kumar et al. 2017; Lindén and Jonsson 2013; Michael and Scott 1996; Prabhu et al. 2018; Shun and Blelloch 2014].

In this paper, we design memory management techniques that support all programs, including entangled ones, while preserving the benefits of the disentanglement-based designs, and realize the following scalability properties:

- each thread can allocate memory without synchronizing with other threads,
- each thread can reclaim memory without synchronizing with other threads,
- threads may share objects without any copy operations (a.k.a promotions) or object movement at a cost proportional to the amount of entanglement.

It is not difficult to support entanglement without retaining the properties above. For example, we could synchronize concurrent tasks that communicate via a shared cell, we could rely on a stop-the-world collection algorithm, or we could promote objects accessed by concurrent tasks/threads [Guatto et al. 2018]. Our goal is to avoid all synchronization and promotion and ensure good performance and scalability.

We achieve this goal by introducing the concept of object-level entanglement, a concept that treats entanglement as a continuous property of memory objects as opposed to prior treatment of entanglement as an all-or-none property of programs. To this end, we present a semantics that classifies each memory object as entangled or disentangled (Section 2). To determine when an object becomes entangled, the semantics tracks the threads forked by the program and observes their memory actions. When threads join, the semantics resolves any entanglement that they created and updates the corresponding objects to be disentangled. To quantify the amount of entanglement, the semantics computes two cost metrics, entanglement factor and entanglement ceiling, allowing us to account for the work/time ond space cost of our memory manager.

We present a memory manager that implements the semantics to distinguish between disentangled and entangled objects and manages them within a hierarchical heap memory architecture (Section 3). Our memory manager organizes memory as a dynamic hierarchy of heaps that mirrors the parallel structure of a computation; it does this by creating/joining heaps for each task fork/join operation. Within the heap hierarchy, the memory manager distinguishes between disentangled and entangled objects by observing the memory actions of parallel threads using read/write barriers. For efficiency, the memory manager only uses read and write barriers for mutable data and does not use any barriers for immutable data and furthermore, the memory manager tracks entangled obejcts in a fully concurrent non-blocking fashion, i.e., without requiring any concurrent tasks to synchronize and being lock free. The memory manager also tracks when threads join and resolves the entanglement between them punctually, thereby ensuring that entanglement does not "snowball" to include a large number of objects. We show that the memory manager implements the semantics precisely, spending work and space proportional to the cost metrics defined by the semantics.

The memory manager allows threads to allocate and reclaim memory independently with the requirement that they may not relocate entangled objects. To support this, we design a "hybrid" collection algorithm that mixes non-moving and copying collection. The algorithm pins live entangled objects in place and copy-collects disentangled objects, satisfying the non-moving requirement for entangled objects and also retaining the performance benefits of copying reclamation. This algorithm solves a key challenge in memory management for hierarchical heaps which is to avoid synchronization and blocking so as not to harm the scalability primitives by creating "choke points" that require synchronization between concurrently executing tasks or garbage collections.

The proposed techniques use a relatively sophisticated memory architecture and lean heavily on concurrency for efficiency and scalability. A natural concern therefore is practicality: can these techniques be made to work well in practice. This paper includes a major implementation effort that extends the MPL compiler suite for the Parallel ML language to support our techniques. The extended compiler is able to compile all Parallel ML programs including unrestricted use of references and mutation. Using this new compiler, we have implemented a variety of benchmarks, including sophisticated parallel algorithms that use state-of-the-art nonblocking concurrent data structures. Our experiments show that MPL performs well, incurring relatively small overheads compared to sequential runs, and scaling well as the number of cores increase. Notably, MPL delivers performance and compactness simultaneously: parallel runs on dozens of cores usually consume less memory than sequential runs and deliver significant speedups. We also perform a more modest comparison with several other languages including Go, Java, Multicore OCaml, and C++. Our results show that MPL is competitive with these languages.

This paper shows that parallel functional programming—long lauded for its safety benefits—can perform and scale well. Specific contributions that make this result possible include the following.

- Realization of entanglement as a continous property
- A semantics for identifying entanglement and metrics for quantifying its cost
- Memory management techniques for tracking and managing entanglement
- Proofs that the proposed tracking techniques incur small work/time and space overheads in terms of the cost metrics
- An implementation that extends the MPL compiler for Parallel ML.
- Parallel benchmarks that use a variety of mutation (memory effects) and their evaluation

2 LANGUAGE AND ENTANGLEMENT

To characterize entanglement as an object-level property, we give a call-by-value functional language with fork-join parallelism and mutable references. The language's operational semantics classifies memory objects as entangled or disentangled at each step of the evaluation. It identifies entangled objects by tracking the parallel structure of evaluation and the memory accesses of mutable references. At the end of evaluation, it computes two cost metrics: an ***entanglement factor***, ϵ, which quantifies the total amount (size) of entangled objects, and an ***entanglement ceiling***, δ, which tracks the peak amount (size) of entangled objects at any step.

We use the cost metrics to bound the overheads of our entanglement tracking techniques (Section 3). In particular, we show that the techniques require $O(\epsilon)$ extra work $O(\delta)$ extra space.

Types	τ	$::=$	$\mathsf{nat} \mid \tau \times \tau \mid \tau \to \tau \mid \tau\ \mathsf{ref}$
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 Locations	ℓ		
Numbers	m	$\in$	$\mathbb{N}$
Variables	x, f		
Storables	s	$::=$	$m \mid \mathsf{fun}\ f\ x\ \mathsf{is}\ e \mid \langle \ell, \ell \rangle \mid \mathsf{ref}\ \ell \mid \mathsf{ref}\ \ell$
Memory	μ	$\in$	*Locations* $\rightharpoonup$ *Storables*
Task Identifiers	v, u		
Task Tree	T	$::=$	$\mathsf{Leaf}(v) \mid \mathsf{Par}(v, T, T)$
Reader History	H	$\in$	*Locations* $\rightharpoonup 2^{\mathit{Tasks}}$
Allocation Map	α	$\in$	*Locations* $\rightharpoonup$ *Tasks*

Fig. 1. Syntax

We note that our semantics only observes accesses on mutable objects and does not track accesses on immutable objects. This allows us to skip read barriers on immutable objects which is crucial for the performance of our entanglement tracking techniques.

2.1 Syntax

The syntax of the language, shown in Figure 1, includes the usual constructs and in addition contains parallel pairs $\langle e_1 \parallel e_2 \rangle$ where e_1 and e_2 may execute in parallel. It includes expressions $\mathsf{ref}\ e$, $!\, e$, and, $e_1 := e_2$ which introduce mutable references and operate on them. We can easily extend the syntax, if desired, to have other mutable objects such as arrays.

To track memory operations, the language defines ***storables*** s, which are allocated in memory, and ***memory locations*** ℓ, which are indices into the memory. The language steps storables to locations by allocating new locations in a memory store μ and mapping them to the corresponding storables. We write $\mu[\ell \hookrightarrow s]$ to extend the memory μ by mapping ℓ to s and write $\mu(\ell)$ to refer the storable assigned to location ℓ.

Locations are the only irreducible form of the language. Locations that are mapped to mutable references are called ***mutable locations*** and those that are mapped to other types of storables are called ***immutable locations***. We say that a location ℓ has a pointer to another location ℓ' if the storable at location ℓ mentions ℓ'. The pointer is mutable if the location ℓ is mutable and otherwise, if the location ℓ is immutable, its pointers are immutable.

2.2 Task Tree

When evaluating a program, the semantics tracks the structure of evaluation using a ***task tree***. The task tree arranges the tasks of the program with vertices that represent tasks and edges that express parent/child relationships between tasks. Each task is either a ***leaf task*** that has no children, or an ***internal task*** that has children and is suspended while waiting for its children to complete.

When the evaluation starts, the task tree only contains the root task. The tree grows when a leaf task *forks* and creates two child tasks in the tree; this suspends the parent until the children are evaluated. The tree shrinks when the child tasks *join* after they finish as they are removed from the tree; this makes their parent a leaf and it resumes.

We label the tasks of the task tree with identifiers u, v, etc. We represent the leaves of the tree with the form $\mathsf{Leaf}(u)$ and internal vertices with the form $\mathsf{Par}(u, T_1, T_2)$, where T_1 and T_2 are subtrees.

2.3 Entanglement Sources, Regions, and Cost Metrics

Tasks create entanglement when they read locations allocated by other concurrent tasks. To track entanglement, the semantics intercepts the reads of mutable references and identifies *entanglement sources.* Each entanglement source causes a region of memory to become entangled.

Entanglement sources. A memory location ℓ becomes an ***entanglement source*** when:

- a task u performs a mutable read (dereference) and obtains ℓ as the result, and
- the task u is concurrent with the task that allocated ℓ.

The semantics identifies entanglement sources by maintaining an ***allocator map*** α, mapping each location to the task which allocates it, and a ***reader history*** H, mapping each location to a set of tasks where each task has performed a dereference operation resulting in that location. Given a task tree T, reader history H, and allocation map α, the ***set of entanglement sources***, $\mathcal{E}(H)$, is:

$$\mathcal{E}(H) = \{\ell \mid \exists\, u \in H(\ell) : \mathsf{concurrent}(u, \alpha(\ell))\},$$

where the relation $\mathsf{concurrent}(u, v)$ holds if neither u nor v is an ancestor of the other in tree T.

Entanglement region. An entanglement source affects its vicinity: a task can use the entanglement source to read locations that are reachable from it and create more entanglement. If the source is mutable, such reads will be accounted by the reader history. However, the reader history does not track reads of immutable locations.

To account for this, we associate with each entanglement source an ***entanglement region*** which contains all locations reachable from the source using immutable pointers only. We can formally define it as follows: let $\mathsf{out}(\ell)$ be the set of locations to which location ℓ points to. Then the entanglement region of location ℓ, written $\mathsf{er}(\ell)$, is defined by the recurrence:

$$\mathsf{er}(\ell) = \begin{cases} \{\ell\} \cup \displaystyle\bigcup_{\ell' \in \mathsf{out}(\ell)} \mathsf{er}(\ell'), & \text{if } \ell \text{ is immutable} \\ \{\ell\} & \text{if } \ell \text{ is mutable} \end{cases}$$

If a location ℓ belongs to an entanglement region, we say that ℓ is ***entangled***.

