Provably Space-Efficient Parallel Functional Programming

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

Because of its many desirable properties, such as its ability to control effects and thus potentially disastrous race conditions, functional programming offers a viable approach to programming modern multicore computers. Over the past decade several parallel functional languages, typically based on dialects of ML and Haskell, have been developed. These languages, however, have traditionally underperformed procedural languages (such as C and Java). The primary reason for this is their hunger for memory, which only grows with parallelism, causing traditional memory management techniques to buckle under increased demand for memory. Recent work opened a new angle of attack on this problem by identifying a memory property of determinacy-race-free parallel programs, called disentanglement, which limits the knowledge of concurrent computations about each other's memory allocations. The work has showed some promise in delivering good time scalability.

In this paper, we present provably space-efficient automatic memory management techniques for determinacy-race-free functional parallel programs, allowing both pure and imperative programs where memory may be destructively updated. We prove that for a program with sequential live memory of R^*, any P-processor garbage-collected parallel run requires at most $O(R^* \cdot P)$ memory. We also prove a work bound of $O(W + R^* \cdot P)$ for P-processor executions, accounting also for the cost of garbage collection. To achieve these results, we integrate thread scheduling with memory management. The idea is to coordinate memory allocation and garbage collection with thread scheduling decisions so that each processor can allocate memory without synchronization and independently collect a portion of memory by consulting a collection policy, which we formulate. The collection policy is fully distributed and does not require communicating with other processors. We show that the approach is practical by implementing it as an extension to the MPL compiler for Parallel ML. Our experimental results confirm our theoretical bounds and show that the techniques perform and scale well.

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

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

ACM Reference Format:
Jatin Arora, Sam Westrick, and Umut A. Acar. 2021. Provably Space-Efficient Parallel Functional Programming. *Proc. ACM Program. Lang.* 5, POPL, Article 18 (January 2021), 33 pages. https://doi.org/10.1145/3434299

1 INTRODUCTION

Nearly every computing device today, ranging from smartphones with 10 cores, and workstations with dozens of cores [Sodani 2015], to servers with hundreds [Corp. 2017], and even thousands of cores [Robinson 2017], is a parallel computer. There has been significant research on developing programming languages for programming such hardware, which has led to development

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

2475-1421/2021/1-ART18
https://doi.org/10.1145/3434299

of *structured* or *nested parallelism*. Nested parallelism relieves the programmer from the burden of managing parallelism manually, allowing them instead to use high-level constructs such as parallel tuples, parallel-for, fork-join, and async-finish, and relies on a thread scheduler to create and schedule parallel tasks automatically and efficiently. Many effective scheduling algorithms have been designed and implemented (e.g., [Acar et al. 2002, 2018; Arora et al. 2001; Blelloch et al. 1997; Blumofe and Leiserson 1999]).

Many procedural parallel programming languages and libraries based on these principles have been devised including Intel Thread Building Blocks (a C++ library) [Intel 2011], Cilk (an extension of C) [Blumofe et al. 1996; Frigo et al. 1998], OpenMP [OpenMP 5.0 2018], Task Parallel Library (a .NET library) [Leijen et al. 2009], Rust [Rust Team 2019], Java Fork/Join Framework [Lea 2000], Habanero Java [Imam and Sarkar 2014], and X10 [Charles et al. 2005]. These languages have the advantage of performance on their side but make writing parallel programs challenging because of their lax control over effects or mutation. With little or no control over effects, it is easy for the programmers to create race conditions that can have disastrous consequences [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 developed parallel functional programming languages that make things much simpler and safer, e.g., multiLisp [Halstead 1984], Id [Arvind et al. 1989], NESL [Blelloch 1996; Blelloch et al. 1994], several forms of parallel Haskell [Hammond 2011; Li et al. 2007; Marlow and Jones 2011; Peyton Jones 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; Westrick et al. 2020; Ziarek et al. 2011]. Some of these languages only support pure or mutation-free functional programs but others such as Parallel ML [Guatto et al. 2018; Westrick et al. 2020] allow using side effects. Because functional languages also support higher order functions (e.g., map, filter, reduce over collections of data), they enable expressing parallel algorithms elegantly and succinctly.

Functional programs, however, fall short when it comes to efficiency and scalability. The primary reason for this is memory: functional languages are memory hungry and 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 and Jones 2011]. This allocation rate increases even more with parallelism, because multiple cores can allocate at the same time. To overcome this fundamental challenge, researchers have proposed assigning each processor its own "processor-local heap" where it can allocate independently without synchronizing with other processors. In nested-parallel programs, this technique can require copying objects, a.k.a., "promotion", from a processor-local heap to the shared heap when the scheduler migrates a thread from one processor to another. For decades, this tug of war between synchronization-free allocation, which is essential for performance of parallel programs, and thread-scheduling, which is essential for scalability seemed unwinnable. Several variants of the processor-local-heap architecture dating back to 1990's [Auhagen et al. 2011; Doligez and Gonthier 1994; Doligez and Leroy 1993; Marlow and Jones 2011; Sivaramakrishnan et al. 2020] have been proposed but none guarantee provable space and work bounds. In contrast, nearly all automatic memory management techniques proposed for the now outdated sequential machines or programming models are provably space and work (time) efficient [Jones et al. 2011].

Recent work on disentanglement has made some progress on this problem. The observation behind disentanglement is that in many parallel programs, a thread does not (need to) know about the allocations of other concurrently executing threads. Disentanglement holds for a fork-join program if it is 1) purely functional [Raghunathan et al. 2016], 2) uses side effects but is determinacy-race-free [Westrick et al. 2020], or 3) uses side effects and has data races but it does not make

allocations of one thread visible to other concurrently executing threads. Using disentanglement, prior work [Raghunathan et al. 2016; Westrick et al. 2020] proposed techniques that allow processors to allocate memory without synchronizing with other processors, and to avoid copying (promoting) data due to thread scheduler actions. Prior work also proposed a memory reclamation technique but (as pointed out by the authors) it is quite conservative and can allow garbage to accumulate.

