Disentanglement in Nested-Parallel Programs

SAM WESTRICK, Carnegie Mellon University, USA
ROHAN YADAV, Carnegie Mellon University, USA
MATTHEW FLUET, Rochester Institute of Technology, USA
UMUT A. ACAR, Carnegie Mellon University, USA

Nested parallelism has proved to be a popular approach for programming the rapidly expanding range of multicore computers. It allows programmers to express parallelism at a high level and relies on a run-time system and a scheduler to deliver efficiency and scalability. As a result, many programming languages and extensions that support nested parallelism have been developed, including in C/C++, Java, Haskell, and ML. Yet, writing efficient and scalable nested parallel programs remains challenging, primarily due to difficult concurrency bugs arising from destructive updates or effects. For decades, researchers have argued that functional programming can simplify writing parallel programs by allowing more control over effects but functional programs continue to underperform in comparison to parallel programs written in lower-level languages. The fundamental difficulty with functional languages is that they have high demand for memory, and this demand only grows with parallelism.

In this paper, we identify a memory property, called ***disentanglement***, of nested-parallel programs, and propose memory management techniques for improved efficiency and scalability. Disentanglement allows for (destructive) effects as long as concurrently executing threads do not gain knowledge of the memory objects allocated by each other. We formally define disentanglement by considering an ML-like higher-order language with mutable references and presenting a dynamic semantics for it that enables reasoning about computation graphs of nested parallel programs. Based on this graph semantics, we formalize a classic correctness property—determinacy race freedom—and prove that it implies disentanglement. This establishes that disentanglement applies to a relatively broad class of parallel programs. We then propose memory management techniques for nested-parallel programs that take advantage of disentanglement for improved efficiency and scalability. We show that these techniques are practical by extending the MLton compiler for Standard ML to support this form of nested parallelism. Our empirical evaluation shows that our techniques are efficient and scale well.

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

Additional Key Words and Phrases: disentanglement, data race, parallel computing, memory management, functional programming

ACM Reference Format:
Sam Westrick, Rohan Yadav, Matthew Fluet, and Umut A. Acar. 2020. Disentanglement in Nested-Parallel Programs. *Proc. ACM Program. Lang.* 4, POPL, Article 47 (January 2020), 32 pages. https://doi.org/10.1145/3371115

Authors' addresses: Sam Westrick, Carnegie Mellon University, Pittsburgh, PA, USA, swestric@cs.cmu.edu; Rohan Yadav, Carnegie Mellon University, Pittsburgh, PA, USA, rohany@alumni.cmu.edu; Matthew Fluet, Rochester Institute of Technology, Rochester, NY, USA, mtf@cs.rit.edu; Umut A. Acar, Carnegie Mellon University, Pittsburgh, PA, USA, umut@cs.cmu.edu.

2475-1421/2020/1-ART47
https://doi.org/10.1145/3371115

1 INTRODUCTION

Hardware advances of the past decade have brought shared-memory parallelism to the mainstream. Nearly every computing device today is a parallel computer, including smartphones with 10 cores, workstations with dozens of cores [Sodani 2015], rack-mounted servers with hundreds of cores [Corp. 2017], and high-end machines with thousands of cores [Robinson 2017]. As a response to these developments, nested-parallelism has emerged as a promising technique for utilizing parallel hardware. This technique allows parallel computations to spawn other "nested" parallel computations, which in turn enables programmers to express parallelism at a high level, for example by using constructs such as parallel loops and fork/join (spawn/sync) that permit parallel evaluation of (recursive) functions. Nested parallelism typically relies on a thread scheduler to create and schedule parallel tasks automatically and efficiently, thus relieving the programmer from the burden of managing parallelism manually. Many effective scheduling algorithms have been designed and implemented [Acar et al. 2002, 2018b; Arora et al. 2001; Blelloch et al. 1997; Blumofe and Leiserson 1999].

Nested parallelism has been adopted by many programming languages and extensions including those based on procedural languages such as Intel Thread Building Blocks (a C++ library) [Intel 2011], Cilk (an extension of C) [Blumofe et al. 1996; Frigo et al. 1998], OpenMP [OpenMP Architecture Review Board [n.d.]], Task Parallel Library (a .NET library) [Leijen et al. 2009], Java Fork/Join Framework [Lea 2000], Habanero Java [Imam and Sarkar 2014], and X10 [Charles et al. 2005], as well as functional languages including multiLisp [Halstead 1984], Id [Arvind et al. 1989], NESL [Blelloch et al. 1994], several forms of parallel Haskell [Li et al. 2007; Marlow 2011; Peyton Jones et al. 2008], and several forms of parallel ML [Fluet et al. 2011; Guatto et al. 2018; Ohori et al. 2018; Raghunathan et al. 2016; Sivaramakrishnan et al. 2014; Spoonhower 2009].

Even though this work has made significant progress, writing parallel programs continues to present a number of important challenges. Many of these challenges stem from operations on shared memory, especially in the presence of memory effects or destructive updates. Memory effects (mutation) are crucial for theoretical and practical efficiency of nested parallel programs, both in the underlying runtime system (e.g., to support communication for the purposes of scheduling), and at the application level (e.g., to implement collection data structures using mutable arrays). The challenge is that the same memory effects that improve efficiency can lead to race conditions, which typically harm correctness in complex and unpredictable ways [Adve 2010; Allen and Padua 1987; Bocchino et al. 2011, 2009; Boehm 2011; Emrath et al. 1991; Mellor-Crummey 1991; Netzer and Miller 1992; Steele Jr. 1990].

Researchers have therefore argued for decades that pure (mutation or effect free) functional programming can make things much simpler and safer [Arvind et al. 1989; Blelloch 1996; Blelloch et al. 1994; Fluet et al. 2011; Halstead 1984; Hammond 2011; Li et al. 2007; Marlow 2011; Ohori et al. 2018; Peyton Jones et al. 2008; Raghunathan et al. 2016; Sivaramakrishnan et al. 2014; Spoonhower 2009; Ziarek et al. 2011]. Pure functional programming is safe for parallelism and avoids data races. Furthermore, because they support higher order functions (e.g., map, filter, reduce over collections of data), functional languages enable expressing parallel algorithms elegantly and succinctly. Functional programs, however, fall short when it comes to efficiency and scalability. The main reason for this is the absence of side effects, which leads to increased demand for memory, causing functional languages allocate at a very high rate [Appel 1989; Appel and Shao 1996; Auhagen et al. 2011; Doligez and Gonthier 1994; Doligez and Leroy 1993; Gonçalves 1995; Gonçalves and Appel 1995; Marlow 2011]. Functional programming languages need not be pure and can support effects (and typically do), but handling of effects complicates the memory subsystem of functional languages in the parallel context. Although efficient memory management techniques have been

developed for sequential functional languages, this remains an open problem in the parallel setting. Fundamentally, the problem is that parallel hardware makes functional languages, which are already memory hungry, even more hungry by making it possible for multiple processors to operate on the memory at the same time.

One way to solve this problem is to develop efficient support for effects in parallel functional languages and use type systems and powerful compositional facilities to tame the correctness challenges that they pose. Guatto et al. made some progress towards this end and reported significant efficiency improvements solely using run-time techniques. However, the range of effects they support are limited to those in the sequential "leaves" of a parallel computation [Guatto et al. 2018]. We believe that going beyond Guatto et al.'s work requires developing a deeper understanding of both the role of effects in nested parallel programs and the invariants maintained by parallel programs and, ultimately, developing a whole new class of memory management techniques that can meet functional programs demand for memory on modern hardware.

In this paper, we take important steps towards understanding the role of effects in nested parallel programs and how they can be exploited to improve efficiency. We consider an effectful nested-parallel language, which extends the lambda calculus with references and nested parallelism (Section 2) and show that all *determinacy-race-free* [Feng and Leiserson 1997] nested-parallel programs exhibit a ***disentanglement*** property (Section 3.3). At a high level, the disentanglement property ensures that a thread cannot access memory allocated by a concurrently executing thread. Intuitively speaking, race-free programs are disentangled because they prohibit communication between concurrently executing threads. Disentanglement, however, is weaker than race-freedom and allows certain kinds of races: it allows concurrently executing threads to communicate via shared data that is known to both threads, i.e., via objects allocated by shared ancestor threads in the "call graph".

Disentanglement enforces a separation property between the memory objects allocated by concurrently executing threads: they cannot point to each other. We show that this property can be exploited to support effects efficiently in nested parallel languages. The basic idea is to assign each thread its own *heap*, or segment of memory, in which the thread performs all of its allocations. Because of disentanglement, we then know that concurrently executing threads have heaps that cannot directly point to each other. We show that we can organize memory to support efficient and parallel allocation and reclamation of memory. The idea behind this organization is to represent memory as a tree of heaps where the leaves of the tree correspond to concurrently executing threads and the nodes of the tree correspond to suspended parents of forked threads. The tree dynamically grows and collapses as the computation proceeds, e.g., as new threads are created at forks and as threads are destroyed at joins. This design allows concurrently executing threads to allocate memory with no synchronization. Furthermore, it allows concurrent threads to share and side-effect data allocated by ancestors without needing to synchronize or *promote* (copy) data; instead, disentanglement makes it possible to delay promotions until opportune moments, such as during garbage collections.

We present MPL, a compiler and runtime system that implements these techniques by extending the MLton compiler [MLton [n.d.]]. In order to evaluate MPL, we consider a variety of state-of-the-art parallel benchmarks that have been developed in the context of procedural parallel programming languages (C/C++ and extensions). Such benchmarks are highly effectful, utilizing destructive updates extensively for efficiency, but nonetheless we found that nearly all of them are disentangled. This is because the benchmarks either (a) are race-free and therefore disentangled, or (b) employ carefully crafted races for improved efficiency benefits in a manner that does not violate disentanglement. In an evaluation on these benchmarks, we show that MPL performs well: it achieves small overheads compared optimized sequential baselines, and scales well as the number

$$
\begin{array}{rlll}
\textit{Variables} & x, f & & \\
\textit{Numbers} & n & \in & \mathbb{N} \\
\textit{Memory Locations} & \ell & & \\
\textit{Types} & \tau & ::= & \mathsf{nat} \mid \tau \times \tau \mid \tau \rightarrow \tau \mid \tau \ \mathsf{ref} \\
\textit{Storables} & s & ::= & n \mid \mathsf{fun}\ f\ x\ \mathsf{is}\ e \mid \langle \ell, \ell \rangle \mid \mathsf{ref}\ \ell \\
\textit{Expressions} & e & ::= & \ell \mid s \mid x \mid e\ e \mid \langle e, e \rangle \mid \mathsf{fst}\ e \mid \mathsf{snd}\ e \mid \mathsf{ref}\ e \mid\ !\, e \mid e := e \mid \langle e \parallel e \rangle \\
\textit{Memory} & \mu & \in & \textit{Locations} \rightharpoonup \textit{Storables} \\
\textit{Actions} & \alpha & ::= & \mathbf{A}\ell \Leftarrow s \mid \mathbf{R}\ell \Rightarrow s \mid \mathbf{W}\ell \Leftarrow s \\
\textit{Computation Graphs} & g & ::= & \bullet \mid \alpha \mid g \oplus g \mid g \otimes g \\
\textit{Open Computation Graphs} & G & ::= & [g] \mid g \oplus (G \otimes G)
\end{array}
$$

Fig. 1. Syntax

of cores increases. We also compare MPL to other languages by considering the classic problem of sorting. The results show that MPL generates code that is within a factor of two of Cilk/C and outperforms other memory-managed languages including Java, Go, and Haskell.

The contributions of this paper include the following.

- A theory of disentangled effects (Sections 2 and 3).
- Algorithms and techniques for memory management of disentangled programs (Section 4).
- An implementation of Parallel ML which extends the MLton compiler with support for parallel execution and memory management (Section 5).
- An empirical evaluation (Section 6).

2 LANGUAGE AND GRAPH SEMANTICS

We consider a simple fork-join (nested-parallel) language that, in order to define disentanglement, has two novel features. (1) The operational semantics explicitly allocates memory for *all* data, mutable and immutable alike. This lets us account fully for all memory operations. (2) During execution, a program constructs an execution trace called a *computation graph*. A computation graph records the history of a computation in terms of actions that are performed upon a shared memory and a partial order on these actions, which captures the structure of parallelism. The generality of computation graphs makes them suitable for defining both disentanglement and determinacy-race-freedom (Section 3).

Typically, a big-step semantics might be used to construct computation graphs. However, since our computational model permits both parallelism and side-effects, the semantics must account for fine-grained interleaving of (concurrent) computations. We therefore use a small-step semantics, which, due to possible interleavings of parallel steps that can affect a shared memory, is non-deterministic. In order to construct computation graphs in a small-step manner, we define *open computation graphs* (Section 2.3) which encode the structure of active parallel tasks, allowing each small step to extend the computation graph "at the right place".

For completeness, we provide a type system for the language and prove progress and preservation in the Appendix. Note that the language on its own does not statically or dynamically enforce any guarantees of disentanglement and/or race-freedom. This is intentional, allowing us to define these properties as behaviors of execution that are not necessarily exhibited by all programs.