Cost metrics. To quantify the amount of entanglement, the semantics calculates entanglement factor and entanglement ceiling. The ***entanglement factor***, ϵ, accumulates the amount of memory that becomes entangled throughout an execution. The ***entanglement ceiling***, δ, tracks the maximum amount of entangled objects at any point of the execution. The ceiling provides a reference for the peak amount of entanglement by accounting for the fact that this amount decreases when tasks join. As an example, consider two parallel subcomputations whose executions have entanglement factors ϵ_1, ϵ_2 and ceilings δ_1, δ_2. Then for their sequential composition, the entanglement factor is $\epsilon_1 + \epsilon_2$ and the ceiling is $\max(\delta_1, \delta_2)$. Because the first computation finishes before the second begins, its entanglement is resolved and does not affect the second. The entanglement ceiling accounts for this resolution.

2.4 Semantics

Our semantics, presented in Figure 2, is a transition relation that steps a program state consisting of six components: allocator map α, reader history H, memory μ, entanglement factor ϵ, task tree T, and expression e. The relation has the form: $(\alpha\,; H\,; \mu\,; \epsilon\,; T\,; e) \longrightarrow (\alpha'\,; H'\,; \mu'\,; \epsilon'\,; T'\,; e')$.

Fork and parallel evaluation. The rule Fork (see Figure 2) forks two children, v and w to evaluate the components of the parallel pair $\langle e_1 \parallel e_2 \rangle$. It creates the par-node $\mathsf{Par}(u, \mathsf{Leaf}(v), \mathsf{Leaf}(w))$. Then, the rule ParL steps the left side of the parallel pair e_1 with subtree T_1. The rule ParR for stepping the right side is similar. Rules ParL and ParR may interleave non-deterministically.

$$\frac{\ell \notin \mathrm{dom}(\mu) \qquad \mu' = \mu[\ell \hookrightarrow s] \qquad \alpha' = \alpha[\ell \hookrightarrow v]}{\alpha \,;\, \epsilon \,;\, H \,;\, \mu \,;\, \mathsf{Leaf}(v) \,;\, s \longrightarrow \alpha' \,;\, \epsilon \,;\, H \,;\, \mu' \,;\, \mathsf{Leaf}(v) \,;\, \ell} \ \textsc{Alloc}$$

$$\frac{\alpha \,;\, \epsilon \,;\, H \,;\, \mu \,;\, T \,;\, e \longrightarrow \alpha' \,;\, \epsilon' \,;\, H' \,;\, \mu' \,;\, T' \,;\, e'}{\alpha \,;\, \epsilon \,;\, H \,;\, \mu \,;\, T \,;\, (\mathsf{ref}\ e) \longrightarrow \alpha' \,;\, \epsilon' \,;\, H' \,;\, \mu' \,;\, T' \,;\, (\mathsf{ref}\ e')} \ \textsc{RefS}$$

$$\frac{\alpha \,;\, \epsilon \,;\, H \,;\, \mu \,;\, T \,;\, e \longrightarrow \alpha' \,;\, \epsilon' \,;\, H' \,;\, \mu' \,;\, T' \,;\, e'}{\alpha \,;\, \epsilon \,;\, H \,;\, \mu \,;\, T \,;\, (!\, e) \longrightarrow \alpha' \,;\, \epsilon' \,;\, H' \,;\, \mu' \,;\, T' \,;\, (!\, e')} \ \textsc{BangS}$$

$$\frac{\mu(\ell) = \mathsf{ref}\ \ell' \qquad H' = H[\ell' \hookrightarrow H(\ell') \cup \{v\}] \qquad \epsilon' = \epsilon + |\mathsf{er}(\mathcal{E}(H')) \setminus \mathsf{er}(\mathcal{E}(H))|}{\alpha \,;\, \epsilon \,;\, H \,;\, \mu \,;\, \mathsf{Leaf}(v) \,;\, (!\, \ell) \longrightarrow \alpha \,;\, \epsilon \,;\, H' \,;\, \mu \,;\, \mathsf{Leaf}(v) \,;\, \ell'} \ \textsc{Bang}$$

$$\frac{}{\alpha \,;\, \epsilon \,;\, H \,;\, \mu_0[\ell \hookrightarrow \mathsf{ref}\ \ell''] \,;\, \mathsf{Leaf}(v) \,;\, (\ell := \ell') \longrightarrow \alpha \,;\, \epsilon \,;\, H' \,;\, \mu_0[\ell \hookrightarrow \mathsf{ref}\ \ell'] \,;\, \mathsf{Leaf}(v) \,;\, \ell'} \ \textsc{Upd}$$

$$\frac{}{\alpha \,;\, \epsilon \,;\, H \,;\, \mu \,;\, \mathsf{Leaf}(u) \,;\, \langle e_1 \parallel e_2 \rangle \longrightarrow \alpha \,;\, \epsilon \,;\, H \,;\, \mu \,;\, \mathsf{Par}(u, \mathsf{Leaf}(v), \mathsf{Leaf}(w)) \,;\, \langle e_1 \parallel e_2 \rangle} \ \textsc{Fork}$$

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

$$\frac{\alpha' = \alpha\{u/v\}\{u/w\} \qquad H' = H\{u/v\}\{u/w\}}{\alpha \,;\, \epsilon \,;\, H \,;\, \mu \,;\, \mathsf{Par}(u, \mathsf{Leaf}(v), \mathsf{Leaf}(w)) \,;\, \langle \ell_1 \parallel \ell_2 \rangle \longrightarrow \alpha' \,;\, \epsilon \,;\, H' \,;\, \mu \,;\, \mathsf{Leaf}(u) \,;\, \langle \ell_1, \ell_2 \rangle} \ \textsc{Join}$$

Fig. 2. Language Dynamics.

Allocation. The rule Alloc (see Figure 2) shows how storables are allocated. The rule steps a storable s to a fresh location ℓ ($\ell \notin \mathrm{dom}(\mu)$), and extends the memory store μ with location ℓ mapped to storable s. It assigns the location its allocator task in the allocator map α.

Mutable reads and entanglement factor. The rules RefS and BangS step `ref` e and $!\,e$ respectively by stepping the sub expression e.

Rule Bang performs the dereference operation and updates the reader history and the entanglement factor. When a task v dereferences a mutable location ℓ to read a location ℓ', the rule adds the reader v to the reader history of location ℓ'. Because this dereference may increase the amount of entanglement, the rule updates the entanglement factor. To update the entanglement factor, the rule first identifies locations entangled because of this read. We can write the set of entangled locations as $\cup_{\ell \in \mathcal{E}(H)} \mathsf{er}(\ell)$, where er computes the region for a source, H is a reader history, and $\mathcal{E}(H)$ contains all the sources for history H. This set is the union of entanglement regions of all the entanglement sources. We write this concisely as $\mathsf{er}(\mathcal{E}(H))$.

The rule computes the set difference $\mathsf{er}(\mathcal{E}(H')) \setminus \mathsf{er}(\mathcal{E}(H))$ because the difference represents all entangled locations in the new (after the read) reader history H' that were not entangled in the old (before the read) history H. The rule increases the entanglement factor by the size of this set, which it computes with $|.|$ (see Figure 2).

Mutable updates. Rule Upd creates a pointer from a mutable location ℓ to a location ℓ'. Mutable updates do not affect entanglement because the task already can access both locations.

Join. Once a parallel pair is fully evaluated, we turn it into a sequential tuple. When this happens, we remove the children from the tree and as a result, the parent becomes the leaf. The rule Join replaces the par-node $\mathsf{Par}(u, \mathsf{Leaf}(v), \mathsf{Leaf}(w))$ with the leaf node $\mathsf{Leaf}(u)$. Because the children have terminated, we transfer the ownership of the allocations of the children to the parent. The Join rule does this by substituting task u for the children tasks v and w in the allocator map α.

After the join, the locations entangled between children tasks v and w are no longer entangled because tasks v and w are no longer concurrent (they have joined). To reflect this in the program state, the Join rule remaps the accesses of the children tasks (v and w) to the parent task (u). In the semantics, the rule modifies the reader history and substitutes task u for tasks v and w. The modified reader history H' is equal to $H\{u/v\}\{u/w\}$.

Entanglement ceiling. Entanglement ceiling quantifies the maximum amount of memory reachable from entanglement sources at any step of an execution. Formally, suppose an execution starts with state S_0, proceeds as $S_0 \longrightarrow S_1 \ldots S_n$, and finishes with state S_n. Then, the entanglement ceiling, represented as δ, for this execution is $\max_i |\mu^+(\mathcal{E}(H_i))|$, where $\mathcal{E}(H_i)$ is the set of entanglement sources for history H_i and μ^+ computes memory reachable from them. Entanglement ceiling bounds the space cost incurred by our memory manager for managing entanglement, as the memory manager preserves all memory reachable from entanglement sources.

Other rules. Other rules are standard and we skip them for brevity. The evaluation begins from the state $(\emptyset\,;\emptyset\,;\emptyset\,;0\,;\mathsf{Leaf}(v)\,;e)$ where the allocator map α, the reader history H, and the memory μ are empty, the task tree is $\mathsf{Leaf}(v)$, the entanglement factor is 0, and e is the program. At the end of evaluation, the program state contains the result and also the entanglement factor of the execution.

3 MEMORY MANAGEMENT WITH ENTANGLEMENT

The memory manager for a parallel functional language must support high rates of allocation and scalable, parallel collection. Recent work proposed a new memory architecture for task-parallel programs that assigns each task its own heap and organizes the heaps into a dynamic tree that mirrors the dependencies between tasks [Westrick et al. 2020]. This architecture allows each processor to allocate and reclaim heaps independently without any synchronization, as long as the computation is "disentangled". Disentanglement disallows inter-heap pointers between heaps belonging to concurrent tasks, removing all memory dependencies between their heaps and enabling independent memory management. For programs with restricted effects that guarantee disentanglement, prior work demonstrates provable and practical efficiency [Arora et al. 2021]. However, unrestricted effects and entanglement present a challenge for this architecture, because they create inter-heap pointers without any limitation, creating thus dependence between concurrent tasks.

We tackle this challenge by tracking the memory objects affected by entanglement and shielding other objects and operations from entangled objects. We track ***entanglement sources***—the objects where entanglement begins and their ***entanglement regions***—the objects that become dependent on other concurrent tasks as a result of entanglement sources. To reclaim memory independently, we use algorithms that keep the objects of entanglement regions in place, but move and compact other objects. By identifying the entanglement region and keeping its objects in place, we resolve the dependence between concurrent tasks. For example, if a task decides to reclaim its heap, it does not relocate the entangled objects, allowing the other concurrent tasks to access the objects.

The reader may now be concerned that in an execution a large number of objects will become entangled. This turns out to be not the case, because entanglement is rare and temporary: entanglement is rare, because it results only due to non-deterministic races; entanglement is temporary because in fork-join task parallel programs any two concurrently executing tasks will eventually synchronize and a vast majority will synchronize quickly.