In this paper, we consider nested-parallel (fork-join) languages and present results for executing them on multiprocessor machines in a provably space efficient manner. **The key idea behind our techniques is to partition memory into a hierarchy of heaps and *schedule heaps* by actively mapping them to processors, much like a thread scheduler that assigns threads (or tasks) to processors.** The heap-scheduler makes assignments by observing thread scheduling decisions, which may migrate threads/tasks between processors. Each processor in turn allocates memory only in the heaps that are assigned to it and is responsible for collecting them. Our techniques apply to all disentangled programs, including purely functional, and imperative programs that use destructive updates. Because disentanglement is currently defined for fork-join programs only, in this paper, we consider the fork-join programming model. Extending our techniques to more general models of parallelism, e.g., futures, requires generalizing the disentanglement theory accordingly (Section 11).

We present a collection policy that determines when a processor can garbage collect to meet desired space and work efficiency bounds. The collection policy is fully distributed: each processor makes its decisions independently of all other processors, without any synchronization. To collect its heaps, a processor can use one or a combination of suitable garbage collection algorithms from the literature [Jones et al. 2011].

We bound the space for P-processor runs in terms of live (reachable) space of sequential runs. One challenge in bounding space of parallel runs is the non-determinism inherent in parallel executions, where parallel computations that are ordered by a sequential run may complete in a different order, resulting in different space complexity. To account for this non-determinism, we show that it suffices to consider a "little bit of" non-determinism by defining a metric, which we call ***unordered reachable space***. This quantity bounds the reachable space over all sequential computations, where the two sides of a parallel pair are executed in either order (i.e., left before right, and right before left).

We describe our techniques by first presenting a cost semantics (Section 3) that constructs a task tree for the computation, and computes the maximum reachable space during the computation. We then present a scheduling algorithm that as it executes tasks, also organizes the memory into a hierarchy of heaps, maps the heaps to the processors, and performs garbage collection. This scheduling algorithm thus extends a traditional thread/task scheduler with the ability to "schedule" memory, in terms of heaps, and garbage collections. For scheduling threads, our scheduling algorithm follows the same techniques as a classic thread scheduling algorithm such as work stealing, and permits flexibility in terms of steal strategies (e.g., randomized, steal-half, round-robin, etc). Our bounds do not depend on specific stealing strategies and apply more generally.

We establish space and work bounds on P-processor computations (Sections 6 and 7). For space, we prove that for a determinacy-race-free nested-parallel program with unordered sequential reachable space of R^*, any P-processor run with our integrated scheduler requires $O(R^* \cdot P)$ space. We also prove that the total work for a P-processor execution is $O(W + R^* \cdot P)$, where W is the work of the computation, i.e., the time for a sequential run. This bound includes the cost for garbage collection. The additive term $R^* \cdot P$ is a consequence of parallel execution, where each processor could in the worst case allocate as much as R^* space and collect it, and is therefore difficult to avoid.

Because our technique involves parallel programs and modern multicore architectures, its implementation requires complex engineering, e.g., due to many low-level concurrency issues

involved, and due to the interaction with the thread scheduler. A legitimate concern is whether the approach can be practical. We extend MPL, a compiler and runtime system for the Parallel ML language, that builds on the industry-strength high-performance compiler [MLton [n.d.]]. We also present a modest empirical study, focusing on the main points of our contributions, including space and time behavior under varying number of processors. We consider a variety of state-of-the-art highly optimized parallel benchmarks that have been developed in the context of procedural parallel programming languages (C/C++ and extensions) and were recently ported to Parallel ML in prior work [Westrick et al. 2020]. The experiments empirically confirm our theoretical results, incurring small overheads compared optimized sequential baselines, scaling well with available cores, and achieving tight space bounds. Notably, for most benchmarks, 70-processor executions consume up to 5-fold space compared to sequential ones, while achieving up to 50 fold speedups.

The contributions of this paper include the following.

- A scheduling algorithm that integrates memory management and thread scheduling.
- Space and work bounds for memory-managed nested parallel programs.
- An implementation that extends the MPL compiler for Parallel ML.
- An empirical evaluation showing evidence that functional languages can compete and even out-compute procedural languages in performance.

Our results give strong evidence for the hypothesis that many safety benefits of functional languages for parallelism may be realized with little or no performance penalty.

2 PRELIMINARIES

2.1 Fork-join

Fork-join is a disciplined synchronization strategy for parallel programs based on ***tasks*** organized into a dynamic ***task tree***, where each task is either ***active***, ***passive***, or ***suspended***. Initially, a fork-join program consists of a single active root task. At any moment, an active task can ***fork*** into two or more child tasks; this is performed by (1) spawning two new tasks for the children, (2) suspending the execution of the task that forked, and then (3) executing the (now active) children in parallel. The parent remains suspended while the children execute. When a task completes, it becomes *passive* and waits for its sibling(s) to complete. As soon as all siblings below a node have completed, they ***join*** with the parent task, which deletes the child tasks and lets the parent resume as an active task.

We say that two tasks are ***concurrent*** when neither task is an ancestor of the other. That is, concurrent tasks could be siblings, cousins, etc. By suspending the execution of the parent at each fork, we guarantee that no task is ever running concurrently with one of its descendants.