2.1 Syntax

Figure 1 shows the syntax of the language.

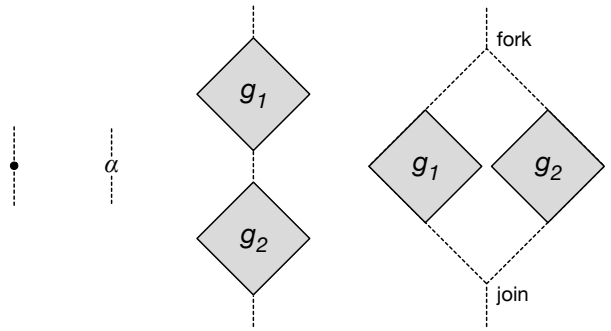

Fig. 2. A series-parallel computation graphs is either the empty graph •, a single action α, a sequential composition $g_1 \oplus g_2$, or a parallel composition $g_1 \otimes g_2$. The dashed lines are control dependencies, implicitly pointing down.

Types. The types include a base type of natural numbers, as well as products (tuples/pairs), functions, and mutable references.

Memory Locations and Storables. To account precisely for memory operations, the language distinguishes between *storables* s, which are stored in memory, and *memory locations* ℓ. Storables consist of natural numbers, named recursive functions, pairs of memory locations, and mutable references to other memory locations. Locations are the only irreducible form of the language; that is, all terminating expressions eventually step to a memory location.

Expressions. Expressions consist of memory locations, storables, variables and applications, pairs and their projections, mutable references with explicit lookup and update, and the parallel pair $\langle e_1 \parallel e_2 \rangle$, which is used to execute e_1 and e_2 in parallel. For convenience we will use the syntactic sugar (let $x = e_1$ in e_2) to mean (fun $f\ x$ is e_2) e_1, where f does not appear free in e_2.

Memory. A separate memory μ is used to map locations to storables. We write $\mathrm{dom}(\mu)$ for the set of locations mapped by μ, $\mu(\ell)$ to look up the storable associated with ℓ, and $\mu[\ell \hookrightarrow s]$ to extend μ with a new mapping (with the implicit requirement that $\ell \notin \mathrm{dom}(\mu)$).

2.2 Computation Graphs and Actions

Traditionally, a nested parallel computation is represented by using a directed acyclic graph, or ***dag***, that consists of vertices and edges. Each vertex represents an executed instruction and each edge represents the control dependency between two instructions. We augment the dag by annotating every vertex with the ***action*** it performed upon shared memory. Actions, denoted α, can be one of the following:

- $\mathbf{A}\ell \Leftarrow s$ is the allocation of location ℓ, initialized with contents s.
- $\mathbf{R}\ell \Rightarrow s$ is a read (lookup) at ℓ which returned s.
- $\mathbf{W}\ell \Leftarrow s$ is a write (update) at ℓ which stored s.

We call the dag augmented with actions a ***computation graph***. Due to the structure of nested (fork-join) parallelism, computation graphs have a *series-parallel* structure, as depicted in Figure 2. In particular, a computation graph can be any one of the following.

- The empty (no-op) graph, denoted •.
- A single action α.
- The sequential composition of graphs g_1 and g_2, denoted $g_1 \oplus g_2$, where there is an edge connecting the last vertex of g_1 to the first of g_2, indicating that all of g_1 happened before g_2.

- The parallel composition of graphs g_1 and g_2, denoted $g_1 \otimes g_2$, indicating that neither g_1 nor g_2 happened before the other. In this case there are two special vertices which arise: a *fork* and a corresponding *join*. The fork, which has out-degree two, is connected to each of the first vertices of g_1 and g_2. The join has in-degree two, and its incoming neighbors are the last vertices of g_1 and g_2.

2.3 Open Computation Graphs

As described thus far, computation graphs g can be understood as representing the history of "completed" computations. However, while a computation is in progress, we need a way of constructing its computation graph one step at a time. To do so, we exploit a specific structure dictated by the nesting of parallel tasks.

At every moment during execution, the tasks of a nested-parallel program can be organized into a tree structure, called a ***task tree***, where each node represents a task. Each task in the tree has either exactly two children (its subtasks) or no children. A characteristic feature of nested parallelism is that, when a task forks two subtasks, the task suspends its own execution until both subtasks complete. Therefore in the task tree, each internal task is suspended, and the leaves are "active" tasks that may step in parallel. When both children of an internal node terminate, the corresponding leaves disappear from the tree and the internal node becomes a leaf, resuming its execution.

Each time a leaf task takes a step, it may perform an action that needs to be recorded in the computation graph. To locate where in the computation graph this new action should go, we partition the computation graph into many smaller computation graphs and organize them in a tree structure mirroring the task tree. We call this tree structure an ***open computation graph***, denoted G. In an open computation graph, each node records the local history of its corresponding task in the task tree.

The internal nodes of an open computation graph have the form $g \oplus (G_1 \otimes G_2)$ where g is the history of the corresponding task up until the moment it forked two subtasks, and G_1 and G_2 are the open computation graphs of its subtasks. A leaf has the form $[g]$: this is a computation that may be extended with a new action when the corresponding task takes a step (e.g., a step from $[g]$ to $[g \oplus (\mathbf{A}\ell \Leftarrow s)]$).

2.4 Operational Semantics

The operational semantics, defined in Figure 3, is a single-step relation

$$\mu \,;\, G \,;\, e \longmapsto \mu' \,;\, G' \,;\, e'.$$

Each step takes a memory μ, an open computation graph G, and an expression e and produces a new state consisting of μ', G', and e'. Steps are non-deterministic due to possible interleavings of rules ParL and ParR.

Allocation. The allocation rule Alloc is the only way to create new memory locations. It steps a storable s to a fresh location ℓ, extends the memory by mapping ℓ to s, and records $\mathbf{A}\ell \Leftarrow s$ in the computation graph.

Reading from Memory. There are four rules which read from memory: function application (rule App), pair projection (rules Fst and Snd), and reference lookup (rule Bang). The semantics does *not* distinguish between reads of mutable and immutable data. In rule App, the function at location ℓ_1 is applied to the argument at location ℓ_2. This is accomplished by reading from ℓ_1 (to acquire the source code of the function) and substituting both ℓ_1 and ℓ_2 into the function body e_b. Note that rule App performs a read at ℓ_1 but not at ℓ_2.

Execution $\boxed{\mu\,;G\,;e \longmapsto \mu'\,;G'\,;e'}$

$$\frac{\ell \notin \mathrm{dom}(\mu)}{\mu\,;[g]\,;s \longmapsto \mu[\ell \hookrightarrow s]\,;[g \oplus (\mathbf{A}\ell \Leftarrow s)]\,;\ell}\ \textsc{Alloc}$$

$$\frac{\mu\,;G\,;e_1 \longmapsto \mu'\,;G'\,;e_1{}'}{\mu\,;G\,;(e_1\ e_2) \longmapsto \mu'\,;G'\,;(e_1'\ e_2)}\ \textsc{AppSL} \qquad \frac{\mu\,;G\,;e_2 \longmapsto \mu'\,;G'\,;e_2{}'}{\mu\,;G\,;(\ell_1\ e_2) \longmapsto \mu'\,;G'\,;(\ell_1\ e_2')}\ \textsc{AppSR}$$

$$\frac{\mu(\ell_1) = \mathsf{fun}\ f\ x\ \mathsf{is}\ e_b}{\mu\,;[g]\,;(\ell_1\ \ell_2) \longmapsto \mu\,;[g \oplus (\mathbf{R}\ell_1 \Rightarrow \mathsf{fun}\ f\ x\ \mathsf{is}\ e_b)]\,;[\ell_1, \ell_2\ /\ f, x]e_b}\ \textsc{App}$$

$$\frac{\mu\,;G\,;e_1 \longmapsto \mu'\,;G'\,;e_1{}'}{\mu\,;G\,;\langle e_1, e_2\rangle \longmapsto \mu'\,;G'\,;\langle e_1', e_2\rangle}\ \textsc{PairSL} \qquad \frac{\mu\,;G\,;e_2 \longmapsto \mu'\,;G'\,;e_2{}'}{\mu\,;G\,;\langle \ell_1, e_2\rangle \longmapsto \mu'\,;G'\,;\langle \ell_1, e_2'\rangle}\ \textsc{PairSR}$$

$$\frac{\mu\,;G\,;e \longmapsto \mu'\,;G'\,;e'}{\mu\,;G\,;(\mathsf{fst}\ e) \longmapsto \mu'\,;G'\,;(\mathsf{fst}\ e')}\ \textsc{FstS} \qquad \frac{\mu(\ell) = \langle \ell_1, \ell_2\rangle}{\mu\,;[g]\,;(\mathsf{fst}\ \ell) \longmapsto \mu\,;[g \oplus (\mathbf{R}\ell \Rightarrow \langle \ell_1, \ell_2\rangle)]\,;\ell_1}\ \textsc{Fst}$$

$$\frac{\mu\,;G\,;e \longmapsto \mu'\,;G'\,;e'}{\mu\,;G\,;(\mathsf{snd}\ e) \longmapsto \mu'\,;G'\,;(\mathsf{snd}\ e')}\ \textsc{SndS} \qquad \frac{\mu(\ell) = \langle \ell_1, \ell_2\rangle}{\mu\,;[g]\,;(\mathsf{snd}\ \ell) \longmapsto \mu\,;[g \oplus (\mathbf{R}\ell \Rightarrow \langle \ell_1, \ell_2\rangle)]\,;\ell_2}\ \textsc{Snd}$$

$$\frac{\mu\,;G\,;e \longmapsto \mu'\,;G'\,;e'}{\mu\,;G\,;(\mathsf{ref}\ e) \longmapsto \mu'\,;G'\,;(\mathsf{ref}\ e')}\ \textsc{RefS}$$

$$\frac{\mu\,;G\,;e \longmapsto \mu'\,;G'\,;e'}{\mu\,;G\,;(!\,e) \longmapsto \mu'\,;G'\,;(!\,e')}\ \textsc{BangS} \qquad \frac{\mu(\ell_1) = \mathsf{ref}\ \ell_2}{\mu\,;[g]\,;(!\,\ell_1) \longmapsto \mu\,;[g \oplus (\mathbf{R}\ell_1 \Rightarrow \mathsf{ref}\ \ell_2)]\,;\ell_2}\ \textsc{Bang}$$

$$\frac{\mu\,;G\,;e_1 \longmapsto \mu'\,;G'\,;e_1{}'}{\mu\,;G\,;(e_1 := e_2) \longmapsto \mu'\,;G'\,;(e_1' := e_2)}\ \textsc{UpdSL} \qquad \frac{\mu\,;G\,;e_2 \longmapsto \mu'\,;G'\,;e_2{}'}{\mu\,;G\,;(\ell_1 := e_2) \longmapsto \mu'\,;G'\,;(\ell_1 := e_2')}\ \textsc{UpdSR}$$

$$\frac{}{\mu_0[\ell_1 \hookrightarrow s]\,;[g]\,;(\ell_1 := \ell_2) \longmapsto \mu_0[\ell_1 \hookrightarrow \mathsf{ref}\ \ell_2]\,;[g \oplus (\mathbf{W}\ell_1 \Leftarrow \mathsf{ref}\ \ell_2)]\,;\ell_2}\ \textsc{Upd}$$

$$\frac{}{\mu\,;[g]\,;\langle e_1 \parallel e_2\rangle \longmapsto \mu\,;g \oplus ([\bullet] \otimes [\bullet])\,;\langle e_1 \parallel e_2\rangle}\ \textsc{Fork}$$

$$\frac{}{\mu\,;g \oplus ([g_1] \otimes [g_2])\,;\langle \ell_1 \parallel \ell_2\rangle \longmapsto \mu\,;[g \oplus (g_1 \otimes g_2)]\,;\langle \ell_1, \ell_2\rangle}\ \textsc{Join}$$

$$\frac{\mu\,;G_1\,;e_1 \longmapsto \mu'\,;G_1'\,;e_1'}{\mu\,;g \oplus (G_1 \otimes G_2)\,;\langle e_1 \parallel e_2\rangle \longmapsto \mu'\,;g \oplus (G_1' \otimes G_2)\,;\langle e_1' \parallel e_2\rangle}\ \textsc{ParL}$$

$$\frac{\mu\,;G_2\,;e_2 \longmapsto \mu'\,;G_2'\,;e_2'}{\mu\,;g \oplus (G_1 \otimes G_2)\,;\langle e_1 \parallel e_2\rangle \longmapsto \mu'\,;g \oplus (G_1 \otimes G_2')\,;\langle e_1 \parallel e_2'\rangle}\ \textsc{ParR}$$

Fig. 3. Language Dynamics.

```
type point = int × int

fun transpose (P: point array) (n: int) =
  let
    fun tr i j =
      if j-i = 1 then
        let ⟨x, y⟩ = P[i]
        in P[i] := ⟨y, x⟩
        end
      else
        let
          mid = (i + j) / 2
          ⟨_,_⟩ = ⟨tr i mid ‖ tr mid j⟩
        in
          ⟨⟩
        end
  in
    tr 0 n
  end
```

Fig. 4. The function transpose transposes each element in array P of size n.