To identify entanglement regions, we use the ideas of our entanglement semantics, described in Section 2. Doing so correctly and efficiently poses several challenges. First, unlike the semantics, we must support simultaneous execution of tasks and consider that entanglement may be created concurrently by several tasks. Second, like the semantics, we must recognize the resolution of entanglement so as not to penalize computations more than necessary, but do so without maintaining any access records (reader history in the semantics) per object. Finally, we must mediate between the concurrent tasks and the collector, e.g., as a task is in the process of pinning the entanglement region of an object, the objects within the region may be undergoing relocation by the collector.

In the rest of this section, we present our techniques and describe how we overcome these challenges. Our techniques culminate in a theorem that bound the work (run-time) and space cost of tracking entanglement in terms of the cost metrics defined by the semantics. Before we dive into details we illustrate our data structures, give an overview, and consider a simple example.

3.1 Data Structures and Overview

Our memory manager partitions the memory into heaps and gives each task its own heap, which the task uses to allocate. The memory manager organizes the heaps as a ***heap tree*** that mirrors the task tree. In this tree, the internal heaps correspond to the suspended tasks (that are waiting for the children to finish) and the leaf heaps are used for new allocations by the active leaf tasks.

The memory manager maintains the heap tree data structure and also maintains, for each task, a ***source set*** for tracking entanglement sources, and a ***remembered set*** for tracking other roots. At a fork, it creates two empty heaps for the children tasks and initializes their source set and remembered set as empty sets. At a join, it merges the parent heap with the children heaps and merges the respective source and remembered sets of the children with those of the parent. [1]

Barriers and collector. To enable independent memory management, the barriers and the collector work together and account for pointers between independent heaps by tracking the entanglement that creates such pointers. The barriers intercept memory accesses and identify entanglement sources. When they identify a source, they add it to the source set of the task that allocated it. The collector treats the sources as roots and keeps them live.

The barriers and the collector also achieve a key performance goal: "barrier-less immutable accesses". Rather than intercepting immutable accesses to manage entanglement, the barriers ***pin*** the ***entanglement region*** of every source and the collector refrains from reclaiming or relocating pinned objects. The entanglement region spans all immutable objects that may be accessed via the source: it starts at the source (which may be mutable or immutable) and goes up to its "mutable frontier", i.e., it contains all objects reachable from the source by only considering immutable pointers. Here immutable pointers mean pointers of immutable objects. When a barrier identifies an entanglement source, it pins the source's entanglement region to ensure that the collector does not relocate or reclaim any object of the region. This way the barriers and the collector manage entangled immutable objects without intercepting immutable accesses.

Example. Consider the heap tree shown in Figure 3 with mutable objects depicted as squares and immutable objects as circles. The black pointers are either internal to a heap or are between objects of ancestor-descendant heaps whereas the red pointers are between objects of independent heaps. The objects at the end of red pointers are entanglement sources. The regions of the sources are represented with blue bubbles. One of the bubbles wraps only a single square demonstrating that if the source is mutable, its region only contains the source itself. Another bubble illustrates that the entanglement region does not extend beyond mutable objects, but may contain series of immutable objects (circles). As the figure shows, entanglement regions may overlap.

Expired entanglement sources. Entanglement sources increase space usage because the collector assumes that these sources are live. However, we can remove entanglement sources when two

[1]The merging of heaps step is a constant time operation and does not involve any copying of data.

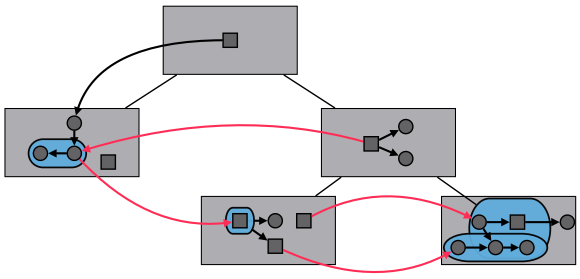

Fig. 3. The figure shows a heap tree with mutable and immutable objects represented as squares and circles respectively. The black pointers are between ancestor-descendant objects and the red pointers are between concurrently allocated objects. The targets of the the red pointers are the entangled roots that are added to the ent-set. Each entangled root has a entanglement region, which includes objects upto a mutable frontier. The entanglement regions are depicted using blue bubbles. The figure shows two entanglement regions that overlap and also shows an entangled region with a single mutable object.

tasks join and the entanglement between them is resolved, leading to the expiration of the sources entangled between them. By removing expired sources, we can remove the space cost of keeping them live. The challenge lies in tracking when sources expire. To do this, the barriers track an ***exp-depth*** (expiration depth) for each entanglement source. The exp-depth represents the depth at which all tasks that read the source through a mutable reference join. Once a source's heap reaches exp-depth in the heap tree, it is guaranteed that all the relevant joins have occurred and the source has expired. At this point, we can discard the source.

To avoid any bottlenecks and preserve parallelism, we discard expired sources lazily. When discarding a source, we must unpin its entanglement region and if we do so eagerly at joins, it would delay the joins and harm parallelism. We observe that keeping expired sources only impacts the garbage collector's efficiency, and therefore we filter expired sources at the time of collection.

Collection algorithms. The barriers guarantee that entangled objects are pinned. With this guarantee, we can collect any heap of the tree as long as the we respect pinning and not move pinned objects. We can use a variety of collection algorithms. In our design, we treat internal and leaf heaps differently. We collect internal heaps by adapting a concurrent mark-sweep algorithm for our heap hierarchy. For leaf heaps, we pause the corresponding leaf task and use a hybrid algorithm, an algorithm that does not relocate pinned objects because they are entangled, but copies and compacts other objects. The compaction aspect of this algorithm is crucial for our performance because it improves locality and defragments the heap. In fact, initially we tried a mark-sweep approach for the leaf heaps and observed performance hits of upto 20% in time and 75% in space.

3.2 Barriers

We use read and write barriers on mutable objects to track entanglement sources, entanglement regions, and their expiration depth. The barriers ensure the following invariants for each source:

(1) the source is in the source set of the task that allocated it,
(2) the source's entanglement region is pinned, and
(3) the source's exp-depth is correct, i.e., it expires once its heap reaches the exp-depth.

We give pseudocode for the barriers that shows how these invariants are maintained. In addition, the code considers the interaction between the collector and the barrier, e.g., it shows how a barrier may find a “forwarding pointer” on an object it is about to pin, because the collector has moved it. To ensure that a barrier is never blocked by a collection and vice versa, our code is lock-free.

```
1  procedure read ($\ell$, $t$):
2    $\ell'$ = !$\ell$
3    $t'$ = allocator($\ell'$)
4    if (concurrent($t, t'$)) {
5      (is_source, ed) = get_sinfo($\ell$)
6      if (is_source) {
7        set_sinfo($\ell'$)
8        return $\ell'$
9      }
10     else {
11       $\ell''$ = pin_region($\ell'$)
12       $e_{t'} \leftarrow e_{t'} \cup \{\ell''\}$
13       set_sinfo($\ell''$)
14       return $\ell''$
15     }
16   }
17   // fast path for disentangled reads
18   else return $\ell'$
19
20 procedure set_sinfo ($\ell$, $t$, $t'$):
21   sinfo = get_sinfo($\ell$)
22   (is_source, ed) = sinfo
23   if (!is_source)
24     ed = DCADepth ($t$, $t'$)
25   else
26     ed = min (DCADepth ($t$, $t'$), ed)
27   nsinfo = (true, ed)
28   if cas (sinfo_addr($\ell$), sinfo, nsinfo)
29     return
30   else set_source ($\ell$, $t$, $t'$)

31 procedure write ($\ell$, $\ell'$):
32   $t'$ = allocator($\ell'$)
33   if (allocator($\ell$) $\neq t'$)
34     $r_{t'} \leftarrow r_{t'} \cup \{(\ell, \ell')\}$
35   $\ell \leftarrow \ell'$
36   return
37
38 procedure pin_region ($\ell$):
39   $h$ = get_header ($\ell$)
40   if (forwarded($h$)) {
41     $\ell'$ = get_forward_object (h)
42     return pin_region ($\ell'$)
43   }
44   if (is_pinned($h$)) {
45     return $\ell$
46   }
47   if (mutable($h$)) {
48     if cas (addr_header($\ell$), h, pinned_version(h))
49       return $\ell$
50     else return pin_region ($\ell$)
51   }
52   else {
53     foreach $i \in$ ptr_fields($\ell$) {
54       $\ell'_i$ = pin_region($\ell[i]$)
55       if ($\ell'_i \neq \ell[i]$) {
56         $\ell[i] \leftarrow \ell'_i$
57       }
58     }
59     if cas(addr_header($\ell$), h, pinned_version(h))
60       return $\ell$
61     else return pin_region ($\ell$)
62   }
```

Fig. 4. Pseudocode read shows our read barrier. It uses pin_region and set_sinfo as helper functions to pin entanglement regions and maintain the exp-depth of entanglement sources respectively. The procedures account for concurrency with other tasks and also with the garbage collector. All procedures are lock-free.

3.2.1 Read barrier for mutable objects. Procedure read in Figure 4 shows the pseudocode for the read barrier. It takes as arguments the reader task t and an object ℓ and returns the result of the read, the object ℓ'. But first, it performs an entanglement check by comparing task t to the allocator task of object ℓ', task t'. The procedure allocator returns the task which allocated the object. If t' is concurrent to t (line 4), then the read is entangled, otherwise the read is disentangled. In the case of disentanglement, the barrier returns without further actions.

In case of entanglement, object ℓ' is an entanglement source and the read barrier acts to ensure the three invariants. It calls the procedure get_sinfo to check if the object is already flagged as a source. The procedure get_sinfo returns an object called sinfo, a tuple of two elements containing a boolean flag that tells if the object is a source and the exp-depth of the source. If the object is flagged as a source then the first two invariants are guaranteed and we only need to ensure the third invariant, i.e., ensure that the exp-depth is correct. The read barrier calls set_info, to update the exp-depth appropriately (discussed later in detail).

If the object is flagged as a source, the barrier (i) calls procedure pin_region to pin the entanglement region of ℓ', (ii) adds the object to the source set $e_{t'}$ of task t', and (iii) calls procedure set_sinfo to mark the object ℓ' as a source and set its exp-depth (see lines 11-13).