2.2 Heap Hierarchy

We give each task a ***heap*** which stores all objects allocated by that task. This essentially assigns "ownership" of each object to the task which performed the allocation. New tasks are initialized with fresh empty heaps, and when a group of siblings join with their parent, we merge their heaps into the parent heap, as illustrated in Figure 1. In this way, all data allocated by a task is returned to its parent upon completion. Note that this is purely a "logical" merge that takes constant time and does not require copying any data (see Section 8 for more details).

The heaps form a dynamic tree which mirrors the task tree. We call this (dynamic) tree the ***heap hierarchy***, and use similar terminology for heaps as for tasks: internal heaps are ***suspended***, and leaf heaps are either ***active*** or ***passive*** (determined by the status of their corresponding tasks).

Every pointer in memory can be classified as either *up*, *down*, *internal*, or *cross*, depending on the relative positions of objects within the heap hierarchy. In particular, consider two objects x and y

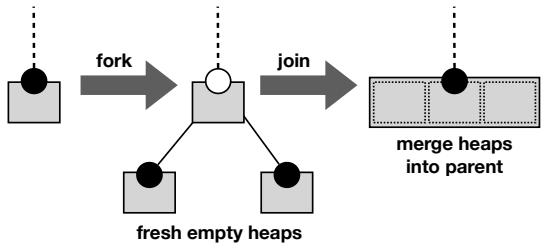

Fig. 1. Forks and joins. Active or passive tasks are black circles, and suspended tasks are white circles. Each task has a heap, drawn as a gray rectangle.

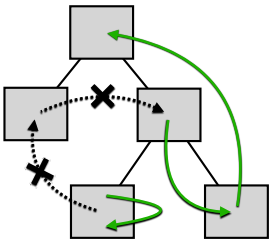

Fig. 2. A disentangled heap hierarchy. Up, down, and internal pointers (solid) are permitted. Cross-pointers (dotted) are disallowed.

and their corresponding heaps $H(x)$ and $H(y)$, and suppose x points to y (i.e. x has a field which is a pointer to y). We classify this pointer as follows:

(1) if $H(x)$ is a descendant of $H(y)$ then the pointer is an ***up-pointer***;
(2) if $H(x)$ is an ancestor of $H(y)$ then it is a ***down-pointer***;
(3) if $H(x) = H(y)$ then it is an ***internal pointer***;
(4) otherwise, it is a ***cross-pointer***.

2.3 Disentanglement

Fork-join programs often exhibit a memory property known as ***disentanglement***, which intuitively is the property that **concurrent tasks remain oblivious to each other's allocations**. Specifically, in a disentangled program, no task will ever obtain a reference to an object allocated by some other concurrent task. This ensures that **the heap hierarchy has no cross-pointers**; that is, disentangled programs only use up-, down-, and internal pointers, as illustrated in Figure 2. Note that down-pointers can only be created via mutation.

We assume throughout this paper that all programs are disentangled, which can be guaranteed in multiple ways. The simplest approach is to disallow mutation entirely: Raghunathan et al. [2016] proved that disentanglement is guaranteed by construction for strict (call-by-value) purely functional languages. More generally, Westrick et al. [2020] proved that all determinacy-race-free programs are disentangled, meaning that we could instead just verify that our programs are race-free (e.g. with an off-the-shelf race-detector). Essentially, they observed that a cross-pointer could only be created by reading a down-pointer into some concurrent heap, which is racy because the read conflicts with the write that created the down-pointer. Note however that **disentanglement permits data races**, and even allows for arbitrary communication between tasks as long as this communication is facilitated by pre-allocated memory (i.e. memory allocated by common ancestors).

2.4 Cost Bounds and Determinism

Taking into account all costs of memory management (including e.g. GC), we are interested in bounding the ***work*** (total number of instructions executed) and ***space*** (maximum memory footprint throughout execution) required for parallel execution. We would ideally like to state these bounds in terms of the work and space required for sequential execution, because this eliminates the need to reason about non-determinism of scheduling. However, if a program itself is non-deterministic, then it is possible for two different parallel executions to have arbitrarily different amounts of work and space (e.g. due to decisions based on the outcome of a race). Therefore, **for cost analysis, we assume programs are deterministic** in the sense of being *determinacy-race-free* (defined

$$
\begin{array}{rrcl}
\textit{Variables} & x, f & & \\
\textit{Numbers} & m & \in & \mathbb{N} \\
\textit{Memory Locations} & \ell & & \\
\textit{Types} & \tau & ::= & \mathtt{nat} \mid \tau \times \tau \mid \tau \rightarrow \tau \mid \tau\ \mathtt{ref} \\
\textit{Storables} & s & ::= & m \mid \mathtt{fun}\ f\ x\ \mathtt{is}\ e \mid \langle \ell, \ell \rangle \mid \mathtt{ref}\ \ell \\
\textit{Expressions} & e & ::= & \ell \mid s \mid x \mid e\ e \mid \langle e, e \rangle \mid \mathtt{fst}\ e \mid \mathtt{snd}\ e \mid \mathtt{ref}\ e \mid\ !\, e \mid e := e \mid \langle e \parallel e \rangle \\
\textit{Memory} & \mu & \in & \textit{Locations} \rightharpoonup \textit{Storables} \\
\textit{Computation Graph} & g & ::= & \bullet \mid n \mid \alpha_{|s|} \mid g \oplus g \mid g \otimes g \\
\textit{Task Tree} & \mathcal{T} & ::= & [g] \mid g \oplus (\mathcal{T} \otimes \mathcal{T})
\end{array}
$$

Fig. 3. Syntax

below), which guarantees that parallel and sequential executions are similar enough in order to prove our bounds. Under this assumption, disentanglement is guaranteed [Westrick et al. 2020], so our techniques are applicable. Note that even though we only provide performance bounds for race-free programs, **our memory management techniques are nevertheless correct for all disentangled programs, including those that are non-deterministic** (e.g. due to a race).