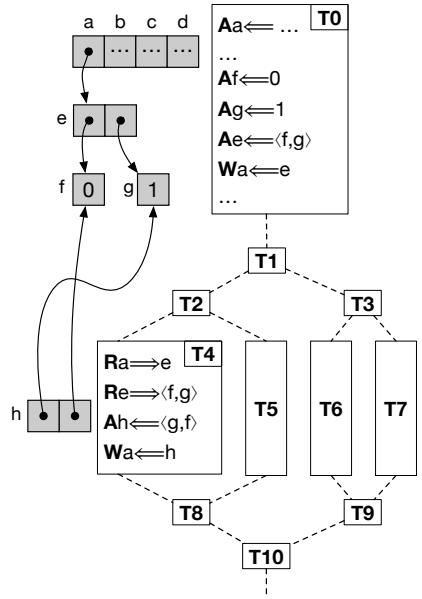

Fig. 5. Computation graph for transpose on an input array of length 4.

Writing to Memory. The rule Upd updates the storable at ℓ_1 to refer to ℓ_2. This is the only way the contents of an existing memory location can change during execution.

Parallelism. Parallelism is accomplished through four rules: forking new tasks (rule Fork), joining completed tasks (rule Join), and subtask stepping (rules ParL and ParR). The Fork rule records the beginning of two new parallel tasks in the computation graph. When two subtasks have completed, rule Join assembles their results as a standard pair and records that the tasks have completed in the computation graph. The ParL and ParR rules non-deterministically interleave steps of the subtasks, recording their actions in the appropriate subgraph. Note that the shape of the open computation graph determines whether a parallel pair forks, evaluates the subtasks, or joins.

2.5 Example: Transposing Points in 2D

Consider a function transpose, shown in Figure 4 using an ML-like syntax, that takes an array of points in 2D space and transposes each point in parallel by swapping its x- and y-coordinates. For this example, we assume that the language has arrays, which are natural extensions of mutable references (whereas a ref is a single mutable location, an array is a sequence of many mutable locations). The function relies on a recursive function tr that takes two indices specifying a segment of the input array. If the segment has size 1 then the function allocates a fresh point whose coordinates are the derived from the x- and y-coordinates of the sole element in the segment.[1] Otherwise, the function splits the segment in the middle into two segments, and transposes them recursively in parallel. Because this function performs constant work for each and every element of the array, the transpose function requires asymptotically linear work in n. Its span—the longest chain of dependencies—is logarithmic (in n). The function therefore exposes significant parallelism.

[1] A more realistic implementation would control granularity by reverting to a sequential transpose below a threshold size.

$\boxed{A \vdash g \text{ de}}$

$$\frac{}{A \vdash \bullet \text{ de}} \qquad \frac{\text{locs}(s) \subseteq A}{A \vdash (\mathbf{A}\ell \Leftarrow s) \text{ de}} \qquad \frac{\ell \in A \qquad \text{locs}(s) \subseteq A}{A \vdash (\mathbf{R}\ell \Rightarrow s) \text{ de}} \qquad \frac{\ell \in A \qquad \text{locs}(s) \subseteq A}{A \vdash (\mathbf{W}\ell \Leftarrow s) \text{ de}}$$

$$\frac{A \vdash g_1 \text{ de} \qquad A \uplus \mathsf{A}(g_1) \vdash g_2 \text{ de}}{A \vdash g_1 \oplus g_2 \text{ de}} \qquad \frac{A \vdash g_1 \text{ de} \qquad A \vdash g_2 \text{ de}}{A \vdash g_1 \otimes g_2 \text{ de}}$$

$\boxed{A \vdash G \text{ de}}$

$$\frac{A \vdash g \text{ de}}{A \vdash [g] \text{ de}} \qquad \frac{A \vdash g \text{ de} \qquad A \uplus \mathsf{A}(g) \vdash G_1 \text{ de} \qquad A \uplus \mathsf{A}(g) \vdash G_2 \text{ de}}{A \vdash g \oplus (G_1 \otimes G_2) \text{ de}}$$

Fig. 6. Definition of disentanglement. Variable A denotes the set of known allocations.

Figure 5 summarizes the computation graph for an execution of `transpose` with `n = 4` elements. The diagram uses a single vertex to represent entire tasks (sequential regions), and dashed lines to indicate control dependencies between tasks. All dashed lines implicitly point down. The gray square boxes represent memory objects and are labeled with their memory locations, and the solid arrows are pointers in memory. The input array consists of memory locations {a, b, c, d}, each of which points to a pair of locations which in turn point to integers. We assume that the root task **T0** allocates and initializes the input array and calls `transpose`, the root of which is the task **T1**. Task **T1** then forks two subtasks **T2** and **T3**, which in turn each fork two more (**T4**-**T7**). The tasks **T4**, **T5**, **T6**, and **T7** perform the steps of reading from the array, allocating new tuples with x- and y-coordinates swapped, and writing these tuples back into the array. We show the specific actions of **T4** and omit the actions of tasks **T5**-**T7**, which are all similar to **T4**. As depicted, the pointers show the state of memory *before* the write in **T4** occurs; after the write, location **a** should point to **h**. When the tasks **T4**, **T5**, **T6**, and **T7** all complete, they join "back up" with tasks **T8**, **T9**, and **T10**, at which point the computation is complete.

3 DISENTANGLEMENT

To define disentanglement, we look more closely at the actions in the computation graph and define two notions—knowledge and use—where we say that actions *know* locations and *use* locations. An action ***knows*** a location ℓ if ℓ was allocated by the action itself or by an ancestor in the computation graph. An action ***uses*** a location ℓ if ℓ is being accessed by the action or ℓ is part of the storable being allocated, written, or read by the action. Specifically, the actions $\mathbf{A}\ell \Leftarrow s$, $\mathbf{W}\ell \Leftarrow s$, and $\mathbf{R}\ell \Rightarrow s$ each *use* exactly the locations $\ell \cup \text{locs}(s)$. (The function $\text{locs}(e)$, defined in the Appendix, is the set of locations mentioned by expression e.) We can then define disentanglement as the property that *every action uses only locations that it knows.*

Example. Returning to the `tranpose` example of Section 2.5, we can see that this computation is disentangled, as each action in Figure 5 only uses locations that were allocated by itself or ancestors.

3.1 Definition

The formal definition of disentanglement is given in Figure 6 as a judgement $A \vdash g$ de, which establishes that computation graph g is disentangled with respect to known allocations A. The judgement $A \vdash G$ de similarly establishes disentanglement on open computation graphs G. Both judgements are given in terms of two auxiliary functions (defined in the Appendix): $\mathsf{A}(g)$ is the set of locations allocated by g, and $\text{locs}(e)$ is the set of locations mentioned by expression e.

The rules establish for every action that all locations used by that action are known. For reads $\mathbf{R}\ell \Rightarrow s$ and writes $\mathbf{W}\ell \Leftarrow s$, this is verified by checking that ℓ and $\mathsf{locs}(s)$ are among the known allocations A. For allocation actions $\mathbf{A}\ell \Leftarrow s$, the rules only need to establish that $\mathsf{locs}(s)$ are among the known allocations, since ℓ is allocated by this action and therefore is certainly known. The set of known allocations A accumulates at sequential compositions $g_1 \oplus g_2$, allowing g_2 to know all allocations of g_1. Similarly, in open computation graphs $g \oplus (G_1 \otimes G_2)$, all allocations of g_1 are known to G_1 and G_2. Crucially, in parallel compositions, the two subgraphs do not know of each other's allocations.

3.2 Checking for Disentanglement

It is possible to check at run-time that a computation is disentangled by checking only the results of read actions. That is, if every read in a computation uses only locations that it knows, then the computation is disentangled. Note however that this approach is execution-dependent. A program which *can* violate disentanglement might not necessarily do so in all of its possible executions. For example, entanglement might only occur due to a race condition.

Correctness. To see why this approach is correct, consider the notion of *discovery*, which is a violation of disentanglement: we say that an action ***discovers*** location ℓ if the action uses ℓ without knowing about it, and additionally none of the action's ancestors use ℓ. In all computations that violate disentanglement, there is at least one discovery. Furthermore, discovery is only possible at read actions, because memory-safe programs can only obtain pointers either by allocating new objects or by following in-memory pointers. Therefore a computation is disentangled if and only if none of its reads discover locations.

Practicality. To facilitate efficient checking in practice, not all reads need to be checked. We don't need to check reads of non-pointer data (e.g. reading from an array of "unboxed" integers) because these cannot possibly discover a location. We also don't need to check reads of immutable data because, if such a read discovers a location, then there must be another discovery at an ancestor of the read which can be blamed instead. For example, if the action '$\mathbf{R}\ell \Rightarrow \langle \ell_1, \ell_2 \rangle$' discovers ℓ_1, then there must also have been a previous discovery of ℓ, because immutable data can only be constructed in terms of locations that are allocated or discovered by ancestor actions.

3.3 Determinacy-Race-Free Computations are Disentangled

In this section, we show that disentanglement is guaranteed when a computation is free of a certain kind of data race called a *determinacy race*. A ***determinacy race*** occurs when two concurrent actions both access the same location, and at least one of these accesses modifies the location [Feng and Leiserson 1997]. As the name suggests, the lack of determinacy races is sufficient to guarantee determinism [Emrath and Padua 1988; Steele Jr. 1990]. A computation with no determinacy races is ***determinacy-race-free*** (DRF).

We can determine whether or not a computation is DRF by inspecting its computation graph. Specifically, we need to verify that for every pair of concurrent actions α_1 and α_2, which access the same location, that they are both read actions. A location is *accessed* when its contents are either inspected or modified. Specifically, each of $\mathbf{A}\ell \Leftarrow s$, $\mathbf{R}\ell \Rightarrow s$, and $\mathbf{W}\ell \Leftarrow s$ are considered to *access* location ℓ. The actions which modify a location are writes and allocations: both of $\mathbf{A}\ell \Leftarrow s$ and $\mathbf{W}\ell \Leftarrow s$ modify location ℓ. We treat allocations as modifications because allocations also initialize the location, which in an implementation requires a write to the location.

With this setup, we can formally define determinacy-race-freedom on computation graphs with a judgement $F \vdash g$ drf which establishes that computation graph g is DRF with respect to a "forbidden" set of locations F. The definition is given in Figure 7, together with a corresponding judgement

$\boxed{F \vdash g \text{ drf}}$

$$\frac{}{F \vdash \bullet \text{ drf}} \qquad \frac{\ell \notin F}{F \vdash (\mathbf{A}\ell \Leftarrow s) \text{ drf}} \qquad \frac{\ell \notin F}{F \vdash (\mathbf{W}\ell \Leftarrow s) \text{ drf}} \qquad \frac{\ell \notin F}{F \vdash (\mathbf{R}\ell \Rightarrow s) \text{ drf}}$$

$$\frac{F \vdash g_1 \text{ drf} \qquad F \vdash g_2 \text{ drf}}{F \vdash g_1 \oplus g_2 \text{ drf}} \qquad \frac{F \cup \mathsf{AW}(g_2) \vdash g_1 \text{ drf} \qquad F \cup \mathsf{AW}(g_1) \vdash g_2 \text{ drf}}{F \vdash g_1 \otimes g_2 \text{ drf}}$$

$\boxed{F \vdash G \text{ drf}}$

$$\frac{F \vdash g \text{ drf}}{F \vdash [g] \text{ drf}} \qquad \frac{F \vdash g \text{ drf} \qquad F \cup \mathsf{AW}(G_2) \vdash G_1 \text{ drf} \qquad F \cup \mathsf{AW}(G_1) \vdash G_2 \text{ drf}}{F \vdash g \oplus (G_1 \otimes G_2) \text{ drf}}$$

Fig. 7. Definition of determinacy-race-freedom. Variable F denotes a "forbidden" set of locations (that are allocated or updated by a concurrent task).

$F \vdash G$ drf for open computation graphs G. These are defined in terms of another auxiliary function (defined in the Appendix): $\mathsf{AW}(g)$ is the set of locations allocated and written by g.

To see how the forbidden set F is used in the definition, consider the case for parallel composition. In order for $g_1 \otimes g_2$ to be DRF, we need to verify that every location modified by g_2 is not accessed by g_1, and vice-versa. We capture this constraint by extending the set of forbidden locations for g_1 with the allocated and written locations of g_2 (and vice-versa). Then at each individual action, we only need to verify that the accessed location is not forbidden. Note that we do not accumulate forbidden locations in sequential compositions $g_1 \oplus g_2$, because in these cases we know that g_1 and g_2 did not happen concurrently.