3.2.2 Entanglement region and its pinning. We implement pinning by reserving a bit in the header of each object, and when we set it we say that we ***pin*** the object. The procedure pin_region pins the objects in the entanglement region of its argument, accounting for the facts that that each object in the region may be relocated by the collector, or may simultaneously be pinned by other tasks. The procedure's pseudocode is shown by Figure 4.

Relocation by the collector. First, the procedure checks if its argument object has been relocated by the collector. When a collector relocates an object, it creates a copy of the object and installs a forwarding pointer to the copy in the header of the object. The procedure pin_region calls get_header to read the header of the argument object ℓ and calls forwarded to check if the header is a forwarding pointer (see Figure 4). If the header is a forwarding pointer, the procedure pin_region recursively calls itself on the copy of the object and returns (line 40). This recursive call pins the entanglement region of the object's copy and returns it.

When the header is not a forwarding pointer, the procedure pin_region checks if the header is pinned. If the header is pinned, it is guaranteed that the entanglement region of the object is also pinned (see "pin sharing" below). In this case, the procedure returns (line 45).

Mutable and immutable objects. In the case where the object is not pinned, the procedure pin_region pins the entanglement region of the object, a region that only includes the object itself when it is mutable, but goes up to the object's mutable frontier when it is immutable.

When the object is mutable, the procedure attempts to pin the object by calling cas (which stands for atomic compare-and-swap) on the header of the object (line 48). The procedure cas takes three arguments: the address of the header addr_header(ℓ), the expected version of the header h, and the pinned version of the header pinned_version(h). The cas updates the object's header if it finds the expected version h, returning true and otherwise performs no updates, returning false. If the cas fails, the procedure loops by calling itself.

When the object is immutable, the procedure must pin the entanglement region of the immutable object, consisting of all objects pointed by the immutable object and their respective entanglement regions. To do so, it recursively calls itself on each object pointed by the immutable object. Specifically, it calls procedure ptr_fields to obtain a sequence of pointers in the immutable object and recurses on the ith pointer $\ell[i]$, thereby pinning the entanglement region from $\ell[i]$ (line 54). However, this recursive call may return a copy of the argument object $\ell[i]$, if that object has been forwarded by the collector. In such a case, the procedure pin_region deletes the pointer from the immutable object to the relocated object and replaces it with a pointer to the copy (ℓ_i' in line 56). After it iterates through all the pointers, the procedure attempts to pin the immutable object. If it succeeds it returns the immutable object, and otherwise, it recurses.

Pin sharing. Multiple tasks may concurrently attempt to pin the same objects, raising the question if we can use the efforts of one to help others. To do so, tasks pin objects in a specific order. The order is determined by pointers of immutable objects: if an immutable object ℓ points to another object ℓ', then object ℓ' is pinned before object ℓ. This ordering guarantees that when an object is pinned all objects in its entanglement region are also pinned, because the region only contains immutable pointers from the object. Thus, if an object is pinned, we can return without pinning any further, see line 45 in Figure 4.

3.2.3 Expiration depth of entanglement sources. An object becomes an entanglement source when it is read by concurrent tasks through some mutable object. When all such concurrent tasks join with the allocator task of the source, the object ceases to be a source and becomes expired. We can determine when a source expires by tracking its depth in the heap tree. An entanglement source expires when its depth in the heap tree becomes shallower than its exp-depth. The exp-depth of a source can be calculated as follows:

Definition 3.1. The exp-depth of an entanglement source ℓ is equal to $\mathsf{DCADepth}(v_0, v_1 \ldots v_n, \alpha(\ell))$, where tasks $v_0 \ldots v_n$ read object ℓ from some mutable object and the function DCADepth calculates the depth of the deepest common ancestor of a set of tasks in the task tree.

The key insight is that when the readers $(v_0 \ldots v_n)$ and the allocator $(\alpha(\ell))$ of the entanglement source have joined, the depth of the entanglement source becomes equal to the depth of their deepest common ancestor. This depth is the exp-depth.

The procedure set_sinfo calculates the exp-depth in an online fashion. It takes the source ℓ, its allocator t, and its reader t' and sets the sinfo object of source ℓ. Recall that sinfo is a tuple which contains a flag that tracks whether the object is a source and also contains its exp-depth. We give the pseudocode of the procedure in Figure 4. The procedure first reads the sinfo of the object ℓ by calling get_sinfo and then checks if the object is already a source. If not, then the exp-depth of the source is equal to $\mathsf{DCADepth}(t, t')$. Otherwise, if the object is a source, it already has exp-depth ed, and the procedure picks the minimum among that depth and the depth $\mathsf{DCADepth}(t, t')$. To update the object's depth, it creates a new sinfo object named nsinfo, containing the correct exp-depth. Then, it attempts to install the nsinfo on the object using a cas and if it fails (because another task updated the sinfo), it tries again. In our implementation we flatten the sinfo tuple to a word.

3.2.4 Write barrier for mutable updates. The write barrier tracks inter-heap pointers created by mutable updates and adds such pointers to the remembered sets. We show its pseudocode in procedure write of Figure 4. It takes as arguments a mutable object ℓ and an object ℓ', and compares their allocator tasks to check if they are in different heaps (line 33). If so, it adds the tuple (ℓ, ℓ') to the remset $(r_{t'})$ of task t'. After this, it performs the update. The collector treats remembered set entries as roots, ignoring those that may have become internal as a result of joins.

3.3 Cost Analysis

We analyze the performance of our entanglement tracking algorithm by giving bounds on its work cost and space cost (in terms of liveness). Our proofs are in the Appendix.

Theorem 3.2 (Work bound). *An execution with work W requires $O(W + \epsilon)$ work to execute including the cost of entanglement tracking, where ϵ is the entanglement factor of the execution.*

Our entanglement tracking algorithm has three sources of overhead: identifying entanglement sources, pinning their entanglement region, and maintaining the exp-depth. The work bound shows that these overheads are localized to entangled objects, i.e., accesses to disentangled objects are not penalized. The bound also confirms that the penalty to entangled objects is additive: it is paid once per entangled object and does not increase with the number of times an entangled object is accessed. Our proof assumes that contention per object is low, i.e., we assume that each compare-and-swap (cas) instruction succeeds after a constant number of attempts.

Theorem 3.3 (Liveness bound). *If R is the maximum live memory of an execution of a parallel program, then maximum live memory of that execution, including the impact on liveness from entanglement tracking, is $O(R + \delta)$, where δ is the entanglement ceiling of that execution.*

Our entanglement tracking algorithm keeps entanglement sources live and increases the liveness of the execution for the garbage collector. The bound states that the algorithm only keeps $O(\delta)$ additional memory live, confirming that our entanglement tracking is precise: entanglement sources are kept live only for the duration they are entangled.

3.4 Garbage Collection

The collector collects internal heaps and leaf heaps with different algorithms. For internal heaps, the collector uses a snapshot-at-the-fork strategy: before a task forks and its heap becomes internal, the collector creates a snapshot of the heap's live objects. In this snapshot, the collector includes the source set and the remembered set of the heap. Later, at the time of collection, the collector uses this snapshot and keeps the reachable objects of the "snapshotted heap" in place and reclaims the unreachable objects of the heap; it does so with a concurrent mark-sweep algorithm. The collector deletes the snapshot, after a join, when the internal heap becomes a leaf again.

For leaf heaps, the collector uses a hybrid collection algorithm that does not move pinned objects (because they may be read by other tasks), but relocates unpinned objects to compact the heap. The algorithm runs in three parts. First, it prepares the roots by deleting expired entanglement sources from the source set and by deleting stale entries from the remembered set. Second, it traces the heap from these roots, keeping the pinned objects in place and relocating the others. Third, it frees the garbage objects, and also frees old objects that were relocated. In all three parts, the algorithm accounts for concurrent readers, consisting of other tasks that concurrently attempt to read and pin the objects of the heap. To guarantee that the collector never blocks a concurrent reader, the hybrid algorithm is lock-free. We discuss how it traces a heap and frees objects below. We leave the details of filtering entanglement sources and remembered set entries to the Appendix.

3.4.1 Tracing. To trace a heap, our hybrid algorithm maintains a ***mark*** bit in the object header and maintains a ***to-space*** for relocating the objects. Initially, every object is unmarked. By the end, the algorithm ensures that every pinned object is marked and every unpinned object is either garbage (unreachable) or has a forwarding pointer, pointing to the object's copy in the to-space.

The algorithm uses a stack of objects to trace the heap and begins by adding all the roots to the stack. Then, it iteratively pops an object from the stack and checks if the popped object is pinned. If so, it marks the object; otherwise, if the object is unpinned, it copies the object to the to-space and installs a forwarding pointer in the header of the object (unless the object is already in the to-space). After either marking the object or copying it, the algorithm adds the pointers within the object to the stack. This continues until the stack is empty.

```
procedure try_copy (ℓ):
 h = get_header (ℓ)
 if (is_pinned (h)) return ℓ
 ℓ' = copy_in_tospace(ℓ)
 if cas (addr_header(ℓ), h,
  fwd_ptr(ℓ'))
  return ℓ'
 else {
  delete_in_tospace (ℓ')
  return ℓ
  }
```

Fig. 5. procedure try_copy

To copy an unpinned object to the to-space, the collector must account for concurrent readers that may attempt to pin the object. Figure 5 presents the pseudocode that accounts for such readers. The procedure try_copy takes an object ℓ and attempts to copy it to the to-space. The procedure reads the header of the object and checks if the header is pinned. If the header is pinned, it returns the object without copying it. Otherwise, it copies the object and attempts to replace the object's header with a pointer to the object's copy. This pointer is called a forwarding pointer because it "forwards" the readers of the object to the object's copy. The procedure try_copy uses a cas operation (atomic compare and-swap) to install the forwarding pointer on the header. If the cas succeeds, the procedure returns the copy of the object. The cas operation can fail if a concurrent reader changes the header by pinning the object before the procedure replaces the header with a forwarding pointer. Because the object is pinned in this case, the procedure deletes the copy and returns the original object.

3.4.2 Freeing objects. By tracing the heap, the hybrid algorithm pins or relocates live objects to the to-space, leaving the garbage unpinned objects in the from-space. But, it cannot immediately free unpinned objects of the from-space as they may be accessed by concurrent tasks that have not detected that the algorithm has relocated them. Specifically, a read barrier issued by a concurrent task may attempt to pin an unpinned object of the from-space because it has not reached the point where it discovers the forwarding pointer. The algorithm can only free objects in the from-space after all barriers have crossed this "discovery point" and discovered the relocation. The read barrier discovers the relocation when it attempts to pin the object with a cas operation (compare-and-swap), but the operation fails because the collector has forwarded the object. (See line 11 in Figure 4: the location ℓ' may be in the from-space because the object has been relocated to location ℓ''. In this case, the procedure pin_region discovers the forwarding pointer to location ℓ'' after a cas fails.)