A ***determinacy race*** occurs when two concurrent tasks access the same memory location, and at least one of these accesses modifies the location [Feng and Leiserson 1997]. This is essentially the same notion as a "data race"; however there is a subtlety which leads us to prefer the term "determinacy race". Programs which are determinacy-race-free are **deterministic in a strong sense**: not only is the final result the same in every possible execution, but also the specifics of how that result is computed are precisely the same every time.[1] That is, each determinacy-race-free program has a unique *computation graph*: a directed, acyclic graph where vertices are executed instructions and edges are sequential dependencies.

3 LANGUAGE

For a formal analysis of our techniques, we consider a simple call-by-value functional language extended with fork-join (nested) parallelism. Richer constructs like arithmetic operators and arrays could be added, but we omit them for brevity. The language allows for unrestricted side-effects and does not statically enforce either disentanglement (required for our memory management techniques) or race-freedom (required for our cost bounds). The choice of a functional language here is not essential, and our analysis can be extended to an imperative setting.

In addition to defining how an expression evaluates to a value, our operational semantics defines some intensional aspects: the task tree and the space usage. Task trees are suitable for defining both: the garbage collection strategy and the scheduling algorithm The space usage gives the maximum memory footprint of a program's parallel run. Together with the task trees, the space usage serves as a basis for the asymptotic analysis of work and space of garbage-collected evaluation.

3.1 Syntax

Figure 3 gives the syntax for the language studied in this work.

Types. The types include a base type of natural numbers, function types and product types for expressing parallel pairs. The type system also supports mutable references.

Memory Locations and Storables. We distinguish between storables s, which are always allocated in the heap, and memory locations ℓ. Storables include natural numbers, named recursive functions,

[1] This is also known as *internal determinism* [Blelloch et al. 2012] at the level of individual memory reads and writes.

pairs of memory locations, and mutable references to other memory locations. Storables are not irreducible but only step to locations. Locations are the only irreducible form of the language.

Expressions. Expressions in our language include variables, locations, storables, and introduction and elimination forms for the standard types. Parallelism is expressed using parallel pairs ($\langle e \parallel e \rangle$). For an expression e, we use $\mathrm{locs}(e)$ to denote the set of locations referenced by it.

Memory. In order to give an operational semantics for memory effects, we include a map μ from locations to storables. We refer to μ as the memory, $\mathrm{dom}(\mu)$ for the set of locations mapped by μ, $\mu(\ell)$ to look up the storable mapped to ℓ and $\mu[\ell \hookrightarrow s]$ to extend μ with a new mapping.

3.2 Cost Semantics

The cost semantics of the language is based on a standard transition (small-step) semantics of a functional language with mutation. The semantics relation is written as:

$$\rho \vdash \mu\,;\mathcal{T}\,;R\,;e \rightarrow \mu'\,;\mathcal{T}'\,;R'\,;e'.$$

The semantics records the work done during the evaluation in a task tree $\mathcal{T}$. The context in the relation, ρ, is the set of locations referred by the evaluation context of e. It is used to compute the memory footprint of the evaluation. The maximum memory footprint (space usage) so far is stored in R. The inference rules for the semantics are given in Figure 4.

Task Trees. At any step, the program's evaluation can be organized into a tree whose vertices represent *tasks*, and edges represent the control dependencies between them. Each task of the tree is represented as a ***computation graph*** (g) consisting of computation nodes. These nodes record all the steps taken by that task.

At the start of evaluation, the tree only has the ***root task***. Since no work has been done yet, the root task is represented as the empty graph $[\bullet]$. The square brackets indicate that the task is a leaf. The rule Fork precedes the evaluation of parallel pairs and adds two new leaves ($[\bullet], [\bullet]$) using the parallel composition ($\otimes$). The rule also removes the square brackets from the parent task to mark that it is suspended. The execution of these subtasks can be interleaved non-deterministically by rules ParL and ParR. Once both the subtasks finish, the rule Join executes. This rule deletes the vertices corresponding to the subtasks but adds their computation graphs to the parent using the sequential composition ($\oplus$). The parent task becomes a leaf again and resumes execution.

Other aspects of the language (sequential pairs, functions, and mutation) are sequential and do not alter the structure of the tree. Instead, their evaluation extends the leaf tasks with computation nodes. For simplicity, we assume that each step of their evaluation (with rules App, Bang, Upd) requires a unit of computation and their rules add a computation node (n) to the graph. The computation node added by rule Alloc is represented by $\alpha_{|s|}$, where $|s|$ is the amount of memory allocated. We assume that an allocation of $|s|$ units of memory requires $|s|$ units of computation.

Space Usage. Space usage is the maximum memory footprint of an evaluation. The locations in the memory store, μ, can be viewed as nodes of a directed graph, in which edges are the pointers between locations. For two locations ℓ, ℓ' we say that ℓ *points to* ℓ' or $\ell \rightarrow_\mu \ell'$ if the storable at ℓ has a pointer to the storable at ℓ'. Locations that are explicitly referenced by the program expression are called 'roots'. The context ρ tracks the roots for the evaluation context of the sub-expression that steps, i.e., for any step with context ρ and sub-expression e, $(\rho \cup \mathrm{locs}(e))$ contains all the roots of the program. To achieve this, the rules extend the context appropriately in the premise before evaluating a sub-expression. For example, in rule ASL (application step left) the roots of e_2 are added to ρ before evaluating e_1. The locations reachable from the roots are potentially being used in the evaluation and count towards its memory footprint. At step t, we denote the set of roots (***root***