Races can violate disentanglement. Intuitively, races can violate disentanglement, because two tasks can communicate by concurrently reading and writing at a shared memory location. For example, consider the program 'let x = ref 0 in $\langle x := 1 \parallel !\, x \rangle$'. This program allocates a shared location ℓ for the ref, and then spawns two subtasks. In one possible execution, the left-hand subtask gets to run completely before the right-hand subtask executes. In this case, the left-hand task allocates a location ℓ' for the value 1 and then writes a pointer to ℓ' at shared location ℓ. Next, the right-hand task executes, reading from ℓ and discovering ℓ', which violates disentanglement. In this situation, there was a race at ℓ.

Although races can violate disentanglement, it is possible to avoid this issue by preallocating any data that might possibly be shared amongst concurrent tasks. That is, we could rewrite the example program as 'let x = ref 0 in let $y = 1$ in $\langle x := y \parallel !\, x \rangle$', which is disentangled. For more details about how to utilize data races in disentangled programs, see Section 3.4.

Race-freedom preserves disentanglement. When races are disallowed, disentanglement is guaranteed, because shared memory locations cannot be used to communicate pointers to freshly allocated locations. Theorem 3.1 establishes this fact. The theorem states that if at any moment we pause a program and observe that it has (so far) been free of determinacy races, then the program also has been disentangled.

Theorem 3.1 (DRF $\Rightarrow$ DE). *For any* $\emptyset\,;[\bullet]\,;e_0 \longmapsto^* \mu\,;G\,;e$ *where* $\mathsf{locs}(e_0) = \emptyset$, *if* $\emptyset \vdash G$ drf, *then* $\emptyset \vdash G$ de.

Proof. The full proof is presented in the Appendix; a sketch of the proof is as follows. Consider this property: "for every leaf task and for every location ℓ known by that task, if ℓ is not forbidden by DRF, then each $\ell' \in \mathsf{locs}(\mu(\ell))$ is known by that task." This property is captured by a judgement $A\,;F \vdash_\mu G\,;e$ drfde, defined in Figure 8, which also implies both $F \vdash G$ drf and $A \vdash G$ de. Initially,

$$\boxed{A\,;F \vdash_\mu G\,;e\ \text{drfde}}$$

$$\frac{F \vdash g\ \text{drf} \qquad A \vdash g\ \text{de} \qquad \text{locs}(e) \subseteq A \uplus \text{A}(g) \qquad \forall \ell \in (A \uplus \text{A}(g)) \setminus F.\ \text{locs}(\mu(\ell)) \subseteq A \uplus \text{A}(g)}{A\,;F \vdash_\mu [g]\,;e\ \text{drfde}}$$

$$\frac{F \vdash g\ \text{drf} \qquad A \vdash g\ \text{de} \qquad \begin{array}{l} A \uplus \text{A}(g)\,;F \cup \text{AW}(G_2) \vdash_\mu G_1\,;e_1\ \text{drfde} \\ A \uplus \text{A}(g)\,;F \cup \text{AW}(G_1) \vdash_\mu G_2\,;e_2\ \text{drfde} \end{array}}{F\,;A \vdash_\mu g \oplus (G_1 \otimes G_2)\,;\langle e_1 \parallel e_2\rangle\ \text{drfde}}$$

$$\frac{A\,;F \vdash_\mu g \oplus (G_1 \otimes G_2)\,;e\ \text{drfde}}{A\,;F \vdash_\mu g \oplus (G_1 \otimes G_2)\,;(\text{fst}\ e)\ \text{drfde}} \quad \textit{...similarly for } (\text{snd}\ e), (\text{ref}\ e), \textit{ and } (!\,e)$$

$$\frac{\begin{array}{l} \neg(e_1\ \text{loc}) \\ \text{locs}(e_2) \subseteq A \uplus \text{A}(g) \end{array} \qquad A\,;F \vdash_\mu g \oplus (G_1 \otimes G_2)\,;e_1\ \text{drfde}}{A\,;F \vdash_\mu g \oplus (G_1 \otimes G_2)\,;(e_1\ e_2)\ \text{drfde}} \quad \textit{...similarly for } \langle e_1, e_2\rangle \textit{ and } (e_1 := e_2)$$

$$\frac{\ell_1 \in A \uplus \text{A}(g) \qquad A\,;F \vdash_\mu g \oplus (G_1 \otimes G_2)\,;e_2\ \text{drfde}}{A\,;F \vdash_\mu g \oplus (G_1 \otimes G_2)\,;(\ell_1\ e_2)\ \text{drfde}}$$

Fig. 8. Strengthening of disentanglement with the guarantees of simultaneous determinacy-race-freedom.

all of these properties hold (of $\mu_0 = \emptyset$, $G_0 = [\bullet]$, and e_0). We prove a single-step lemma that, given $A\,;F \vdash_\mu G\,;e$ drfde, if a step is taken to μ', G', and e' where $F \vdash G'$ drf, then $A\,;F \vdash_{\mu'} G'\,;e'$ drfde. The theorem follows by induction on the derivation of the $\longmapsto^*$ judgement. □

3.4 Disentanglement Permits Some Races

Superficially, it may appear that disentanglement prevents data races, because it does not allow concurrent tasks to have knowledge of each other's allocations. But this is not correct. Disentanglement permits many kinds of races, and is general enough to even permit arbitrary communication in some cases.

To understand the interplay between data races and disentanglement, consider that any race between two concurrent tasks may be classified either as a *write-write* race or a *read-write* race. A ***write-write*** race occurs when both tasks modify the same location, whereas a ***read-write*** race occurs when one of the tasks reads a location that the other task modifies. Write-write races are always safe for disentanglement, because writes can never *discover* new locations (Section 3.2). In the case of read-write races, however, we have to be careful to ensure that the reading task does not discover new locations. That is, a read-write race is disentangled only when the data being written was allocated by a common ancestor. This leads to a simple but powerful observation: as long as all possibly shared data is pre-allocated, disentanglement permits *arbitrary* communication between concurrent tasks.

Examples. The following examples illustrate a number of use-cases for disentangled races.

- In a parallel search, we can use a shared "found-it" flag to quit early once a suitable element has been found. Specifically, we allocate the flag and then begin searching in parallel with many subtasks. When one of the subtasks finds a desirable element, it sets the flag; meanwhile, all subtasks regularly poll the flag to check if they can quit early. Therefore we have a read-write race, but this example is nevertheless disentangled because we allocate the flag before the subtasks begin.

- We can extend the previous example to non-deterministically select one suitable element. To do this, we allocate a mutable pointer and then instruct each subtask to set the pointer to the element it finds (if any). Multiple subtasks might then race the update the pointer (a write-write race), but this is disentangled because none of the subtasks ever read the pointer. Once all subtasks complete, the pointer may be safely dereferenced.
- In graph search algorithms where the number of vertices in the graph is known, we can use one "visited" flag per vertex to guarantee that each vertex is processed at most once. All of these flags must be allocated when the search begins, so that the read-write races on the flags are safe for disentanglement. This technique is used in our evaluation (Section 6.2) in the *bfs* benchmark, where an atomic compare-and-swap is used to visit vertices in parallel.
- We can implement concurrent union-find (dynamic disjoint sets), for example as described in [Blelloch et al. 2012], on a fixed number of nodes. Union-find is a crucial subcomponent in graph algorithms such as minimum-spanning-tree, where one union-find node is used per vertex in the graph. When the number of vertices is known ahead-of-time, all nodes may be allocated at the start of the algorithm. The *mst* benchmark in our evaluation (Section 6.2) uses this approach.
- Any concurrent collection data-structure (such as a queue, stack, hash table, etc.) is safe for disentanglement as long as all data associated with the structure can be pre-allocated. In particular, the size of the collection—including the cumulative sizes of its elements—must be bounded, so that sufficient space can be allocated up-front. Such a collection may then be used by multiple concurrent tasks to communicate freely.

4 MEMORY MANAGEMENT FOR DISENTANGLED PROGRAMS

We now describe a parallel memory management scheme for disentangled, nested-parallel programs. The goal is to be able to manage memory in an efficient and scalable manner by taking advantage of properties guaranteed by disentanglement.

4.1 Preliminaries

A ***memory object***, or simply ***object***, is a contiguous section of memory that is allocated as a unit. Objects may store both non-pointer data (e.g. numbers) and pointers to other objects. During execution, programs allocate new objects and read and write existing objects. The objects of an execution form a ***memory graph*** where vertices are objects and (directed) edges are pointers between objects. Memory graphs evolve over time as the program executes: allocations add new vertices and (possibly) edges, and writes can delete existing edges, replacing them with new edges pointing at different objects.

A ***heap*** is a set of objects. Many heaps can exist simultaneously, but they must be disjoint: each object exists in at most one heap. Heaps are an abstract data type (we describe how to implement them in Section 5) that offer a variety of natural operations: creation of a fresh empty heap, allocation of a new object in a heap, deletion of an object from a heap, and moving an object from one heap to another. We also permit *merging* two heaps (unioning their contents), and querying which heap contains an object. We write $H(x)$ for the heap that contains object x; in general, this is a dynamic query, as objects may be moved between heaps.

The ***roots*** are the set of objects mentioned explicitly by the program state (e.g. in the semantics of Section 2, the roots of expression e are $\mathrm{locs}(e)$). As a program executes, the roots change. Every object in the memory graph is either ***live*** or ***garbage***, depending upon whether or not it is reachable from the roots (by following pointers in the memory graph). As the program executes, live objects may become garbage by either (a) dropping a root, or (b) deleting an edge of the memory graph

(with an update). Garbage objects will never again be used by the program, and so they may be de-allocated (reclaimed) by deleting them from their corresponding heaps. The goal of ***garbage collection*** is to reclaim space occupied by garbage objects.

4.2 Heap Hierarchy

We give each task its own heap and organize heaps into a tree that mirrors the task tree (Section 2.3). We call this tree of heaps the ***heap hierarchy***. New objects are placed in the heap of the task that performed the allocation. When a task forks, its subtasks are initialized with two fresh (empty) heaps and, when both subtasks of a task complete, their heaps are merged with the heap of the parent task. This puts heaps and tasks in a one-to-one correspondence.

In the heap hierarchy, we can use the ancestor/descendant relationships of heaps to give each memory graph pointer a direction: *up*, *down*, or *cross*. A pointer from object x to object y is classified as follows: if $H(x)$ is a descendant (inclusive) of $H(y)$, the pointer is an ***up-pointer***; if $H(x)$ is a proper ancestor of $H(y)$, it is a ***down-pointer***; otherwise, it is a ***cross-pointer***.

Relationship to Computation Graphs. The heap hierarchy directly implements the structure of allocations in an open computation graph G, where the memory locations of Section 2 are used as object identifiers. We derive the heap hierarchy corresponding to G as follows: if $G = [g]$ then it is just the single heap containing the objects $\mathsf{A}(g)$, otherwise the heap hierarchy of $G = g \oplus (G_1 \otimes G_2)$ is a heap containing the objects $\mathsf{A}(g)$ with two children which are the heap hierarchies of G_1 and G_2, respectively.

We can see that this correspondence between the heap hierarchy and an open computation graph is correct by examining forks, joins, and allocations. Forks are witnessed by replacing a leaf $[g]$ with $g \oplus ([\bullet] \otimes [\bullet])$, which is implemented by creating two empty heaps, corresponding to the new leaves $[\bullet]$. Joins occur when a graph $g \oplus ([g_1] \otimes [g_2])$ is replaced by $[g \oplus (g_1 \otimes g_2)]$; this corresponds to three heaps $h = \mathsf{A}(g)$, $h_1 = \mathsf{A}(g_1)$, and $h_2 = \mathsf{A}(g_2)$ being merged into a single heap $h \uplus h_1 \uplus h_2 = \mathsf{A}(g \oplus (g_1 \otimes g_2))$. Finally, for each allocation, a leaf $[g]$ is replaced by some $[g \oplus (\mathbf{A}\ell \Leftarrow s)]$. Since tasks store locally allocated data in their own heaps, this corresponds to extending the heap $\mathsf{A}(g)$ with a fresh location ℓ, forming a heap $\mathsf{A}(g) \uplus \{\ell\} = \mathsf{A}(g \oplus (\mathbf{A}\ell \Leftarrow s))$.

4.3 Guarantees of Disentanglement

Disentanglement provides a strong guarantee on the directions of the pointers in the memory graph: every pointer is either an up-pointer or a down-pointer (Property 1). Furthermore, disentanglement guarantees that the roots of the program only point up (Property 2).

Property 1. *Throughout execution of a disentangled program, all pointers in the memory graph are either up-pointers or down-pointers.*

Property 2. *Throughout execution of a disentangled program, for every task, every root of that task lies within either its own heap or an ancestor heap.*

Formally stating these properties can be done in a manner similar to the drfde judgement. In particular, in Figure 8, the component highlighted in blue is essentially the statement of Property 1 (one would only need to eliminate the use of F which is specific to drfde), and the components highlighted in red capture Property 2. A formal proof can then proceed in a manner similar to the proof of Theorem 3.1. The gist of the proof is as follows. Initially when there are no allocated objects, both the memory graph and roots are empty, so both properties hold initially. We have Property 1 throughout the execution of a disentangled program because (a) allocations can only create up-pointers in the memory graph (because new allocations are always in the task's heap and by Property 2 the locations of the newly allocated storable lie within the task's heap or ancestor

heaps), (b) writes can only create either up-pointers or down-pointers (again, because by Property 2 the written-to location and the stored location both lie within the task's heap or ancestor heaps), and (c) heap merges can only cause down-pointers to become up-pointers (and therefore cannot introduce cross-pointers). We have Property 2 throughout the execution of a disentangled program because new allocations are always in leaf heaps, and because at each read we are guaranteed by Property 1 that any newly obtained pointers are into the task's heap or ancestor heaps.