To ensure that all read barriers have crossed the discovery point, the hybrid algorithm employs epoch-based reclamation [Brown 2015]. [2] This technique *retires* from-space objects instead of freeing them, marking the objects with the current global *epoch*. It frees retired objects only after all the read barriers have crossed into a future epoch. Because epochs are incremented after read barriers cross the discovery point, all retired objects from the old epoch become safe to reclaim.

To amortize the work cost of retiring, our algorithm retires whole pages instead of individual objects. That is, the algorithm retires a page only if all objects within that page are garbage. As an optimization, the algorithm avoids the cost of retiring by directly freeing pages that are not susceptible to entanglement. (For example, if a garbage page is only reachable from the "local roots" of the collection, then the algorithm frees the page directly.)

After the algorithm retires the unpinned objects of the from-space, it traces the to-space to unmark the marked objects.

4 IMPLEMENTATION

To support parallel functional programming with effects, we extend the run-time system of the open-source [3] MPL (MaPLe) compiler for Parallel ML [Arora et al. 2021; Westrick et al. 2022, 2020]. MPL supports the entire Standard ML language and extends it with parallel tuples, i.e., fork-join parallelism. We extend the runtime system to support entanglement by implementing the memory management techniques described in Section 3. We also extend these techniques to (i) directly support operations such as compare-and-swap, (ii) remove all barriers on unboxed values, and (iii) reduce the overhead for mutable accesses, by only calling the read barrier on select mutable objects.

Reducing barrier overhead with entanglement frontier. Our entanglement tracking algorithm issues a read barrier on mutable reads, thereby triggering an entanglement check: when a task v reads a mutable location ℓ, the check performs a graph query that tests whether the read is entangled, i.e., if $\mathsf{concurrent}(v, \mathsf{allocator}(\ell))$. This check has non-zero overhead and slows the barrier down. We reduce the read barrier's overhead by reducing the frequency of these checks with a conservative heuristic.

We track an ***entanglement frontier*** which spans all mutable objects whose reads may cause entanglement. We only perform entanglement checks for frontier objects. When a frontier object is

[2]Epoch-based reclamation is a technique developed for lock-free data structures which face a similar problem: when a thread removes a node from a lock-free data structure, it is not safe to immediately free that node, because other threads may still access the node.
[3]https://github.com/MPLLang/mpl

read, the read barrier triggers an entanglement check as described in Section 3. In contrast, when a non-frontier object is read, the read barrier skips entanglement checking and takes a fast path, a path has no overhead except for the cost of testing whether the object is in the frontier. Testing this is cheap: we maintain a bit in the object header and set it for frontier objects. Since the object is already being read, testing this bit has near zero overhead.

Our barriers add two types of mutable objects to the frontier because they may create entanglement. First, if an object points to another object outside its heap, it allows a task to read a pointer into another task's heap and create entanglement. The write barrier detects when a task creates an inter-heap pointer from an object (to some other object) and adds the object to the entanglement frontier. Second, when an object is at the boundary of an entanglement region, it is entangled, and other (concurrent) tasks may read it, creating more entanglement. The read barrier adds such objects to the frontier when it pins an entanglement region.

Frontier objects become non-frontier after joins. Because joins merge heaps, they turn inter-heap pointers into internal pointers and void the first type of frontier objects. Furthermore, because joins expire entangled objects, they void the second type of frontier objects. To identify these changes to the frontier, the barriers associate a"join point" with every frontier object. When a frontier object reaches its join point, it is removed from the frontier. We reserve a bit in the header of mutable objects and toggle it to add/remove objects from the frontier.

5 EVALUATION

In this section, we evaluate and establish the practicality of our techniques. We compare our implementation, called MPL*, against multiple other language implementations (including both procedural and functional languages) and demonstrate its efficiency and scalability.

Benchmarks. Our evaluation considers two different benchmarking suites, totaling 48 benchmarks across five different languages.

The first suite includes 26 different benchmarks written in MPL*. Half (thirteen) of these benchmarks are highly concurrent and entangled. To implement these benchmarks, we first implemented a number of non-blocking concurrent data structures, including Harris's non-blocking list [Harris 2001], the Lindén-Jonsson priority queue [Lindén and Jonsson 2013], the Michael-Scott non-blocking queue [Michael and Scott 1996], Kumar et al.'s persistent arrays [Kumar et al. 2017], and many others. Using these, we then implemented sophisticated parallel algorithms for

- quantum circuit synthesis,
- delaunay triangulation,
- various graph analyses, including reachability/connectivity, $O(k)$-spanner, low-diameter decomposition boundaries, etc., and
- deduplication via concurrent hashing.

In addition, we implemented synthetic benchmarks that operate on concurrent data structures by mixing parallelizable work with updates and queries on the shared data structure(s).

Some of our benchmarks—such as the quantum synthesis and delaunay triangulation—are complex and have taken multiple person-months of work (each) to implement. The benchmarks are heavily concurrent; for example, the quantum circuit synthesizer implements a parallel version of the Solovay-Kitaev algorithm [Dawson and Nielsen 2006; Kitaev et al. 2002] and uses both concurrent hash tables and concurrent lists. Other benchmarks are from various problem domains, such as graph analysis, text processing, digital audio processing, image analysis and manipulation, numerical algorithms, computational geometry, and others; some are from prior work [Westrick et al. 2022].

Our second benchmarking suite contains the implementations of eight benchmarks in four languages (other than MPL*) : C++, Go, Java, and Multicore OCaml. The C++ benchmarks come from PBBS [Anderson et al. 2022; Shun et al. 2012] and ParlayLib [Blelloch et al. 2020]. We ported these to Go, Java, and OCaml, while re-using existing Java implementations of two benchmarks. We selected these benchmarks for diversity (covering both disentanglement and entanglement, as well as both memory- and compute-intensive benchmarks), and for ease of implementation, as it takes significant work to implement each benchmark in multiple languages.

Methodology. To measure timings, we first do a warmup by running the benchmark back-to-back for at least 5 seconds, and then do 20 back-to-back runs. All of this happens in the same program instance. The number reported is the average of the 20 runs, and the warmup is disregarded.

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. Back-to-back runs are executed in the same program instance to ensure that any effects of memory management amortization thresholds are taken into account (for example, a garbage collection might run only once every five runs). We use the maximum resident set size measurement because it takes into account all potential sources of space usage, including allocation freelists, GC metadata, etc., thus allowing for comparison across systems with different memory management strategies.

We write T_P for time on P processors, and similarly R_P for the max residency on P processors. Unless stated otherwise, all times are in seconds and all space numbers (max residencies) are in GB.

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, 1TB of memory, and running Ubuntu version `16.04.7` with Linux version `4.10.0-40-generic x86_64`. In Section 5.1, we use MLton version `20210117`. In Section 5.2, we use MPL version `0.3`. For cross-language comparisons (Section 5.5), we use the following systems: multicore OCaml version `5.0.0+dev4-2022-06-14` with default settings and the library `domainslib` version `0.4.2`; Go version `1.18.4`; g++ version `10.3.0` with the `jemalloc` library, and the compiler flags `-O3`, `-march=native`, `-std=c++17`, and `-mcx16`; Java OpenJDK version `11.0.14`. For Java, we used the `G1GC` collector (which we found yielded the best performance) and controlled the number of threads with the setting `-XX:ActiveProcessorCount=`N.

5.1 Overheads and Scalability

In this section, we compare against the MLton [MLton] compiler, which is a compiler for (sequential) Standard ML. Our MPL* extends MLton with support for parallelism; there are minimal compilation differences between the two systems. (For MLton, we compile the sequential elision of each benchmark.) This comparison therefore allows us to determine the overheads and scalability of our memory manager by using executables produced by MLton as a sequential baseline. We also measure here the entanglement factor ϵ of each benchmark.

The results of the comparison are shown in Figure 6. The column T_s is the sequential baseline time, using MLton. The columns T_1 and T_{72} are the times of our MPL* on 1 and 72 processors, respectively. Similarly, R_s is the maximum residency of the sequential baseline, and R_1 and R_{72} are the single-processor and 72-processor residencies of MPL*. The overheads T_1/T_s indicates the performance of MPL* on 1 processor relative to the sequential baseline; smaller numbers are favorable. Speedups on 72 processors in comparison to the sequential baseline are calculated as T_s/T_{72}, where larger numbers are favorable. To compare space usage, we compute the space blowup on p processors as R_p/R_s. This number indicates how much more memory MPL* uses in comparison to the sequential baseline. Larger blowups indicate more space usage.

We first observe that, on average, MPL* achieves 31x speedup on 72 processors over MLton. These speedups range from 13-49x because some of these benchmarks, like `interval-tree`, `quant-synth`,