$$\frac{R' = \max(\,|E^+_\mu(\rho \cup \mathrm{locs}(s))| + |s|, R\,) \qquad \ell \notin \mathrm{dom}(\mu) \qquad \mu' = \mu[\ell \hookrightarrow s]}{\rho \vdash \mu\,;[g]\,;R\,;s \to \mu'\,;[g \oplus \alpha_{|s|}]\,;R'\,;\ell}\ \textsc{Alloc}$$

$$\frac{\rho \cup \{\mathrm{locs}(e_2)\} \vdash \mu\,;\mathcal{T}\,;R\,;e_1 \to \mu'\,;\mathcal{T}'\,;R'\,;e_1'}{\rho \vdash \mu\,;\mathcal{T}\,;R\,;(e_1\ e_2) \to \mu'\,;\mathcal{T}'\,;R'\,;(e_1'\ e_2)}\ \textsc{ASL} \qquad \frac{\rho \cup \{\ell_1\} \vdash \mu\,;\mathcal{T}\,;R\,;e_2 \to \mu'\,;\mathcal{T}'\,;R'\,;e_2'}{\rho \vdash \mu\,;\mathcal{T}\,;R\,;(\ell_1\ e_2) \to \mu'\,;\mathcal{T}'\,;R'\,;(\ell_1\ e_2')}\ \textsc{ASR}$$

$$\frac{\mu(\ell_1) = \mathtt{fun}\ f\ x\ \mathtt{is}\ e_b}{\rho \vdash \mu\,;[g]\,;R\,;(\ell_1\ \ell_2) \to \mu\,;[g \oplus n]\,;R\,;[\ell_1, \ell_2\ /\ f, x]e_b}\ \textsc{App}$$

$$\frac{\rho \vdash \mu\,;\mathcal{T}\,;R\,;e \to \mu'\,;\mathcal{T}'\,;R'\,;e'}{\rho \vdash \mu\,;\mathcal{T}\,;R\,;(\mathtt{fst}\ e) \to \mu'\,;\mathcal{T}'\,;R'\,;(\mathtt{fst}\ e')}\ \textsc{FstS} \qquad \frac{\mu(\ell) = \langle \ell_1, \ell_2 \rangle}{\rho \vdash \mu\,;[g]\,;R\,;(\mathtt{fst}\ \ell) \to \mu\,;[g \oplus n]\,;R\,;\ell_1}\ \textsc{Fst}$$

$$\frac{\rho \vdash \mu\,;\mathcal{T}\,;R\,;e \to \mu'\,;\mathcal{T}'\,;R'\,;e'}{\rho \vdash \mu\,;\mathcal{T}\,;R\,;(\mathtt{ref}\ e) \to \mu'\,;\mathcal{T}'\,;R'\,;(\mathtt{ref}\ e')}\ \textsc{RefS}$$

$$\frac{\rho \vdash \mu\,;\mathcal{T}\,;R\,;e \to \mu'\,;\mathcal{T}'\,;R'\,;e'}{\rho \vdash \mu\,;\mathcal{T}\,;R\,;(!\ e) \to \mu'\,;\mathcal{T}'\,;R'\,;(!\ e')}\ \textsc{BangS} \qquad \frac{\mu(\ell_1) = \mathtt{ref}\ \ell_2}{\rho \vdash \mu\,;[g]\,;R\,;(!\ \ell_1) \to \mu\,;[g \oplus n]\,;R\,;\ell_2}\ \textsc{Bang}$$

$$\frac{\rho \cup \{\mathrm{locs}(e_2)\} \vdash \mu\,;\mathcal{T}\,;R\,;e_1 \to \mu'\,;\mathcal{T}'\,;R'\,;e_1'}{\rho \vdash \mu\,;\mathcal{T}\,;R\,;(e_1 := e_2) \to \mu'\,;\mathcal{T}'\,;R'\,;(e_1' := e_2)}\ \textsc{USL} \qquad \frac{\rho \cup \{\ell_1\} \vdash \mu\,;\mathcal{T}\,;R\,;e_2 \to \mu'\,;\mathcal{T}'\,;R'\,;e_2'}{\rho \vdash \mu\,;\mathcal{T}\,;R\,;(\ell_1 := e_2) \to \mu'\,;\mathcal{T}'\,;R'\,;(\ell_1 := e_2')}\ \textsc{USR}$$

$$\frac{}{\rho \vdash \mu_0[\ell_1 \hookrightarrow s]\,;[g]\,;R\,;(\ell_1 := \ell_2) \to \mu_0[\ell_1 \hookrightarrow \mathtt{ref}\ \ell_2]\,;[g \oplus n]\,;R\,;\ell_2}\ \textsc{Upd}$$

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

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

$$\frac{\rho \cup \{\mathrm{locs}(e_2)\} \vdash \mu\,;\mathcal{T}_1\,;R\,;e_1 \to \mu'\,;\mathcal{T}_1'\,;R'\,;e_1'}{\rho \vdash \mu\,;g \oplus (\mathcal{T}_1 \otimes \mathcal{T}_2)\,;R\,;\langle e_1 \parallel e_2 \rangle \to \mu'\,;g \oplus (\mathcal{T}_1' \otimes \mathcal{T}_2)\,;R'\,;\langle e_1' \parallel e_2 \rangle}\ \textsc{ParL}$$

$$\frac{\rho \cup \{\mathrm{locs}(e_1)\} \vdash \mu\,;\mathcal{T}_2\,;R\,;e_2 \to \mu'\,;\mathcal{T}_2'\,;R'\,;e_2'}{\rho \vdash \mu\,;g \oplus (\mathcal{T}_1 \otimes \mathcal{T}_2)\,;R\,;\langle e_1 \parallel e_2 \rangle \to \mu'\,;g \oplus (\mathcal{T}_1 \otimes \mathcal{T}_2')\,;R'\,;\langle e_1 \parallel e_2' \rangle}\ \textsc{ParR}$$

Fig. 4. Language Dynamics.

set) as ρ^t. The set of reachable locations is denoted as $E^+_\mu(\rho^t)$ and the cumulative size of storables mapped by this set is represented as $|E^+_\mu(\rho^t)|$. Space usage is formally defined to be $\max_t |E^+_\mu(\rho^t)|$. The semantics maintains the space usage of the evaluation in R. Since the reachable memory can increase only after an allocation, R is updated only after the program allocates (rule Alloc). This rule computes the reachable memory from the root set ($\rho \cup \mathrm{locs}(s)$), adds the size of the newly allocated storable and updates R if the space usage has increased.