4.4 Parallel Garbage Collection

In this section we describe an algorithm called ***subtree collection***, where a ***subtree*** consists of a heap and all of its descendants. As the name suggests, subtree collections are localized to a subtree of the heap hierarchy.

Utilizing Disentanglement. When performing collection on only a small region of the memory graph, it is necessary to find all incoming pointers (x, y) from live objects x outside the region to objects y inside the region, so that the set of live objects inside the region can be determined. In general however, knowing the set of live objects outside the region requires tracing the entire memory graph, which defeats the goal of a localized collection (cheaper collection with smaller scope). A common simplification made is to assume that all incoming pointers are live, which makes it possible to perform collection locally without needing to trace the entire memory graph (at the potential cost of preserving some dead objects). In our case, disentanglement guarantees that all incoming pointers into a subtree are *down-pointers*. This is because of Property 1 and the fact that any up-pointer into a subtree must have originated from within the subtree.

The fact that all incoming pointers into a subtree are down-pointers has multiple benefits. First, it means that in order to perform subtree collection, we only need to remember down-pointers. But more importantly, it means that a subtree collection only needs to access objects within or above the subtree. Since in a nested-parallel program, all ancestor tasks are suspended, this results in *independence* of subtree collections, which in turn enables a conceptually simple parallel garbage collection strategy: perform many subtree collections simultaneously across the hierarchy.

Subtree Collection. Pick a subtree, and let T be the set of heaps that lie within the subtree. We say that an object x is *in-scope* if $H(x) \in T$; otherwise, x is *out-of-scope*. Note that during collection, objects may be moved to different heaps, in which case an object that originally was in-scope may become out-of-scope. In order to preserve disentanglement, objects will only ever be moved upwards in the hierarchy. Subtree collection proceeds in two phases.

(1) A *promotion* phase eliminates down-pointers by moving objects upwards in the hierarchy. Promotion is motivated by efficiency: an object y which is referenced by an out-of-scope object x cannot be reclaimed by a subtree collection. Taking inspiration from generational collectors [Appel 1989; Lieberman and Hewitt 1981; Ungar 1984], rather than let such an object y persist through multiple collections, we can instead *promote* it to a higher heap which is collected less often. In this way, down-pointers are analogous to inter-generational pointers from old objects to young objects. By delaying the promotion of objects until garbage collection, promotion becomes very cheap, as the promotion of many objects can be batched and any performance artifacts of promotion can be hidden from the mutator program. In particular, our implementation (Section 5) does *not* need a mutator read barrier, which is crucial for efficiency.

(2) A *tracing* phase identifies the set of *survivors* S within the subtree that are reachable from the roots. Any object which is not a survivor is garbage.

Once the tracing phase has completed, we finally de-allocate the garbage objects $\{x \notin S \,|\, H(x) \in T\}$, and then subtree collection is complete. We now describe the promotion and tracing phases in detail.

Promotion phase. Promotion proceeds by performing the following.

(1) Let D be the set of candidate down-pointers (x, y) where $H(x) \notin T$ and $H(y) \in T$. (If there are no such down-pointers, promotion is complete.)
(2) Pick $(x, y) \in D$ where $H(x)$ is shallowest amongst $\{H(x') \mid (x', y') \in D\}$.
(3) Promote y by moving it to $H(x)$. (This promotion may create new down-pointers, including candidate down-pointers.)
(4) Repeat.

Once promotion completes, there are no more down-pointers to in-scope objects from out-of-scope objects. Note that it is possible for there to be new down-pointers from promoted objects to out-of-scope objects, after promotion completes. However, promotion cannot create cross-pointers, because it only moves objects upwards in the hierarchy.

The order in which promotion processes down-pointers is important for efficiency: by operating from top to bottom, we guarantee that each object is promoted at most once. In particular, in step 2 of the promotion phase, it is crucial that $H(x)$ is shallowest amongst *all* candidate down-pointers. This guarantees, in chains of down-pointers, that the objects in the chain are promoted in order of shallowest to deepest. Otherwise, the deepest objects in the chain could be promoted multiple times.

Tracing phase. The tracing phase begins with the initial set of survivors $S \leftarrow \{x \in R | H(x) \in T\}$, i.e. the set of in-scope roots. Tracing proceeds by performing the following:

(1) Pick a pointer (x, y) where $x \in S$ and $y \notin S$ and $H(y) \in T$. (If there are no such pointers, tracing is complete.)
(2) Insert y into S.
(3) Repeat.

Once tracing completes, the set S contains all live in-scope objects. After tracing, subtree collection completes by reclaiming the objects $\{x \notin S \mid H(x) \in T\}$.

Example. An example subtree collection is shown in Figure 9. In this example, there are five heaps depicted as large rectangles, and the three bottom-most heaps are in-scope for collection. The small squares are objects, the diamonds are root objects, and the arrows are pointers between objects. During collection, the highlighted groups of objects **A** and **B** are promoted to the topmost heap, and the group **C** is reclaimed.

Correctness. We now argue that subtree collection never reclaims an object that is reachable from the roots. Consider some live object x where initially $H(x) \in T$. There are two cases: either x is promoted to a heap outside the subtree, or it is not. In the former case, x will not be reclaimed because it becomes out-of-scope. In the latter case, consider that due to the lack of cross-pointers, after promotion completes, we have the guarantee that every path in the memory graph which ends at x is entirely contained within the subtree. Because we assumed that x is live, we know there exists a particular path $x_1, \ldots, x_n$ where x_1 is a root and $x_n = x$. This path ends at x, and so $H(x_i) \in T$ for each i. Since $H(x_1) \in T$ and x_1 is a root, we know that $x_1 \in S$ initially in the tracing phase. Therefore once tracing completes, we also know every $x_i \in S$ (because the path is contained within the subtree), including $x = x_n \in S$. No objects in S are reclaimed, so x is not reclaimed.

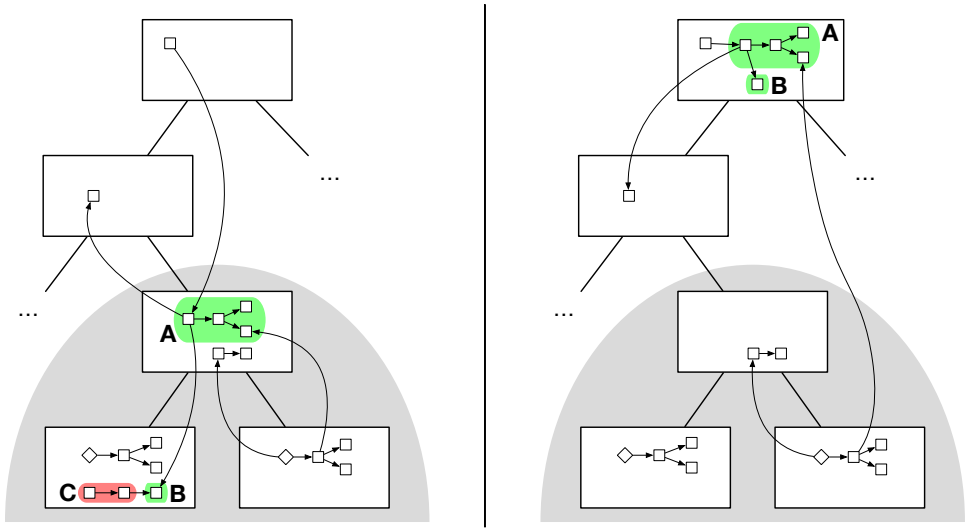

Fig. 9. Before (left) and after (right) an example subtree collection of the bottom-most three heaps. The large rectangles are heaps, the squares are objects, and the diamonds are root objects.

Independence of Subtree Collections. A subtree collection (consisting of both promotion and tracing phases) only accesses objects within the subtree or within ancestor heaps of the subtree, and furthermore only moves objects that lie within the subtree. This means that any two disjoint subtrees—that is, any two subtrees with no heaps in common—may be collected independently and in parallel, because any shared ancestors are guaranteed to be outside the scope of both collections. One subtlety is that two concurrent collections may promote two different objects into the same shared ancestor heap at the same time, however this scenario does not harm independence, because (a) insertions commute, and (b) neither collection will attempt to access the other's promoted objects. Subtree collections, in addition to being independent of other disjoint collections, are also independent of the actions of concurrent tasks. That is, a subtree collection may be performed locally upon the subtree without interrupting tasks that own heaps outside the subtree.

5 IMPLEMENTATION

We implemented our techniques by extending the MLton [MLton [n.d.]] whole-program optimizing compiler. Our implementation, which we call MPL, is available on GitHub at https://github.com/mpllang/mpl. This was a multiple person-years effort and we have in the process updated many parts of the compiler, with most of the effort focusing on two components: scheduling and memory management. The scheduler handles load-balancing, and the memory manager handles low-level aspects such as allocation and garbage collection.

For the programmer, we provide a primitive **par** : $(\texttt{unit} \to \alpha) \times (\texttt{unit} \to \beta) \to \alpha \times \beta$ which takes two functions as argument, executes them in parallel, and returns their results. The parallel pair $\langle e_1 \parallel e_2 \rangle$ of Section 2 may then be translated into Parallel ML by calling **par** with two thunks.

Our implementation is faithful to the semantics of Section 2, with one important difference. For ease and simplicity of presentation, the formal semantics explicitly allocates a memory location for *all* data, even "small" types such as integers, which is not realistic. Practical compilers such as MLton perform flattening optimizations which change the memory representations of objects in order to eliminate unnecessary allocations. For example, while an object of type $(\texttt{int} \times \texttt{int})\ \texttt{array}$ *could* be

represented by an array of pointers to tuples, it is likely better to use an "array-of-structs" layout, avoiding the use of pointers entirely. Such flattening optimizations are crucial for efficiency in many programs. Furthermore, as long as flattening only eliminates allocations, these optimizations appear to be safe for disentanglement. We therefore leave the flattening optimizations turned on. We have encountered no correctness issues due to flattening in our experiments and benchmarks.

Scheduling. We implemented a work-stealing scheduler [Blumofe and Leiserson 1999] with private deques [Acar et al. 2013]. The scheduler maps "user" threads (one-shot continuations, implemented as heap-allocated call-stacks) onto "worker" threads (OS threads, specifically `pthreads`). We use one worker thread per processor. Initially, there is a single user-thread being executed by one of the worker threads. At each steal, the scheduler creates a new user-thread to execute the stolen work. The scheduler coordinates with the runtime in order to create new heaps and merge existing heaps, as tasks fork and join.

Heap Implementation. We logically divide the virtual memory space into fixed-size ***blocks*** of 2^k bytes (we use $k = 12$), appropriately aligned. Each worker-thread recycles blocks in a local freelist, and requests new blocks from the OS when the freelist is exhausted. Blocks are linked into doubly-linked lists to form heaps. This strategy makes it possible to merge two heaps without copying any data: instead, we merge two heaps simply by linking together their two block-lists, which takes constant time. This strategy also makes it possible to permit concurrent promotions into the same heap, as each promotion can reserve blocks in which to store promoted objects. In order to query which heap contains an object, we associate with each heap block a *descriptor*. The descriptor is located at the front of the block, which makes it possible to find the descriptor for any object by zeroing the low-order bits of the object's memory address. In the descriptor, we include a pointer to its parent heap. Since heaps are merged dynamically, this is an instance of the union-find problem; we therefore maintain a disjoint-set data structure with path-compressing parent pointers, offering effectively constant-time heap queries [Tarjan 1975].

Remembered Sets and Write Barriers. We equip each heap with a depth and a *remembered set* in order to efficiently implement the subtree collection algorithm described in Section 4. The depth is simply the depth of the heap in the heap hierarchy, which is easily maintained at forks and joins. The remembered sets store entries of the form (x, i, y), indicating that field $x[i]$ (offset i of object x) might hold a down-pointer to y. Remembered sets are maintained by a *write barrier*, which is a small piece of code that inserted in the compiled program before certain writes to memory. Our implementation has a write barrier for every update of pointer data that might result in a down-pointer. At the write barrier for the update '$x[i] \leftarrow y$', we compare the depths of $H(x)$ and $H(y)$: if $H(y)$ is deeper than $H(x)$, then we "remember" the down-pointer by allocating the entry (x, i, y) in the remembered set for $H(y)$. Note that disentanglement guarantees that x is either an ancestor or a descendant of y, which is why we can determine their relative position in the hierarchy simply by comparing their depths.