	Time (s)					Space (GB)					Bytes Entangled
			OV		SU			BU$_1$		BU$_{72}$	
Benchmark	T_s	T_1	$\frac{T_1}{T_s}$	T_{72}	$\frac{T_s}{T_{72}}$	R_s	R_1	$\frac{R_1}{R_s}$	R_{72}	$\frac{R_{72}}{R_s}$	ϵ_{72}
centrality	14.7	20.9	**1.42**	.466	**32**	33	6.6	**.20**	5.8	**.18**	0
delaunay	8.65	17.1	**1.98**	.667	**13**	2.7	1.7	**.63**	11	**4.1**	9.5 M
find-influencers	14.5	15.5	**1.07**	.433	**33**	35	7.4	**.21**	5.8	**.17**	.23 M
grep	1.43	2.16	**1.51**	.041	**35**	4.6	.61	**.13**	.85	**.18**	0
harris-linked-list	5.84	5.97	**1.02**	.163	**36**	.034	.021	**.62**	.094	**2.8**	3.3 M
hash-dedup	2.46	3.79	**1.54**	.081	**30**	6.8	.92	**.14**	1.3	**.19**	.56 M
interval-tree	2.89	4.30	**1.49**	.067	**43**	.53	.12	**.23**	.64	**1.2**	0
ldd-boundary	22.3	30.8	**1.38**	.705	**32**	31	7.0	**.23**	17	**.55**	.12 G
linden-pq	5.84	6.98	**1.20**	.193	**30**	.14	.11	**.79**	.15	**1.1**	9.3 M
linefit	3.01	2.34	**0.78**	.149	**20**	8.5	8.2	**.96**	8.3	**.98**	.16 M
max-indep-set	13.2	16.3	**1.23**	.379	**35**	27	7.0	**.26**	5.7	**.21**	0
mcss	1.88	4.74	**2.52**	.080	**23**	4.3	4.1	**.95**	4.1	**.95**	0
ms-queue	5.95	7.94	**1.33**	.303	**20**	.27	.18	**.67**	3.3	**12.2**	80 M
msort-int64	3.30	4.45	**1.35**	.085	**39**	5.0	.66	**.13**	.93	**.19**	0
nearest-nbrs	1.32	1.69	**1.28**	.039	**34**	1.5	.98	**.65**	2.1	**1.4**	0
persistent-arr	4.61	7.60	**1.65**	.109	**42**	.65	.60	**.92**	1.5	**2.3**	3.6 K
quant-synth	12.9	17.2	**1.33**	.319	**40**	4.6	.99	**.22**	12	**2.6**	1.8 M
quickhull	2.50	3.59	**1.44**	.115	**22**	15	18	**1.2**	20	**1.3**	0
range-query	14.1	16.2	**1.15**	.287	**49**	13	4.5	**.35**	4.3	**.33**	0
reachability	11.3	14.7	**1.30**	.412	**27**	37	5.9	**.16**	15	**.41**	82 K
reverb	1.01	1.36	**1.35**	.043	**23**	6.9	1.5	**.22**	2.0	**.29**	0
seam-carve	7.58	7.95	**1.05**	.280	**27**	1.2	.13	**.11**	.33	**.28**	0
spanner	11.2	16.9	**1.51**	.421	**27**	34	8.8	**.26**	46	**1.4**	5.9 M
tokens	1.55	1.89	**1.22**	.042	**37**	13	1.1	**.08**	1.2	**.09**	0
triangle-count	4.67	5.18	**1.11**	.110	**42**	2.7	.76	**.28**	1.2	**.44**	0
wc	5.18	7.44	**1.44**	.120	**43**	1.9	3.6	**1.9**	3.6	**1.9**	.28 M
geomean			**1.34**		**31**			**0.34**		**0.68**	

Fig. 6. Comparison with sequential baseline: times, max residencies, overheads (OV), speedups (SU), space blowups (BU), and entanglement factors (ϵ).

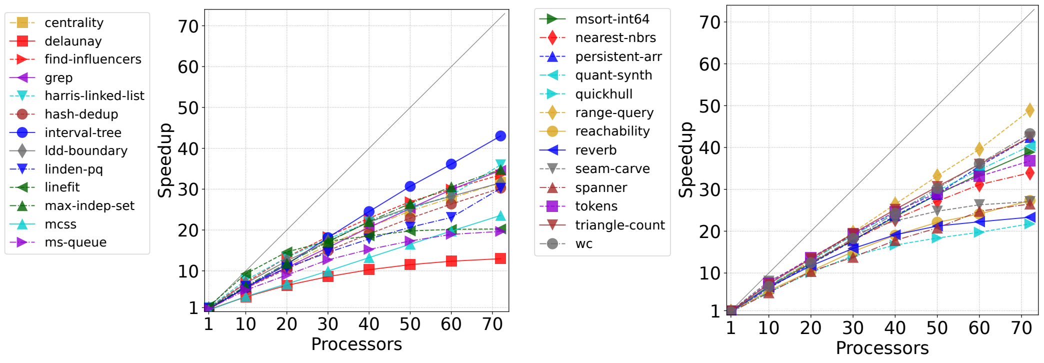

Fig. 7. Speedups

and `triangle-count` are compute bound and see higher speedups, whereas others like `seam-carve`, `delaunay`, and `reverb` are memory bound. The Figure 7, where we plot the speedup of MPL* relative to MLton, also shows this trend. Some benchmarks plateau in speedup before 72 processors (e.g., delaunay, seam-carve, linefit). This is expected, as these benchmarks are memory-bound. All other benchmarks scale approximately linearly with the number of processors.

Across the board, we also see that MPL* almost always uses less memory than MLton (Figure 6). On average, MPL* uses approximately 30% less memory on 72 processors than MLton does sequentially. This is because MPL* has a more aggressive GC policy than MLton; in anticipation of high memory usage for parallel applications, MPL* collects aggressively. One could make MLton just as aggressive by adjusting MLton's default GC parameters. In the Appendix, we compare with aggressive settings for MLton that force it to spend more time on GC and use less space. Even under this setting, MPL*'s memory blowup on 72 processors increases to only 2x on average.

For reference, we report entanglement factors ϵ_{72}, which are measured in terms of the number of bytes pinned by the entanglement manager on 72 processors. We observe that entanglement factors are low in practice. Approximately half of the benchmarks considered here are naturally disentangled, and therefore have an entanglement factor of 0. Amongst the entangled benchmarks, we can see that the number of bytes pinned is generally small, especially in comparison to the maximum resident memory needed for 72 processor execution. For example, ldd-boundary has ϵ_{72} = .12GB, which is less than 1% of its resident memory R_{72} = 17GB.

Altogether, these results demonstrate that MPL* is able to achieve significant speedups while controlling space usage effectively.

5.2 Disentanglement is Not Penalized

To show that our entanglement management techniques do not penalize disentanglement, we compare our MPL* against MPL (version 0.3). MPL is specialized for disentangled programs, and cannot run entangled programs. For this comparison, we include all disentangled benchmarks from Figure 6 as well as multiple other programs from the Parallel ML Benchmark Suite, for a total of 30 benchmarks. Across these benchmarks, we find that MPL* generally has very similar performance to MPL, with no time overhead on average, and less than 4% space overhead on average. (Full data available in the Appendix.) These results show that MPL* is able to take advantage of disentanglement just as well MPL, while being significantly more general: that is, MPL* can safely execute any program, not just disentangled programs. The results are also a sanity check for our theory (Theorem 3.2), which bounds the overhead of tracking entanglement to $O(\epsilon)$, where ϵ is the amount of entanglement. Because these benchmarks are disentangled, we expect to see 0 overhead.

5.3 Entanglement Management Overhead

To further investigate the cost of managing entanglement, we developed a stress test for which we can control the amount of entanglement. The total program work, live memory, and parallelism of this benchmark all stay approximately the same regardless of the amount of entanglement introduced. On this stress test, we varied the amount of entanglement and measured the overhead on 72 processors. Specifically, if the benchmark is x% entangled, then overhead is the ratio $T(x)/T(0)$, where $T(x)$ is the running time with x% entanglement. The results of this stress test are shown in Figure 8. First, we observe that the overhead increases linearly with the amount of entanglement. It is 1 (i.e., no overhead) at 0% entanglement and grows upto a factor of approximately 3 near 100% entanglement. This agrees with our theory: the work of entanglement tracking increases

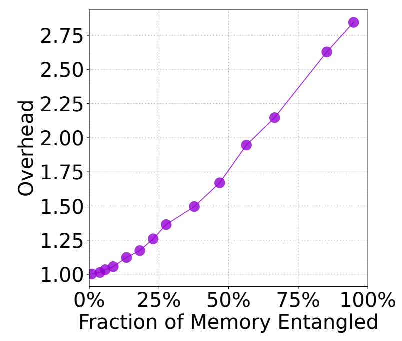

Fig. 8. Stress test for entanglement management overhead

Time (s) on 72 processors:

	C/C++		MPL* (Ours)		Go		Java		OCaml	
	C	$\frac{C}{M}$	M	$\frac{M}{M}$	G	$\frac{G}{M}$	J	$\frac{J}{M}$	O	$\frac{O}{M}$
hash-dedup	.043	**0.53**	.081	**1.00**	.127	**1.57**	.052	**0.64**	.174	**2.15**
linefit	.150	**1.01**	.149	**1.00**	.157	**1.05**	.405	**2.72**	1.18	**7.92**
mcss	.038	**0.47**	.080	**1.00**	.105	**1.31**	.507	**6.34**	.588	**7.35**
msort-int64	.058	**0.68**	.085	**1.00**	.280	**3.29**	.242	**2.85**	.489	**5.75**
primes	.072	**0.58**	.124	**1.00**	.189	**1.52**	.211	**1.70**	.166	**1.34**
sparse-mxv	.050	**1.04**	.048	**1.00**	.092	**1.92**	.089	**1.85**	.727	**15.15**
tokens	.020	**0.48**	.042	**1.00**	.484	**11.52**	.256	**6.10**	.947	**22.55**
wc	.058	**0.48**	.120	**1.00**	.171	**1.43**	.052	**0.43**	.910	**7.58**
geomean		**0.63**		**1.00**		**2.07**		**2.00**		**6.30**

Fig. 9. MPL* vs C++, Java, Go, and OCaml: time (seconds) on 72 processors. The time ratios are relative to MPL* and show how fast it runs w.r.t. other languages (larger ratios favor MPL*). The geomeans show the average of these ratios.

Space (GB) on 72 processors:

	C/C++		MPL* (Ours)		Go		Java		OCaml	
	C	$\frac{C}{M}$	M	$\frac{M}{M}$	G	$\frac{G}{M}$	J	$\frac{J}{M}$	O	$\frac{O}{M}$
hash-dedup	1.40	**1.08**	1.30	**1.00**	1.10	**0.85**	7.70	**5.92**	1.10	**0.85**
linefit	8.80	**1.06**	8.30	**1.00**	8.40	**1.01**	23.0	**2.77**	33.0	**3.98**
mcss	4.80	**1.17**	4.10	**1.00**	4.90	**1.20**	27.0	**6.59**	4.20	**1.02**
msort-int64	1.40	**1.51**	.930	**1.00**	1.70	**1.83**	2.40	**2.58**	6.80	**7.31**
primes	.920	**2.30**	.400	**1.00**	.470	**1.17**	1.90	**4.75**	1.80	**4.50**
sparse-mxv	4.90	**1.09**	4.50	**1.00**	6.30	**1.40**	17.0	**3.78**	7.30	**1.62**
tokens	1.70	**1.42**	1.20	**1.00**	4.60	**3.83**	15.0	**12.50**	17.0	**14.17**
wc	2.50	**0.69**	3.60	**1.00**	3.60	**1.00**	5.60	**1.56**	1.90	**0.53**
geomean		**1.22**		**1.00**		**1.36**		**4.20**		**2.47**

Fig. 10. MPL* vs C++, Java, Go, and OCaml: space (GB) on 72 processors. The space ratios are relative to MPL* and show the proportion of memory MPL* saves, w.r.t. other languages. (larger ratios favor MPL*). The geomeans average these ratios.

linearly with entanglement (Theorem 3.2). Furthermore, the constant factor hidden in the work bound $O(W + \epsilon)$ is relatively small (approximately 3 in this case)

5.4 GC Performance

We collected a number of statistics on GC performance and summarize the data here (the full table is in the Appendix). We observed that the total amount of time spent on GC is generally low, with about half of the benchmarks on 72 processors spending less than 1% work on GC, with a maximum of 25%. This indicates that GC work is well-amortized against program work. We also observe that in almost all benchmarks (22 out of 26), the average time for a leaf GC is less than a millisecond. Furthermore, the leaf GCs reclaim approximately three times as much memory than internal GCs. This is because (i) leaf GCs can happen simultaneously and independently on P processors, and (ii) leaf heaps largely contain young allocations and leaf GC benefits from the generational hypothesis

5.5 Cross-Language Comparisons

In this section, we compare MPL* with four other languages, C/C++, Go, Java, and multicore OCaml, all of which support nested fork-join parallelism in the same style as MPL*. In Go, we use goroutines and channels to implement nested fork-join parallelism. In Java, we use parallel streams and the Java Fork/Join framework. (Note also that we account for Java warmup in our benchmarking methodology, discussed at the top of Section 5.) In OCaml, we use parallel primitives from the `domainslib` library.