The other rules are standard and we skip their description for sake of brevity. The evaluation of a program e starts with the state $(\emptyset\,;[\bullet]\,;0\,;e)$. Suppose it terminates with the state $(\mu\,;[g]\,;R\,;\ell)$. Then the total work done for evaluating e is the number of nodes in the graph g, where each allocation node $(\alpha_{|s|})$ is counted $|s|$ times. Additionally, the max space usage during the evaluation is R, which is the minimum amount of memory required to execute the program.

4 INTEGRATING THREAD SCHEDULING AND GARBAGE COLLECTION

We present a technique that integrates thread scheduling and garbage collection to achieve provably space and work efficient garbage collection.

We consider executing a nested-parallel (fork-join) program on P workers, with identities $0 \leq p < P$. Each worker executes tasks and may perform garbage collection. As is typical with nested-parallel programs, a scheduling algorithm assigns tasks to workers dynamically in an online fashion; each worker then executes the task that they are assigned.

The crux of our approach lies in the interaction between memory and the thread scheduler. Our approach specifically relies on a ***heap scheduler*** that

(1) partitions memory into heaps each of which corresponds in a one-to-one fashion with tasks, and organizes the heaps into a heap hierarchy (tree) that mirrors the task tree, and
(2) dynamically assigns each heap to one and only one worker at any time, partitioning thus the memory between the workers.

Each worker in turn only allocates memory in its heaps and is responsible for collecting the unreachable objects in its heaps by using a garbage collection algorithm. The times for garbage collection, are decided by a fully-distributed ***collection policy***, in which each worker can decide to garbage-collect on its own.

Heap Scheduler. Much like a thread scheduler that distributes threads between workers, we present a heap scheduler that assigns heaps to workers. More specifically, the heap scheduler assigns a ***heap set*** M_p to each worker p such that

- each and every heap is assigned to a worker,
- for different workers p and q, $M_p \cap M_q = \emptyset$.

Thus, the heap scheduler partitions all heaps between the workers. Because heaps of different workers are disjoint, each worker can collect its heaps independently from others.

Our heap scheduler assigns every active heap to the worker that is executing the corresponding task. The most important difference between our heap scheduler and standard thread schedulers is that our scheduler must also assign suspended and passive heaps, i.e., heaps whose tasks are not active. Such heaps take up space, and we must carefully assign them to workers to ensure that they are subject to garbage collection. Our scheduler guarantees the following invariants:

(1) if a worker p is executing a task, then the heap of the task is assigned to p
(2) if a suspended (internal) heap is in M_p, then at least one of its children is also in M_p, and
(3) every passive heap belongs to the same worker as its sibling.

Roughly speaking, these invariants enforce that each heap set is a path-like structure, consisting of the path from an ancestor heap to an active leaf and some passive leaves.

Example. Consider the two heap trees in Figure 5. Since there is a one to one correspondence between tasks and heaps, we will interpret the boxes in figure as either tasks or heaps. The gray boxes represent active or suspended tasks and the black box represents a task that is passive. The active tasks are being executed by four workers indexed 0, 1, 2, and 3. The gray ellipses represent the sets of heaps assigned to the workers inside them. The figure shows example partitions of the

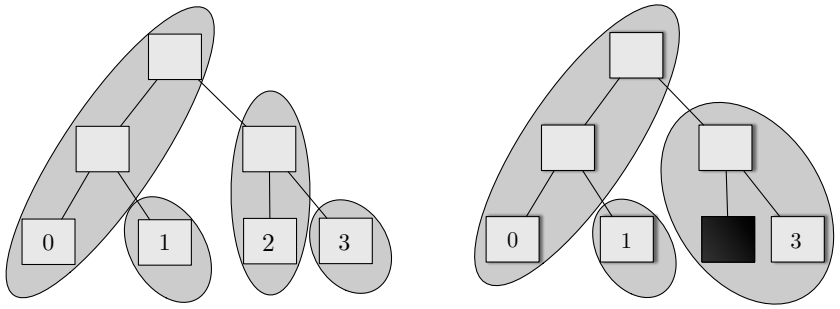

Fig. 5. Example heap sets of the workers before and after 2 finishes its task

heap trees that satisfy the invariants described above. The tree on the left corresponds to a step at which four leaf heaps are active.

Suppose the leaf task at worker 2 terminates. The leaf becomes passive because its sibling is not ready to join. In order to maintain the third invariant, the heap scheduler 'migrates' worker 2's heaps to the worker that has the sibling, i.e., worker 3. The tree on the right shows the heap sets after the migration. Intuitively, the memory state at worker 3 is as if the passive leaf had executed on it (and not on worker 2). This allows us to compare the space usage in this execution to space usage in an execution where both siblings execute sequentially on worker 3. This comparison is crucial for the proof of space bounds in Section 6, as it relates the space of each heap set to the sequential space.