Garbage Collection. Our collection strategy consists of letting many *local collections* run in parallel. A ***local collection*** is a specific kind of subtree collection which operates only on heaps owned by a single worker in the scheduler. In this way, our collection policy is integrated with scheduling algorithm. We implemented subtree collection by adapting a Cheney-style copying collector [Cheney 1970]. For simplicity here we say that objects are "moved" between heaps, but in reality they are copied and all references to them must be updated. This is accomplished in the same way as in a Cheney collection, by evacuating pointers that point into the from-space and installing forwarding pointers so that references to old objects may be updated. The promotion phase proceeds by performing the following two steps for each d progressing from shallow to deep.

(1) For each remembered (x, i, y) of an in-scope heap where $\text{depth}(x) = d$, if $x[i]$ currently points at y, promote y to depth d and update $x[i]$ to point to the new version of y.
(2) Repeatedly promote (to depth d) any object z which is in a local heap at depth strictly greater than d and is pointed to from a promoted object.

During this process, we edit the remembered sets at out-of-scope heaps to remember down-pointers created during promotion. The tracing phase of a local collection performs a Cheney-style collection on each in-scope heap, beginning at the deepest local heap and progressing to the shallowest local heap. This guarantees that when a heap is processed, all up-pointers into the heap have already been evacuated. Additionally, the tracing phase updates all references to objects copied during promotion.

6 EVALUATION

6.1 Methodology

Our implementation, which we call MPL, consists of many major modifications to the underlying MLton compiler and runtime system, raising numerous interesting empirical questions. Our goal here is not to present a detailed analysis of the engineering and implementation decisions, which is outside scope of this paper, but to present an evaluation of the benefits of our approach by using well established parallel benchmarks, and by comparing with other programming languages and systems. For all of our comparisons, we use highly optimized codes from the existing literature and use similarly optimized Parallel ML implementations.

We start by presenting a relatively broad evaluation of the overheads and scalability of our techniques by considering a variety of parallel benchmarks ported to Parallel ML from the state-of-the-art Problem Based Benchmarking Suite [Blelloch et al. 2012; Shun et al. 2012]. We then zoom into a single classic problem, sorting, and present the results of a "sorting competition", where we consider a variety programming systems for nested parallelism. Finally, we compare our MPL to other implementations of Parallel ML, including Manticore [Fluet et al. 2011, 2007] and Guatto et al.'s implementation [Guatto et al. 2018]. These comparisons collectively demonstrate that

- MPL scales well and delivers low overheads compared to sequential Standard ML,
- MPL can outperform memory-managed procedural languages,
- MPL outperforms prior Parallel ML implementations, which support only purely functional programs or a narrow range of effects.

Experimental Setup. We run all of our experiments on a 72-core Dell PowerEdge R930 consisting of 4 × 2.4GHz Intel 18-core E7-8867 v4 Xeon processors and 1TB of memory. Each benchmark is run 10 times and we report the average. The timing results exclude initialization (e.g. loading the input) and teardown. In the sorting competition (Section 6.3), we use the following settings. For C++, we compile with GCC 6.4 with CilkPlus and `-O3 -march=native`. For Java, we compile with OpenJDK `1.8.0_222` and run with `-XX:+UseParallelGC`. To account for HotSpot warmup, we run Java benchmarks 15 times and exclude the first 5 runs. For Go, we compile with version `1.8.1` and use default settings.

6.2 PBBS Benchmarks

We consider the Problem-Based Benchmark Suite (PBBS), which, since its conception in 2012, has been developed and optimized by many researchers [Blelloch et al. 2012; Shun et al. 2012]. PBBS benchmarks use state-of-the-art nested-parallel algorithms designed for modern multicore hardware and rely extensively on side effects for improved efficiency, with almost all benchmarks implemented in C/C++. We selected benchmarks by considering a range of common problem areas,

including sequences, matrices, trees, and graphs. Because race-freedom is a desirable correctness condition in parallelism, we found that many PBBS benchmarks are determinacy-race-free and therefore disentangled (Theorem 3.1). In other cases, data races were utilized for improved efficiency, but upon closer inspection these racy benchmarks still turned out to be disentangled (because, as discussed in Section 3.4, many kinds of races respect disentanglement). Overall, every benchmark we considered was either already disentangled or could be easily made disentangled with small tweaks. We therefore were able to translate a variety of PBBS benchmarks into Parallel ML. We did not exclude any benchmarks from our selection due to entanglement, but neither did we systematically evaluate all of PBBS.[2]

Our selection of benchmarks consists of comparison-based sorts (samplesort, mergesort), dense matrix multiplication (dmm), deduplication (dedup), histogram, barnes-hut, nearest-neighbors (all-nearest), minimum spanning tree (mst), and breadth-first search (bfs). All of these codes are determinacy-race-free except for mst and bfs, which utilize races in a manner that respects disentanglement. We also include the standard parallel Fibonacci benchmark (fib) to contrast compute-bound with memory-bound benchmarks. When translating into Parallel ML, we remained faithful to the C/C++ implementations to the extent possible, except for appropriate details to accommodate our different run-time system such as sequential granularity thresholds.

Inputs. The benchmarks samplesort, mergesort, dedup, and histogram each take an array of 100M 32-bit (uniformly) random integers. Dense matrix multiplication is on two 2048×2048 matrices of 64-bit floats. Barnes-hut takes 1M point-masses and nearest-neighbors takes 2M points, both distributed uniformly randomly within a square (2-dimensional points). The minimum spanning tree benchmark is on the *orkut* social network graph [sna [n.d.]], which has approximately 3M vertices and 117M edges. On the orkut graph, we applied uniform random edge weights in the range $[0, \log_2 n]$ where n is the number of vertices. Breadth-first search is on the *twitter* social network graph [Kwak et al. 2010], which has approximately 42M vertices and 2.4B edges, symmetrized.[3]

Results. Figures 10 and 11 show the results. The column T_s shows run-time for the sequential baseline which is compiled with the MLton compiler, on which we base our MPL implementation. When compiling with MLton, we use the sequential elisions of the Parallel ML benchmark implementations. The column T_1 shows the run-time for the MPL version on a single processor and the column "overhead" shows the ratio T_1/T_s, which captures the overhead of parallelism over sequential computation. For a majority of our benchmarks, the overhead is less than 20%, for `histogram` the overhead is 40%, and for `dmm` the overhead is about a factor of two. These overheads are in the same ball-park as with optimized C implementations that have recently been analyzed on similar machines (e.g., [Acar et al. 2018b, 2015b; Blelloch et al. 2012]), and show that our techniques manage parallelism effectively. The column T_{72} shows the 72-core run-time, and the "speedup" column is the ratio T_s/T_{72}, which is the improvement relative to sequential baseline. Speedups range between approximately 17 and 55. As in the original C-implementations [Blelloch et al. 2012]., we observe that the difference in speedups between different benchmarks mirror their memory access patterns. For example, `fib` is a purely computational benchmark with an effectively empty working set, and scales very well, whereas the graph algorithms such as `bfs` and `mst` are highly irregular and perform irregular memory accesses as they traverse the edges of the graph, causing them to become memory bound as the number of cores increase.

Space Efficiency. We measured the memory footprint of both MLton and MPL on the PBBS benchmarks by collecting the maximum resident set size of each benchmark, as reported by Linux.

[2]The original PBBS presented results for 14 benchmarks [Shun et al. 2012], but more benchmarks have been added since.
[3]For each arc (x, y) in the original graph, the symmetrized graph additionally has (y, x) if it is not already present. The original graph has approximately 1.5B directed edges [Kwak et al. 2010].

	MLton	MPL (Ours)			
	T_s	T_1	over head	T_{72}	speed up
fib	4.3	4.4	1.0	.077	55.8
samplesort	17.3	19.5	1.1	.38	45.5
mergesort	17.0	18.8	1.1	.37	45.9
dmm	12.1	25.9	2.1	.5	24.2
dedup	5.6	6.1	1.1	.16	35.0
histogram	9.1	12.3	1.4	.31	29.4
barnes-hut	6.9	8.0	1.2	.23	30.0
all-nearest	3.1	3.8	1.2	.16	19.4
mst	19.4	17.1	.88	1.1	17.6
bfs	9.1	11.2	1.2	.53	17.2

Fig. 10. Problem-based benchmarks: execution times (in seconds), overheads on 1 processor, and speedups on 72 processors, relative to MLton.

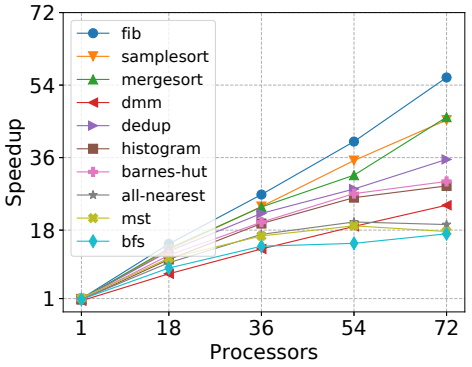

Fig. 11. Problem-based benchmarks: speedups relative to MLton.

We found that in almost all cases, MPL uses at most twice as much as space as MLton. The only exception is the `fib` benchmark on 72 cores, which due to having a very small footprint overall (MLton uses 2MB, and MPL uses 3MB on 1 core and 37MB on 72 cores), requires more memory because of the concurrent threads of execution, each of which needs its own control stack. These results suggest that the space overheads of our techniques can be low. However, a more in-depth evaluation is required to draw any further conclusions. In particular, we have observed that MPL is unable to effectively collect garbage for some workloads. This is due to a limitation of subtree collection, which we discuss in Section 8.

6.3 Sorting Competition

Sorting has been a classic problem for evaluating the effectiveness of parallel programming techniques. Here, we report results from a "sorting competition" where we have included state-of-the-art, highly optimized, codes from a relatively broad array of programming languages and systems, including Cilk (based on C), Java, Go, Haskell, and our own MPL. The input to be sorted in all cases is an array of 100M 32-bit uniformly random integers, generated by a hash function, and we require that the input is not modified (the sort must allocate the result in a new array). For a sequential baseline, we use the C++ `std::sort`. The Cilk implementations, including both highly optimized samplesort and mergesort, are taken from the PBBS benchmark suite. The Java implementation is the standard `java.util.Arrays.parallelSort`, written by Doug Lea for the Java Fork/Join library. The Go implementation is a highly optimized samplesort derived from the PBBS samplesort implementation. The Haskell implementation is taken from Kuper et al's artifact accompanying their PLDI paper [Kuper et al. 2014a]; it implements a parallel merge sort and is optimized to call out to subroutines written in C for the sequential sort used to sort small inputs and also for merging the sorted results from subcalls.

Figure 13 show the execution times for each implementation, and Figure 12 shows the speedups for each implementation. The results show that the Cilk samplesort scales very well, outperforming Go and Java by at least a factor of three on 72 cores. Considering the fact that both Go and Java are procedural and object-oriented languages, we attribute the difference mainly to the costs of automatic memory management. Compared to Haskell, the Cilk samplesort is nearly an order of magnitude faster on 72 cores. Our Parallel ML mergesort compiled with MPL performs best among the memory managed languages and second only to Cilk, consistently performing only

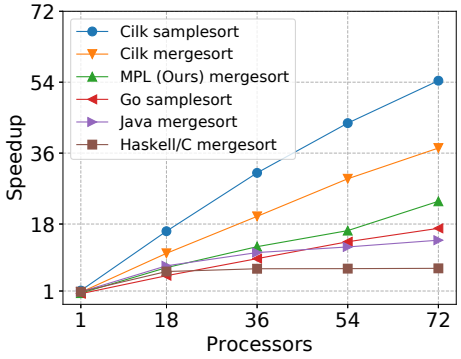

Fig. 12. Sorting competition: speedups relative to C++ std::sort.

	T_1	T_{72}
C++ std::sort	8.8	–
Cilk samplesort	7.9	0.16
Cilk mergesort	12.7	0.24
MPL (Ours) mergesort	18.8	0.37
Go samplesort	27.2	0.52
Java mergesort	11.0	0.63
Haskell/C mergesort	10.6	1.3

Fig. 13. Sorting competition: execution times (in seconds).

	1 Core Time		72 Core Time	
	T_1	$\frac{\text{Manticore}}{\text{Ours}}$	T_{72}	$\frac{\text{Manticore}}{\text{Ours}}$
fib	7.7	1.8	0.13	1.7
tabulate	64.1	21.4	2.7	43.3
map	54.5	20.9	3.0	53.0
filter	32.8	4.8	1.5	11.5
reduce	6.4	3.4	0.18	5.0
scan	66.6	8.3	3.0	20.4

Fig. 14. Manticore times (in seconds) and ratios to our system.

	1 Core Time		72 Core Time	
	T_1	$\frac{\text{Guatto}}{\text{Ours}}$	T_{72}	$\frac{\text{Guatto}}{\text{Ours}}$
usp-tree	22.2	1.8	17.9	40.7
strassen	2.5	1.2	.1	1.6
tourney	5.7	1.5	.25	1.6
msort	3.3	1.2	.13	1.2
dedup*	3.2	1.2	.12	1.3
raytracer	6.5	1.0	.12	0.71

* Not the same benchmark as the PBBS dedup.