We compare these languages on eight benchmarks, which come from the PBBS benchmark suite [Anderson et al. 2022; Shun et al. 2012] and the ParlayLib [Blelloch et al. 2020] library of parallel algorithms and data structures. We port these benchmarks from C/C++ into each of Parallel ML, Go, Java, and OCaml, preserving the underlying algorithms. For example, an array of structs in C/C++ is represented as (i) an array of tuples in Parallel ML and OCaml, (ii) an array of structs in Go, and (iii) an array of objects in Java. Where possible, we use external codes implemented by experts. In Java specifically, we use the `ConcurrentHashMap` and `parallelSort` implementations from the `java.util.concurrent` and `java.util.Arrays` libraries, respectively, for the `hash-dedup` and `msort-int64` benchmarks. Three benchmarks use lock-free data structures and have entanglement: `hash-dedup`, `linefit`, and `wc`.

Results. The results of this comparison are shown in Figures 9 and 10. Figure 9 shows the time of each benchmark, as well as the ratio relative to our MPL*; Figure 10 similarly reports space of each benchmark. Here, the abbreviations are **C** for C/C++, **M** for our MPL*, **G** for Go, **J** for Java, and **O** for OCaml. For example, in the time figure, the column **J** is the running time (in seconds) for Java, and next to it, the column **J/M** is the ratio of Java's running time to MPL*'s running time. Higher ratios are favorable for MPL*.

The results show that among memory-managed languages, MPL* is faster and more space-efficient on all but two benchmarks. In particular, on benchmarks `hash-dedup` and `wc`, Java runs faster than MPL* but consumes more space; on the same benchmarks, both Go and OCaml consume less space than MPL*, but run slower than it. On all other benchmarks MPL* performs better in terms of both speed and space consumption. On the `tokens` benchmark, we observe a large performance gap between MPL* and the other languages, where MPL* is significantly faster and also consumes less space. This performance difference is due to the memory layout of a key data structure—specifically, an array with tuples for elements. MPL* is able to "unbox" the tuples and flatten them into the array; note that MPL* is based on MLton which performs optimizations such as data flattening. In Java, each element of the array is "boxed" and therefore incurs additional allocations. We confirmed that by pre-allocating the elements of the array and excluding this cost, the runtime of Java improves by approximately 2x. We similarly investigated the performance of Go and OCaml on this benchmark, and observed similar behavior.

We also compare against the C/C++ versions of these benchmarks, which (for our purposes here) can be considered the "performance goal". For time performance, MPL* is generally within a factor of 2 of C/C++, and in two cases matches C/C++ (`linefit` and `sparse-mxv`). We observe that MPL* is also competitive in terms of space, with approximately 22% less space on average. This space difference is due to additional space consumed by the allocator used in the C/C++ benchmarks. In particular, the C/C++ benchmarks use an allocator provided by ParlayLib [Blelloch et al. 2020], which is primarily optimized for running time performance and incurs some space overhead to support fast parallel allocation. With the help of the ParlayLib authors, we were able replace the allocator with `jemalloc` [Evans 2006] for some benchmarks. In particular, we were able to rerun `hash-dedup`, `msort-int64`, and `tokens` benchmarks using `jemalloc`, and observed that on these benchmarks, `jemalloc` yields similar time performance while using less space. (For these benchmarks, `jemalloc` uses 0.75x space relative to MPL* on average; in comparison, the default ParlayLib allocator uses 1.4x space relative to MPL* on this subset of benchmarks.) However, we also observed that some benchmarks crash with `jemalloc`, and we could not determine the source of the bug. We have relayed these findings to the ParlayLib authors.

Overall, our conclusion from these results is that MPL* is generally fast, scalable, and space-efficient. Our experiments suggest that parallel functional programming can deliver the same performance and scalability as procedural and imperative languages.

6 RELATED WORK

Advances on multicore chips have motivated the design and implementation of parallel programming languages or extensions to existing programming languages. A variety of programming languages and libraries have converged on task parallelism, a form of parallelism that allows programmers to express parallelism at a high level of abstraction, leaving task creation and scheduling decisions to the run-time system and a task scheduler [Acar et al. 2002, 2018, 2016a; Blelloch et al. 1999; Muller and Acar 2016]. C-based task-parallel systems include Cilk/Cilk++ [Blumofe et al. 1995; Frigo et al. 2009; Intel Corporation 2009a], Cilk-F/L [Singer et al. 2020, 2019], I-Cilk [Muller et al. 2020], Grace [Berger et al. 2009], and Intel's TBB [Intel Corporation 2009b]. Deterministic Parallel Java [Bocchino, Jr. et al. 2009], Fork-Join Java [Lea 2000], and Habanero Java [Imam and Sarkar 2014], extend the memory-managed Java language to support task parallelism. The memory-managed X10 [Charles et al. 2005] was designed from grounds up with concurrency and parallelism in mind and supports async-finish style of task parallelism. The Rust language supports concurrency and parallelism and can ensure memory safety without using a garbage collector [Rust Team 2019]. The memory-managed Go language support task parallelism with go-routines. Parallel functional languages extend Haskell [Chakravarty et al. 2007; Keller et al. 2010; Koparkar et al. 2021; Marlow and Jones 2011] and the ML family, including Manticore [Fluet et al. 2008, 2011], MPL [Arora et al. 2021; Westrick et al. 2022, 2020], MultiMLton [Sivaramakrishnan et al. 2014; Ziarek et al. 2011], multicore OCaml [Sivaramakrishnan et al. 2020], and SML# [Ohori et al. 2018]. Many of the aforementioned languages are garbage collected and employ a variety of memory management techniques.

Our work builds on the prior work on disentanglement and the MPL project. Disentanglement was initially developed for purely functional programs [Raghunathan et al. 2016], and then extended to include all determinacy-race free programs [Westrick et al. 2020]. More recently, techniques were proposed to prevent undefined behavior in case of entanglement by safely terminating the program when entanglement occurs [Westrick et al. 2022]. Other work presented a provably and practically efficient memory manager for determinacy-race free programs [Arora et al. 2021]. This was achieved by introducing a "heap scheduler" that takes decisions about when and where (on what processor) a heap should be collected. By assuming determinacy-race-freedom, the authors are able to use sequential execution as the "baseline" for their bounds. We expect the techniques presented in this paper can enjoy work and space bounds similar to those of Arora et al [Arora et al. 2021], but establishing such bounds seems challenging because of the difficulties of defining a sequential baseline in the presence of arbitrary mutation (memory effects) and non-determinism.

The idea of organizing the memory of a parallel programs as a hierarchy of heaps goes back to 1990s [Alpern et al. 1995] and has since been explored by different researchers [Bauer et al. 2012; Krishnamurthy et al. 1993; Numrich and Reid 1998; Yelick et al. 1998]. MPL's approach differs from prior work because in MPL hierarchical heaps are fully dynamic and are managed (created, maintained, destroyed, and collected) without programmer intervention.

Other parallel functional languages—such as Haskell, Manticore, and Multicore OCaml—employ a two-level memory hierarchy, consisting of a private heap for each processor, and a shared heap. They rely on promotions of shared objects or stop-the-world collections [Auhagen et al. 2011; Le and Fluet 2015; Marlow and Jones 2011; Sivaramakrishnan et al. 2020]. Promotions require copying reachable objects from a shared pointer and can be triggered by scheduler actions, making it difficult to prove bounds on their run time and space impact. Indeed, promotions have proved to be expensive in practice [Guatto et al. 2018; Sivaramakrishnan et al. 2020]. Blelloch and Cheng [Blelloch and Cheng 1999] present provably efficient memory managers for real-time garbage collection for a parallel dialect of ML. Their techniques rely on replication to ensure concurrency, which can incur significant practical overheads.

One of our key contributions is to support unrestricted mutation in the hierarchical heap architecture, and do so without breaking the key advantages of the architecture: independent allocation, reclamation and promotion-free sharing. We propose techniques for dynamically tracking "entanglements" that are created by cross-pointers between concurrent heaps. For garbage collection, we use both concurrent and a hybrid collector. The concurrent collector never moves objects and is similar to standard mark-sweep collectors [Jones et al. 2012] except that it only collects a single heap (region) of memory. The hybrid collector moves non-entangled objects and leaves the entangled objects in place. To accomplish this, our hybrid collector uses techniques that are similar to the "mostly copying" collector of Bartlett [Bartlett 1988, 1989] and the Customizable Memory Manager of Attardi et al [Attardi and Flagella 1994; Attardi et al. 1998], which was designed as a garbage collector for C++ programs. The basic approach is to track "quasi" or "ambigious" pointers, which may be pointers to objects and prevent these objects from moving during garbage collection. Hosking [Hosking 2006] also presents a mostly copying algorithm; this work allows for concurrency but relies on a stop-the-world phase to collect roots. Our approach applies ideas from these to hierarchical heaps and entanglement but differs in the specific barriers used and specific ways of handling pinned objects. These culminate in techniques that allow threads to allocate, access, and reclaim memory independently without using locks or stop-the-world pauses.

In this paper, we consider fork-join programming model, which is a good fit for compute intensive applications but makes it difficult if not impossible to express others such as, interactive parallel applications, for which futures remain a better fit [twi 2011; fac 2015; Acar et al. 2016b; Halstead 1984; Muller et al. 2020; Muller and Acar 2016; Muller et al. 2017, 2018a,b]. It would be interesting to extend these techniques to more general models of computation including futures.