Collection Policy. Each worker manages the heaps that are assigned to it and determines when to perform garbage collection on its own. To this end, a worker p maintains a counter λ_p, that tracks the amount of memory that survived the last collection performed by p. This counter is a rough estimate of the maximum space usage within M_p and guides the worker on when it should collect. Specifically, the worker p ensures that $|M_p| < \kappa \cdot \lambda_p$, where $|M_p|$ is the heap set's size and $\kappa > 1$ is an adjustable constant in the collection policy. That is, the worker makes sure that the size of the memory assigned to it remains within a constant factor of this counter and does not grow arbitrarily. When the worker observes that $|M_p| \geq \kappa \cdot \lambda_p$, it executes a collection algorithm on its heaps. The collection algorithm determines reachability in M_p and reclaims the unreachable locations. After the collection is completed, the worker re-initializes the counter λ_p to the new size of its heap set and resumes executing its assigned task.

In order for garbage collection to be correct, we must ensure that a reachable object is never reclaimed. Determining which objects are reachable is tricky, because the set of reachable objects within a heap may depend on pointers from other workers (for example, an object may be only reachable from a root that lives in a heap assigned to some other worker). To handle this, we equip each heap set with a ***remembered set*** that keeps track of "additional roots" for garbage collection. Essentially, each worker assumes that all objects in the remembered set are reachable and keeps all objects reachable from these "additional roots" alive. To be correct, we must ensure that the remembered set is ***conservative***: the set of objects reachable from the remembered set must include all live objects within the heap set, but it may also include some garbage. Our proofs of performance bounds (Section 6) account for this extra garbage.

To maintain remembered sets efficiently, we take advantage of disentanglement, which ensures that the heap hierarchy is **free of cross-pointers**. Because of this invariant, we only need to reason about up-pointers and down-pointers from heap sets of other workers. The down-pointers are

conceptually simple to handle: we can think of the remembered set as a concurrent data structure that is updated whenever a new down-pointer into the heap set is created. To handle up-pointers, we use a ***snapshotting*** strategy that conservatively estimates reachability within suspended heaps by adding a small number of additional objects to remembered sets. Snapshotting is implemented via two mechanisms: (1) in the work-stealing scheduler (Section 5), whenever a task is stolen, the thief also inserts that task's roots into the appropriate remembered set, and (2) a write barrier ensures that the conservative reachability estimate is not affected by mutating internal pointers.

5 THREAD AND HEAP SCHEDULING WITH WORK STEALING

To make the high-level description of the thread scheduler, heap scheduler, and the collection algorithm more precise, we first present an abstract scheduling algorithm that the workers implement and describe how the policy partitions the heaps of the heap tree. We then develop a collection algorithm that can be used to collect these partitions.

5.1 Thread and Heap Scheduling

Scheduling algorithms like work-stealing assign each worker a double ended queue or ***deque***, of tasks that the worker may execute. The execution starts with the root task placed into the deque of the worker 0. The other deques are empty since there are no other tasks available at the start. All the heap sets are empty, and the counter of all the workers is zero. The execution by workers proceeds in steps. For simplicity, we assume that executing a task requires the worker to execute several instructions, each of which needs one step to complete.

The pseudo-code of the scheduling algorithm is in Figure 6. For presentability, the pseudo-code does not deal with concurrency issues like a real implementation would. We defer those details to Section 8. We also assume an implementation of some helper modules: (1) Module `Deque` provides the type `Deque.deque` for deques and functions like `Deque.empty`, `Deque.popBottom`, `Deque.pushBottom` that are used to modify and query the deque, (2) Module `Task` provides functions like `parent`, `sibling` and `Instructions` that implement the task tree abstraction, and (3) Module `Heap` provides similar functions for heap trees, gives a function `Merge` for merging leaf heaps with their parent, and a function `NewHeap` for creating new heaps.

The code also assumes the abstract function `stealWork`, which is used by the workers to populate their deque. This function ***steals*** tasks by popping from the top of other deques. Various stealing strategies can be used to implement this function. The `collect` function executes a collection algorithm that reclaims the unreachable locations in the input heap set. The collection algorithm is described later in the section.

The worker p first checks its deque ($\mathcal{R}(p)$) for an available task. If the deque is empty, p executes the `stealWork` function. The function call at line 5 returns after it is successful in the steal attempt. The worker then pops off a new task from the bottom of its deque, creates a new heap for the task and adds it to the heap set. We use the variable T_p to refer to the task that p is executing. Then it proceeds as follows: at the start of each step, the worker checks if it needs to collect M_p. If the size of M_p is more than κ times the counter, the worker executes the `collect` function and updates its local counter.

Otherwise, $|M_p|$ is within limits, and the worker p executes an instruction of the task that it is working on. When the task forks, the worker p executes the `fork` case. The `fork` case enqueues the subtasks of T_p to the bottom of the deque and then ***breaks*** from the loop. As a result, the current task is suspended and the worker returns to line 6. The worker then pops off one of the subtasks from the bottom of the deque, creates a new heap for it and starts executing it.