Fig. 15. Guatto et al. times (in seconds) and ratios to our system on their benchmarks.

approximately 50% slower than the Cilk mergesort across all core counts. On 72 cores, MPL is 40% faster than Go, 70% faster than Java, and 350% faster than Haskell/C.

6.4 Comparison to Other Parallel ML Implementations.

There are two other closely related implementations of Parallel ML: Manticore and Guatto et al.'s implementation [Guatto et al. 2018]. The Manticore project offers a ground-up implementation of Parallel ML that is specifically optimized for purely functional code. Under development for over a decade now, Manticore is a relatively mature project and generally exhibits excellent scalability [Fluet et al. 2011, 2007; Raghunathan et al. 2016].

Because Manticore is primarily aimed at purely functional programs, we are not able to use PBBS benchmarks in Manticore. For this comparison, we therefore limit ourselves to smaller sequence primitives such as `map, reduce, scan`, which Manticore system provides as part of its basis library. Our MPL implementations use arrays under the hood but do not side-effect their inputs. All the sequence benchmarks in this comparison operate upon 500M 32-bit integers. Figure 14 shows that MPL incurs significantly less overhead on a single core, with Manticore requiring 10-fold more time on average. On 72 cores, the gap increases with MPL performing as much as 50× faster. The performance gap is due to Manticore spending a significant amount of time in garbage collection and promotion. In contrast, our MPL system allocates significantly less data, avoids promotion completely, and in general is able to remain much more lean and efficient by utilizing effects.

In a separate line of work Guatto et al. have developed a Parallel ML compiler by extending the MLton compiler. Their implementation builds on the work of Raghunathan et al. [Raghunathan et al. 2016], which supports only pure functional programs, extending it to support certain kinds

of local effects, such as those that can be confined to sequential sections of the code. All of these "local" effects supported efficiently by Guatto et al. are trivially disentangled, allowing us to run their benchmarks directly on MPL. Figure 15 shows a comparison by using their benchmark codes. We observe that in almost all cases, MPL is faster both on 1-core and 72-core runs. On the `usp-tree` benchmark, our MPL is 40-fold faster on 72 cores. This benchmark uses more general disentangled effects which are "non-local", causing Guatto et al.'s implementation to suffer from high overheads. This comparison shows that MPL generally outperfroms Guatto et al.'s work for benchmarks that use local effects and can outperform it very significantly when using more general effects.

7 RELATED WORK

There has been much work on designing parallel programming languages based on procedural, object-oriented, and functional programming languages. Systems extending C/C++ include Cilk/-Cilk++ [Blumofe et al. 1995; Frigo et al. 2009; Intel Corporation 2009a], Intel TBB [Intel Corporation 2009b], and Galois [Kulkarni et al. 2007; Pingali et al. 2011]. The Rust language offers a type-safe option for systems-level programming [Rust Team 2019]; the type system of Rust is powerful enough to outlaw races statically [Jung et al. 2018a]. Systems extending Java include Fork-Join Java [Lea 2000], deterministic parallel Java [Bocchino, Jr. et al. 2009], and Habanero [Imam and Sarkar 2014]. X10 [Charles et al. 2005] is designed with concurrency and parallelism from the beginning and supports both imperative an object-oriented features.

All of these systems support memory effects or destructive updates, which make it challenging to write correct parallel programs, because they can lead to determinacy or data races [Allen and Padua 1987; Emrath et al. 1991; Mellor-Crummey 1991; Netzer and Miller 1992; Steele Jr. 1990], which can be very difficult to avoid and usually lead to incorrect behavior [Adve 2010; Bocchino et al. 2011, 2009; Boehm 2011]. There has therefore been much work on ensuring race freedom by detecting or eliminating races via dynamic techniques (e.g., [Cheng et al. 1998; Feng and Leiserson 1999; Kuper and Newton 2013; Kuper et al. 2014b; Mellor-Crummey 1991; Raman et al. 2012; Steele Jr. 1990; Utterback et al. 2016], as well as static techniques including type systems (e.g., [Bocchino et al. 2011; Flanagan and Freund 2009; Flanagan et al. 2008]). More generally, verifying properties of concurrent programs has emerged as an active research area, and in particular many variants of separation logic have been developed (e.g., [Bizjak et al. 2019; Jung et al. 2018b; Reynolds 2002; Turon et al. 2013; Vafeiadis and Parkinson 2007]).

Another class of research considers functional programming languages and extends them to support parallel programming. Notable works include several forms of a Parallel ML language [Acar et al. 2015a; Fluet et al. 2008, 2011; Guatto et al. 2018; Raghunathan et al. 2016], the MultiMLton project [Sivaramakrishnan et al. 2014; Ziarek et al. 2011], the SML# project [Ohori et al. 2018], and the work on several forms of Parallel Haskell [Chakravarty et al. 2007; Keller et al. 2010; Marlow 2011]. Because purely functional programs don't use effects, they avoid race conditions but this comes at the cost of efficiency. There has been significant research on understanding the interaction between functional programming and effects [Gifford and Lucassen 1986; Kuper and Newton 2013; Kuper et al. 2014a; Launchbury and Peyton Jones 1994; Lucassen and Gifford 1988; Park et al. 2008; Peyton Jones and Wadler 1993; Reynolds 1978; Steele 1994; Terauchi and Aiken 2008]. This research shows that type systems for functional programming languages can support disciplined use of effects, and enable reasoning about correctness.

Even though there has been significant research on memory-managed and functional programming languages, parallel programming continues to be a big challenge. Functional languages—and more generally high-level, memory-managed languages—hold the promise of simplifying the task of writing parallel programs, but this usually comes at the cost of significant loss of efficiency, especially in functional languages. The results in this paper show that efficient parallel functional

programming is feasible by taking advantage of the invariants offered by the more precise control over effects that they offer.

Nearly all high level languages today support automatic memory management and numerous techniques for incorporating parallelism, concurrency, and real-time features into memory managers have been developed. Jones et al. [Jones et al. 2011] provides an excellent survey. Here, we contrast the disentanglement-based memory management techniques proposed in Section 4 with prior systems that use processor-local or thread-local heaps combined with a shared global heap that must be collected cooperatively [Anderson 2010; Auhagen et al. 2011; Doligez and Gonthier 1994; Doligez and Leroy 1993; Domani et al. 2002; Marlow 2011].

The Doligez-Leroy-Gonthier (DLG) parallel collector [Doligez and Gonthier 1994; Doligez and Leroy 1993] employs this design, with the invariant that there are no pointers from the shared global heap into any processor-local heap and no cross pointers between processor local-heaps. To maintain this invariant, all mutable objects are allocated in the shared global heap and (transitively reachable) data is promoted (copied) from a processor-local heap to the shared global heap when updating a mutable object. This approach penalizes allocations and updates for mutable data and thus increases the cost of common scheduling and communication actions, such as migrating a user-level thread or returning the result of a child task.

The Manticore garbage collector [Auhagen et al. 2011] is a variant of the DLG design, where the Appel semi-generational collector [Appel 1989] is used for collection of the processor-local heaps. As with the DLG design, Manticore collector can incur large promotion overheads. Recent work [Le and Fluet 2015] has considered extending the Manticore language with mutable state via software transactional memory, but observed that promotions lead to efficiency problems.

The two-level hierarchical model that does not allow pointers from the global to the local heaps incur large overheads when an object allocated locally must be shared, which can happen often in nested-parallel programs due to scheduling actions, which migrate tasks between processes or workers. Adaptations of the two-level model to concurrent and parallel systems therefore devised techniques to relax this invariant. For example, the Glasgow Haskell Compiler (GHC) uses a garbage collector [Marlow 2011] that allows pointers from global to local heaps and relies on a read barrier to promote (copy) data to the global heap when accessed. Although Haskell is a pure language, there are significant side effects due to lazy evaluation. GHC therefore combines elements of the DLG and Domani et al. [Domani et al. 2002] collectors for improved handling of side effects. There has also been important work on a multicore-suitable run-time system and memory manager for OCaml over the past decade. The work is currently ongoing, but the OCaml team appears to be considering a design similar to that of Haskell [Sivaramakrishnan and Dolan 2017].

In contrast to these prior approaches, in our work, we associate heaps with tasks rather that system-level threads or processors. The result is a dynamic hierarchy and that mirrors the structure of the computation. The hierarchy can be arbitrarily deep in principle and grows and shrinks as the computation proceeds. To support sharing, we allow pointers between heaps that have ancestor-descendant relationships. For example, a heap can point to an object allocated in its parent, and a parent can point to an object allocated in its children. The only kind of pointer that is not allowed is a cross-pointer between concurrent heaps. This approach enables taking advantage of important properties of parallel programs, e.g., we can return the result of a child task and migrate threads without copying (promoting) data, and concurrent threads can share the data allocated by their ancestors, and disentangled effects do not require an immediate promotion of data.

In the sequential setting, region-based memory management [Hanson 1990; Ross 1967; Schwartz 1975; Tofte and Talpin 1997] shares some similarities with hierarchical heaps. In a region-based system, a program dynamically creates and destroys *regions*, into which individual objects may be allocated; thus, regions (only) support bulk deallocation (but also supporting garbage collection

has been considered [Elsman 2001]). Statically-scoped regions [Grossman et al. 2002; Tofte and Talpin 1997] are organized as a stack, while dynamically-scoped regions [Grossman et al. 2002; Hanson 1990] impose no particular relationship between regions. In general, pointers from one region to any other region are supported; a type-and-effect system [Tofte and Talpin 1997] or linear types [Fluet et al. 2006; Walker 2001] can be used to guarantee that pointers into deallocated regions will never be followed. The flexibility of allocating new objects into any available region allows for arbitrary memory graphs and avoids the need to promote objects from one region to another, but with the overhead of explicitly managing the set of available regions, rather than implicitly having a single allocation frontier.

Nearly all of the work reviewed above relies on the idea of organizing memory as a hierarchy of heaps, some shallow like most other work, and some possibly deep, like our work. The general idea of hierarchical heaps goes back to 1990s. Early approaches in procedural languages such as Split-C [Krishnamurthy et al. 1993], Co-Array Fortran [Numrich and Reid 1998], and Titanium [Yelick et al. 1998], differentiate between memory that is local and remote to a thread. Alpern et al. developed abstract models of uniprocessor and multiprocessor machines as hierarchies of memories [Alpern et al. 1990]. More recently, the technique was employed in the Sequoia language, which allows the programmer to designate tasks to run on a fixed memory hierarchy by specifying the mapping between tasks and levels [Fatahalian et al. 2006]. The work on Legion [Bauer et al. 2012] builds on Sequoia by allowing the programmer to control data sharing and locality using types and by allowing more dynamic hierarchies. One difference between these approaches and our approach is that in our approach, the memory hierarchy mirrors the evolution of the computation automatically, growing and shrinking dynamically as the computation proceeds.

This dynamic and automatic management of hierarchical memory was first proposed in 2015 [Acar et al. 2015a] and realized concretely for functional programs [Raghunathan et al. 2016]. A recent paper [Guatto et al. 2018] extended the technique to support for isolated effects at the sequential portions of the parallel computation. Handling of more general effects remained unknown until the results presented in this paper.

As with many other parallel programming languages, our work assumes a thread scheduler that operates in the run-time to keep the processors busy to the extent possible by migrating threads between processors as needed. Many scheduling algorithms have been designed to improve a variety of metrics, including time [Acar et al. 2019, 2018b, 2013; Arora et al. 2001; Blumofe and Leiserson 1999], responsiveness [Muller and Acar 2016; Muller et al. 2017, 2018], space [Blelloch et al. 1997; Blumofe and Leiserson 1998; Narlikar 1999], and locality [Acar et al. 2002; Blelloch et al. 2008, 2011; Blelloch and Gibbons 2004]. In our implementation, we use a work stealing algorithm based on private deques [Acar et al. 2013].

An important parameter in many parallel codes is the "granularity" or the "grain" at which computations revert from parallel to sequential. In the current state of the art, researchers and practitioners typically control granularity manually by optimizing their codes to switch from parallel to sequential codes at a certain granularity, e.g., small input sizes. This is the technique used in our benchmarks as well as those that we compare against. This manual approach to granularity control has several important drawbacks and there has been recent works that propose solutions that can automate or semi-automate granularity control [Acar et al. 2018a,b, 2016].

8 DISCUSSIONS

We believe that we have merely scratched the surface in understanding the structured nature of effects as characterized by disentanglement in parallel programs and much interesting research remains to be done.

Checking for disentanglement. Because race-freedom implies disentanglement, we can use the many proposed static and dynamic techniques for race-freedom to check for disentanglement. Disentanglement, however, is more general than race-freedom and many programs that use races carefully for improved performance are also disentangled. One direction of future research is therefore to develop techniques that directly check for disentanglement. Because disentanglement requires checking only that objects acquired by memory reads are created by ancestors, it appears easier to check than race-freedom. Static, possibly type-based, techniques would be preferable to dynamic ones. We believe that this is possible by using type systems that can distinguish between concurrent and dependent tasks (e.g., [Balzer et al. 2019]).