7 CONCLUSION

Prior research proposed memory management techniques that meet the pressing memory demands of parallel functional programs, but only if they do not create entanglement. In this paper, we generalize the proposed techniques to support all programs by treating entanglement as a continuous, object-level property. Specifically, we distinguish between two types of objects: disentangled objects, which constitute a majority of objects, and entangled objects, which are typically a minority but present a significant obstacle to efficient memory management. We develop algorithms that identify, track, and manage entangled objects to mitigate their impact, allowing for effective memory management of the majority objects (i.e., of disentangled objects). Our implementation and experiments demonstrate the effectiveness of this approach, establishing that functional languages can deliver both performance and scalability, while offering important safety benefits.

ACKNOWLEDGMENTS

We thank the anonymous PLDI reviewers and our shepherd, KC Sivaramakrishnan, for their feedback. This work was partially supported by the National Science Foundation under grant numbers CCF-1901381, CCF-2115104, CCF-2119352, and CCF-2107241.

8 DATA AVAILABILITY

All of our implementation and data are recorded in the artifact [Arora et al. 2023]. The artifact is published on Zenodo and it contains the code for our language MPL* and the code for all our benchmarks. We leave further instructions to the document on Zenodo.

REFERENCES

2011. Finagle: A Protocol-Agnostic RPC System. https://twitter.github.io/finagle/.

2015. Folly: Facebook Open-source Library. https://github.com/facebook/folly.

Umut A. Acar, Naama Ben-David, and Mike Rainey. 2017. Contention in Structured Concurrency: Provably Efficient Dynamic Non-Zero Indicators for Nested Parallelism. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Austin, Texas, USA) *(PPoPP '17)*. 75–88.

Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. 2002. The Data Locality of Work Stealing. *Theory of Computing Systems* 35, 3 (2002), 321–347.

Umut A. Acar, Arthur Charguéraud, Adrien Guatto, Mike Rainey, and Filip Sieczkowski. 2018. Heartbeat Scheduling: Provable Efficiency for Nested Parallelism. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. 769–782.

Umut A. Acar, Arthur Charguéraud, and Mike Rainey. 2016a. Oracle-guided scheduling for controlling granularity in implicitly parallel languages. *Journal of Functional Programming (JFP)* 26 (2016), e23.

Umut A. Acar, Arthur Charguéraud, Mike Rainey, and Filip Sieczkowski. 2016b. Dag-calculus: A Calculus for Parallel Computation. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP 2016)*. 18–32.

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

Bowen Alpern, Larry Carter, and Jeanne Ferrante. 1995. Space-limited procedures: A methodology for portable high-performance. In *Programming Models for Massively Parallel Computers*. IEEE, 10–17.

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

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)"*.

Jatin Arora, Sam Westrick, and Umut A. Acar. 2023. *Replication instructions for Article: Efficient Parallel Functional Programming with Effects*. https://doi.org/10.5281/zenodo.7824069

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.

Giuseppe Attardi and Tito Flagella. 1994. *A Customisable Memory Management Framework*. Technical Report TR-94-010. International Computer Science Institute, Berkeley. ftp://ftp.icsi.berkeley.edu:/pub/techreports/1994/tr-94-010.ps.Z Also Proceedings of the USENIX C++ Conference, Cambridge, MA, 1994.

Giuseppe Attardi, Tito Flagella, and Pietro Iglio. 1998. A Customisable Memory Management Framework for C++. *Software Prac. Experience* 28, 11 (Nov. 1998), 1143–1183. ftp://ftp.di.unipi.it/pub/Papers/attardi/SPE.ps.gz

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.

Joel F. Bartlett. 1988. *Compacting Garbage Collection with Ambiguous Roots*. Technical Report 88/2. DEC Western Research Laboratory, Palo Alto, CA. http://www.research.digital.com/wrl/techreports/88.2.ps Also in Lisp Pointers 1, 6 (April–June 1988), 2–12.

Joel F. Bartlett. 1989. *Mostly-Copying Garbage Collection picks up Generations and C++*. Technical Note. DEC Western Research Laboratory, Palo Alto, CA. ftp://ftp.digital.com/pub/DEC/WRL/research-reports/WRL-TN-12.ps Sources available in ftp://ftp.digital.com/pub/DEC/CCgc.

M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. 2012. Legion: Expressing locality and independence with logical regions. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–11. https://doi.org/10.1109/SC.2012.71

Emery D. Berger, Ting Yang, Tongping Liu, and Gene Novark. 2009. Grace: Safe Multithreaded Programming for C/C++. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications* (Orlando, Florida, USA) *(OOPSLA '09)*. Association for Computing Machinery, New York, NY, USA, 81–96. https://doi.org/10.1145/1640089.1640096

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, Phillip B. Gibbons, and Yossi Matias. 1999. Provably efficient scheduling for languages with fine-grained parallelism. *J. ACM* 46 (March 1999), 281–321. Issue 2.

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 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.*

Trevor Alexander Brown. 2015. Reclaiming Memory for Lock-Free Data Structures: There Has to Be a Better Way. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing* (Donostia-San Sebastián, Spain) *(PODC '15)*. Association for Computing Machinery, New York, NY, USA, 261–270. https://doi.org/10.1145/2767386.2767436

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.

Alexander Collins, Dominik Grewe, Vinod Grover, Sean Lee, and Adriana Susnea. 2014. NOVA: A Functional Language for Data Parallelism. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming* (Edinburgh, United Kingdom) *(ARRAY'14)*. Association for Computing Machinery, New York, NY, USA, 8–13. https://doi.org/10.1145/2627373.2627375

Christopher M. Dawson and Michael A. Nielsen. 2006. The Solovay-Kitaev algorithm. *Quantum Inf. Comput.* 6, 1 (2006), 81–95. https://doi.org/10.26421/QIC6.1-6

Jason Evans. 2006. A Scalable Concurrent malloc(3) Implementation for FreeBSD. (01 2006).

Yotam M. Y. Feldman, Artem Khyzha, Constantin Enea, Adam Morrison, Aleksandar Nanevski, Noam Rinetzky, and Sharon Shoham. 2020. Proving Highly-Concurrent Traversals Correct. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 128 (nov 2020), 29 pages. https://doi.org/10.1145/3428196

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.

Keir Fraser. 2003. Practical lock-freedom.

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.

Vincent Gramoli. 2015. More than You Ever Wanted to Know about Synchronization: Synchrobench, Measuring the Impact of the Synchronization on Concurrent Algorithms. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Francisco, CA, USA) *(PPoPP 2015)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/2688500.2688501

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.

Timothy L. Harris. 2001. A Pragmatic Implementation of Non-blocking Linked-Lists. In *Distributed Computing, 15th International Conference, DISC 2001, Lisbon, Portugal, October 3-5, 2001, Proceedings (Lecture Notes in Computer Science, Vol. 2180)*, Jennifer L. Welch (Ed.). Springer, 300–314. https://doi.org/10.1007/3-540-45414-4_21

Antony Hosking. 2006. Portable, Mostly-Concurrent, Mostly-Copying Garbage Collection for Multi-Processors, Vol. 2006. https://doi.org/10.1145/1133956.1133963

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. 2012. *The Garbage Collection Handbook : The Art of Automatic Memory Management.* CRC Press.

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.

Alexei Yu Kitaev, Alexander Shen, Mikhail N Vyalyi, and Mikhail N Vyalyi. 2002. *Classical and quantum computation.* Number 47. American Mathematical Soc.

Chaitanya Koparkar, Mike Rainey, Michael Vollmer, Milind Kulkarni, and Ryan R. Newton. 2021. Efficient Tree-Traversals: Reconciling Parallelism and Dense Data Representations. 5, ICFP, Article 91 (aug 2021), 29 pages. https://doi.org/10.1145/3473596

A. Krishnamurthy, D. E. Culler, A. Dusseau, S. C. Goldstein, S. Lumetta, T. von Eicken, and K. Yelick. 1993. Parallel Programming in Split-C. In *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing* (Portland, Oregon, USA) *(Supercomputing '93)*. ACM, New York, NY, USA, 262–273. https://doi.org/10.1145/169627.169724

Ananya Kumar, Guy E. Blelloch, and Robert Harper. 2017. Parallel functional arrays. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 706–718.

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.

Jonatan Lindén and Bengt Jonsson. 2013. A Skiplist-Based Concurrent Priority Queue with Minimal Memory Contention. In *Principles of Distributed Systems - 17th International Conference, OPODIS 2013, Nice, France, December 16-18, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8304)*, Roberto Baldoni, Nicolas Nisse, and Maarten van Steen (Eds.). Springer, 206–220. https://doi.org/10.1007/978-3-319-03850-6_15

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.

Maged M. Michael and Michael L. Scott. 1996. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, Philadelphia, Pennsylvania, USA, May 23-26, 1996*, James E. Burns and Yoram Moses (Eds.). ACM, 267–275. https://doi.org/10.1145/248052.248106

MLton . 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. 2018a. Competitive Parallelism: Getting Your Priorities Right. *Proc. ACM Program. Lang.* 2, ICFP, Article 95 (July 2018), 30 pages.

Stefan K. Muller, Umut A. Acar, and Robert Harper. 2018b. Types and Cost Models for Responsive Parallelism. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP '18)*.

Robert W. Numrich and John Reid. 1998. Co-array Fortran for Parallel Programming. *SIGPLAN Fortran Forum* 17, 2 (Aug. 1998), 1–31. https://doi.org/10.1145/289918.289920

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

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.

Prakash Prabhu, Stephen R. Beard, Sotiris Apostolakis, Ayal Zaks, and David I. August. 2018. MemoDyn: Exploiting Weakly Consistent Data Structures for Dynamic Parallel Memoization. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques* (Limassol, Cyprus) *(PACT '18)*. Association for Computing Machinery, New York, NY, USA, Article 15, 12 pages. https://doi.org/10.1145/3243176.3243193

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.

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

Julian Shun and Guy E. Blelloch. 2014. Phase-Concurrent Hash Tables for Determinism. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures* (Prague, Czech Republic) *(SPAA '14)*. Association for Computing Machinery, New York, NY, USA, 96–107. https://doi.org/10.1145/2612669.2612687

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. 2020. 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, 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.

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

Sam Westrick, Jatin Arora, and Umut A. Acar. 2022. Entanglement Detection With Near-Zero Cost. In *Proceedings of the 24th ACM SIGPLAN International Conference on Functional Programming (ICFP 2022)*.

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)"*.

Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. 1998. Titanium: a high-performance Java dialect. *Concurrency: Practice and Experience* 10, 11-13 (1998), 825–836.

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.

Received 2022-11-10; accepted 2023-03-31