In type-safe languages such as ML, it is possible to use disentangled effects purely as an optimization that is hidden from the programmer. In this approach, expert programmers implement efficient libraries using disentangled effects, and provide interfaces to these libraries which are purely functional. Other programmers may then safely use the libraries without reasoning about effects. One research question is to formalize the properties required by such implementations, which are "parallel safe" yet effectful.

Procedural languages. In this paper, we took advantage of disentanglement to provide an efficient parallel memory manager for a functional programming language but we do not assume functional programming. Our techniques therefore could be applicable to procedural languages such as Java and Go.

More general effects. It would be interesting and seems possible to extend disentanglement for more general effects by organizing the memory hierarchy more carefully, possibly by considering certain effects as "synchronizing".

Garbage collection for disentanglement. In this paper, we propose subtree collection as a technique to perform garbage collection efficiently in parallel for disentangled programs. Subtree collection, however, may be insufficient to provably guarantee good time and space bounds, because it is unable to collect internal heaps independently of their children. In particular, with MPL, we have observed that garbage can accumulate at shallow heaps in applications that consist of multiple distinct phases, where each phase discards large quantities of data as soon as the next phase begins. To address this problem, we plan to develop a concurrent collection algorithm that can collect internal heaps of the hierarchy independently of their descendants.

Even though subtree collection is applicable to any subtree, our implementation only supports collections on subtrees that have exactly one active leaf task. An immediately interesting research problem is to implement fully general subtree collection on arbitrary subtrees.

9 CONCLUSION

Careful use of effects is crucial for efficiency in parallel programs but supporting them efficiently in functional languages has been a challenge. In this paper, we establish a disentanglement property of effects in nested-parallel programs and propose memory management techniques that take advantage of this property. We implement a memory manager for the Parallel ML language and show that it performs well. This result takes an important step towards closing the gap between low-level parallel languages such as those based on C, and higher level languages such as parallel functional languages, which offer important correctness benefits that are crucial for parallel programming.

ACKNOWLEDGMENTS

This work was partially supported by the National Science Foundation under grant numbers CCF-1408940 and CCF-1408981.

REFERENCES

[n.d.]. Stanford Large Network Dataset Collection. http://snap.stanford.edu/.

Umut A. Acar, Vitaly Aksenov, Arthur Charguéraud, and Mike Rainey. 2018a. Performance Challenges in Modular Parallel Programs. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '18)*. 381–382.

Umut A. Acar, Vitaly Aksenov, Arthur Charguéraud, and Mike Rainey. 2019. Provably and Practically Efficient Granularity Control. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming (PPoPP '19)*. 214–228.

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

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. 2018b. Heartbeat Scheduling: Provable Efficiency for Nested Parallelism. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. 769–782.

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

Umut A. Acar, Arthur Charguéraud, and Mike Rainey. 2015b. A Work-Efficient Algorithm for Parallel Unordered Depth-First Search. In *ACM/IEEE Conference on High Performance Computing (SC)*. ACM, New York, NY, USA, Article 67, 12 pages.

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

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

T. R. Allen and D. A. Padua. 1987. Debugging Fortran on a Shared Memory Machine. In *Proceedings of the 1987 International Conference on Parallel Processing*. 721–727.

B. Alpern, L. Carter, and E. Feig. 1990. Uniform memory hierarchies. In *Proceedings [1990] 31st Annual Symposium on Foundations of Computer Science*. 600–608 vol.2. https://doi.org/10.1109/FSCS.1990.89581

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

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

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

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

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

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

Stephanie Balzer, Bernardo Toninho, and Frank Pfenning. 2019. Manifest Deadlock-Freedom for Shared Session Types. In *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*. 611–639. https://doi.org/10.1007/978-3-030-17184-1_22

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

Ales Bizjak, Daniel Gratzer, Robbert Krebbers, and Lars Birkedal. 2019. Iron: managing obligations in higher-order concurrent separation logic. *PACMPL* 3, POPL (2019), 65:1–65:30.

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

Guy E. Blelloch, Rezaul A. Chowdhury, Phillip B. Gibbons, Vijaya Ramachandran, Shimin Chen, and Michael Kozuch. 2008. Provably good multicore cache performance for divide-and-conquer algorithms. In *In the Proceedings of the 19th ACM-SIAM Symposium on Discrete Algorithms*. 501–510.

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

Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Harsha Vardhan Simhadri. 2011. Scheduling irregular parallel computations on hierarchical caches. In *Proc. ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 355–366.

Guy E. Blelloch and Phillip B. Gibbons. 2004. Effectively sharing a cache among threads. In *SPAA*.

Guy E. Blelloch, Phillip B. Gibbons, Yossi Matias, and Girija J. Narlikar. 1997. Space-efficient Scheduling of Parallelism with Synchronization Variables. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '97)*. 12–23.

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

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

Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1996. Cilk: An Efficient Multithreaded Runtime System. *J. Parallel and Distrib. Comput.* 37, 1 (1996), 55 – 69.

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

Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling multithreaded computations by work stealing. *J. ACM* 46 (Sept. 1999), 720–748. Issue 5.

Robert L. Bocchino, Stephen Heumann, Nima Honarmand, Sarita V. Adve, Vikram S. Adve, Adam Welc, and Tatiana Shpeisman. 2011. Safe nondeterminism in a deterministic-by-default parallel language. In *ACM POPL*.

Robert L. Bocchino, Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. 2009. A type and effect system for deterministic parallel Java. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications (OOPSLA '09)*. 97–116.

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

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

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

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

C. J. Cheney. 1970. A Non-Recursive List Compacting Algorithm. *Commun. ACM* 13, 11 (Nov. 1970), 677–8.

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

Intel Corp. 2017. Knights landing (KNL): 2nd Generation Intel Xeon Phi processor. In *Intel Xeon Processor E7 v4 Family Specification*. https://ark.intel.com/products/series/93797/Intel-Xeon-Processor-E7-v4-Family.

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

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

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

Martin Elsman. 2001. A Stack Machine for Region Based Programs, See [SPACE 2001]. http://www.diku.dk/topps/space2001/program.html#MartinElsman

Perry A. Emrath, Sanjoy Ghosh, and David A. Padua. 1991. Event Synchronization Analysis for Debugging Parallel Programs. In *Supercomputing '91*. 580–588.

Perry A. Emrath and Davis A. Padua. 1988. Automatic Detection of Nondeterminacy in Parallel Programs. In *Proceedings of the Workshop on Parallel and Distributed Debugging*. 89–99.

Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. 2006. Sequoia: Programming the Memory Hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC '06)*. ACM, New York, NY, USA, Article 83.

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

Mingdong Feng and Charles E. Leiserson. 1999. Efficient Detection of Determinacy Races in Cilk Programs. *Theory of Computing Systems* 32, 3 (1999), 301–326.

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

Cormac Flanagan, Stephen N. Freund, Marina Lifshin, and Shaz Qadeer. 2008. Types for atomicity: Static checking and inference for Java. *ACM Trans. Program. Lang. Syst.* 30, 4 (2008), 20:1–20:53.

Matthew Fluet, Greg Morrisett, and Amal J. Ahmed. 2006. Linear Regions Are All You Need. In *Proceedings of the 15th Annual European Symposium on Programming (ESOP)*.

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.

Matthew Fluet, Mike Rainey, John Reppy, Adam Shaw, and Yingqi Xiao. 2007. Manticore: A Heterogeneous Parallel Language. In *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming (DAMP '07)*. 37–44.

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.

Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The Implementation of the Cilk-5 Multithreaded Language. In *PLDI*. 212–223.

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

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

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

Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. 2002. Region-Based Memory Management in Cyclone. In *Proceedings of SIGPLAN 2002 Conference on Programming Languages Design and Implementation (ACM SIGPLAN Notices)*. ACM Press, Berlin, 282–293.

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 (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.

David R. Hanson. 1990. Fast Allocation and Deallocation of Memory Based on Object Lifetimes. *Software Prac. Experience* 20, 1 (Jan. 1990), 5–12.

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. 2011. Intel Threading Building Blocks. https://www.threadingbuildingblocks.org/.

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

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

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

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018a. RustBelt: securing the foundations of the rust programming language. *PACMPL* 2, POPL (2018), 66:1–66:34. https://doi.org/10.1145/3158154

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018b. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20.

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 (ICFP '10)*. 261–272.

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 (Supercomputing '93)*. ACM,

New York, NY, USA, 262–273. https://doi.org/10.1145/169627.169724

Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. 2007. Optimistic Parallelism Requires Abstractions. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. 211–222.

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

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

Lindsey Kuper, Aaron Turon, Neelakantan R. Krishnaswami, and Ryan R. Newton. 2014b. Freeze After Writing: Quasi-deterministic Parallel Programming with LVars. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. ACM, New York, NY, USA, 257–270.

Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media?. In *WWW '10*. ACM, 591–600.

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

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

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

Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. 2009. The design of a task parallel library. In *Proceedings of the 24th ACM SIGPLAN conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. 227–242.

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.

Henry Lieberman and Carl E. Hewitt. 1981. *A Real-Time Garbage Collector Based on the Lifetimes of Objects*. AI Memo 569a. MIT. ftp://publications.ai.mit.edu/ai-publications/pdf/AIM-569a.pdf

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

Simon Marlow. 2011. Parallel and Concurrent Programming in Haskell. In *Central European Functional Programming School - 4th Summer School, CEFP 2011, Budapest, Hungary, June 14-24, 2011, Revised Selected Papers*. 339–401.

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

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

Stefan 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 (PLDI 2017)*. ACM, New York, NY, USA, 677–692.

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

Girija J. Narlikar. 1999. Scheduling threads for low space requirement and good locality. In *11th Annual ACM Symposium on Parallel Algorithms and Architectures*. 83–95.

Robert H. B. Netzer and Barton P. Miller. 1992. What are Race Conditions? *ACM Letters on Programming Languages and Systems* 1, 1 (March 1992), 74–88.

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.

OpenMP Architecture Review Board. [n.d.]. OpenMP Application Program Interface. http://www.openmp.org/

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

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

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

Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, Muhammad Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. 2011. The tao of

parallelism in algorithms. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*. 12–25.

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 (ICFP 2016)*. ACM, New York, NY, USA, 392–406.

Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin T. Vechev, and Eran Yahav. 2012. Scalable and precise dynamic datarace detection for structured parallelism. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*. 531–542.

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

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*. 55–74.

Dan Robinson. 2017. HPE shows The Machine — with 160TB of shared memory. *Data Center Dynamics* (May 2017).

Douglas T. Ross. 1967. The AED free storage package. *Commun. ACM* 10, 8 (Aug. 1967), 481–492.

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

Jacob T. Schwartz. 1975. Optimization of very high level languages (parts I and II). *Computer Languages* 2–3, 1 (1975), 161–194,197–218.

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 (SPAA '12)*. 68–70.

KC Sivaramakrishnan and Stephen Dolan. 2017. A deep dive into Multicore OCaml garbage collector. http://kcsrk.info/multicore/gc/2017/07/06/multicore-ocaml-gc/ Unpublished manuscript.

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.

A. Sodani. 2015. Knights landing (KNL): 2nd Generation Intel Xeon Phi processor. In *2015 IEEE Hot Chips 27 Symposium (HCS)*. 1–24.

SPACE 2001. *Proceedings of the Second workshop on Semantics, Program Analysis and Computing Environments for Memory Management (SPACE'01)*. London. http://www.diku.dk/topps/space2001/

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

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

Guy L. Steele Jr. 1990. Making Asynchronous Parallelism Safe for the World. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages (POPL)*. ACM Press, 218–231.

Robert Endre Tarjan. 1975. Efficiency of a Good But Not Linear Set Union Algorithm. *J. ACM* 22, 2 (April 1975), 215–225.

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

Mads Tofte and Jean-Pierre Talpin. 1997. Region-Based Memory Management. *Information and Computation* (Feb. 1997). http://www.diku.dk/research-groups/topps/activities/kit2/infocomp97.ps

Aaron Turon, Derek Dreyer, and Lars Birkedal. 2013. Unifying refinement and hoare-style reasoning in a logic for higher-order concurrency. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*. 377–390.

David M. Ungar. 1984. Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm. *ACM SIGPLAN Notices* 19, 5 (April 1984), 157–167. Also published as ACM Software Engineering Notes 9, 3 (May 1984) — Proceedings of the ACM/SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, 157–167, April 1984.

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

Viktor Vafeiadis and Matthew J. Parkinson. 2007. A Marriage of Rely/Guarantee and Separation Logic. In *CONCUR 2007 - Concurrency Theory, 18th International Conference, CONCUR 2007, Lisbon, Portugal, September 3-8, 2007, Proceedings*. 256–271.

David Walker. 2001. On Linear Types and Regions, See [SPACE 2001]. http://www.diku.dk/topps/space2001/program.html#DavidWalker

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, 11Ãć–ÂÃÂŘ13 (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.