When T_p is ready to join, the worker executes the `join` case. Note that for the join to execute, the sibling task of T_p should have terminated. If it has not, then T_p becomes passive and its heap is

```
1  λ_p : int                 // Size of live set
2  R(p): Deque.deque         // Work deque
3  repeat
4      if Deque.empty(R(p)) then
5          R(p) ← stealWork()
6      T_p ← Deque.popBottom(R(p))
7      H_p ← NewHeap(T_p)
8      M_p ← M_p ∪ H_p
9      for each I in Instructions(T_p) do
10         if ( |M_p| ≥ κ · λ_p ) then
11             collect(M_p)
12             λ_p ← |M_p|
13         case I of
14             fork(T_1, T_2) →
15                 Deque.pushBottom(T_1)
16                 Deque.pushBottom(T_2)
17                 break
18             join →
19                 if sibling in R(p) then
20                     break
21                 else if sibling has not terminated then
22                     // surrender to worker q where sibling(H_p) ∈ M_q
23                     M_q ← M_q ∪ M_p
24                     M_p ← ∅
25                     break
26                 else // sibling has terminated:
27                     Merge(H_p, sibling(H_p), parent(H_p))
28                     T_p ← parent(T_p)
29                     H_p ← Heap(T_p)
30             otherwise →
31                 execute I
```

Fig. 6. Scheduling algorithm. The break is a control flow construct that exits the for loop.

reassigned to the worker that has the sibling heap. Let the sibling task of T_p be T'_p. When p executes the `join` case, the following three cases arise:

case (i): Sibling task, T'_p, is in $\mathcal{R}(p)$. In this case, the worker breaks from the loop. It pops off the sibling from the bottom of the deque and starts executing it.
case (ii): T'_p is not in $\mathcal{R}(p)$ and is not ready to join. In this case, p finds the worker q such that $H'_p \in M_q$. The worker then ***surrenders*** all its heaps to q. Due to this transfer, the sibling heaps are now assigned to the same worker, and when T'_p terminates, the heaps can be joined with their parent. This transfer does not involve copying of heaps (see Section 8 for more details). In the next step, p will try to find another task to work on.
Note that such a worker q is guaranteed to exist because the sibling task T'_p is neither in the deque nor is it ready to join. This means that it is either suspended or active. Therefore, the heap corresponding to T'_p is in the heap tree and has to be in some worker's heap set (the heap sets partition the heaps of the heap tree).

case (iii): T'_p is ready to join. In this case, the worker p can resume the execution of the parent task since both of the subtasks have completed. The heaps of the subtasks are merged with the parent's heap. The `Merge` function does not copy the contents of the heaps and instead does a 'logical' merge in constant time (see Section 8).

Instructions other than `fork` and `join` match with the `otherwise` case of the pseudo code. These are simply executed without any updates to the state of the worker. Thus, if the worker p does not change the task it is working on, no changes are made to M_p except if some other worker surrenders and synchronizes with it.

5.2 Local Collection

Our collection algorithm is a tracing algorithm that first computes reachability by following pointers from a root set and then collects the unreachable locations. It is *local* in the sense that it only traces the pointers within the worker's heap set. The set of reachable locations in a heap set, however, also depends on pointers from heaps of other workers. We maintain remembered sets to ensure that our algorithm does not have to scan for such pointers.

Remembered Set. A remembered set consists of locations of a heap set that may be referenced by other heap sets. Locations of remembered sets are treated as roots and are assumed to be reachable during the collection. Recall that the root set ρ^t is the set of locations referred by the program at step t. The remembered set for worker p is such that reachability from it and the set ρ^t while only following pointers in M_p is *conservative*. That is the reachability thus computed may consider an unreachable location in M_p to be reachable but not vice-versa. Formally, we denote $E^+_p(\rho)$ as the set of reachable locations from ρ while following pointers within M_p and use $E^+_\mu(\rho)$ to denote reachability while following every pointer in the memory store μ. The remembered set σ^t_p is such that

$$E^+_p(\sigma^t_p \cup \rho^t) \supseteq E^+_\mu(\rho^t) \cap \mathrm{locs}(M_p).$$

Recall that disentanglement restricts the pointer dependence among heaps to up-pointers and down-pointers. Thus, in the remembered set we only need to consider locations reachable from these pointers. Let A_p be the set of ancestor heaps of M_p that are not at worker p. Similarly, let S_p be the set of successor heaps that were stolen from worker p. Lastly, let $D_p \supseteq S_p$ be the set of all the heaps in S_p and their descendants. We use the term *foreign* to refer to tasks/heaps that correspond to the descendant heaps in D_p. By disentanglement, the heaps in sets A_p and D_p are the only ones with pointers into the heap set M_p. Since the set A_p corresponds to ancestor heaps, their tasks are suspended. Thus, only foreign tasks can change reachability in M_p. The remembered set σ_p is updated based on actions of foreign tasks as follows:

(1) *Steal.* The program expression of a stolen task may have pointers into the heaps of M_p. Thus, when a task is stolen from p all the locations referenced by it are added to σ_p.
(2) *Pointer Deletion.* If a foreign task deletes a pointer from ℓ in $(A_p \cup M_p)$ to ℓ' in M_p, then we add ℓ' to σ_p.
(3) *Down Pointer Update.* Suppose some task creates a pointer from location ℓ in A_p to location ℓ' in M_p. Since this is a new down pointer to a heap of M_p, the location ℓ' is added to σ_p.

The first two update rules ensure two properties: (i) All locations in the heap set M_p reachable from a foreign heap are reachable from the remembered set and, (ii) No action of a foreign task changes reachability from the remembered set. A formal proof of these properties can be done by induction. Property (i) holds at the time of steal when a new (foreign) heap is created because we add all the up pointers from the new heap to the remembered set. At the inductive step, if a foreign task creates a new pointer to some location in M_p, then the location is reachable from its

corresponding heap. By property (i) this location is also reachable from the remembered set. Thus this new pointer does not change reachability in any way. Otherwise, if a foreign task deletes a pointer to location ℓ then ℓ is reachable from its heap. By property (i) the location was reachable from σ_p, before the deletion. After the deletion, we add ℓ to σ_p and ensure that it remains reachable.

The down pointer updates ensure that every location reachable from a down pointer is reachable from the remembered set. Thus, the collection algorithm does not need to scan the ancestor heaps A_p to compute reachability in heap set M_p. We make a simplifying assumption: if a location is the target of a down pointer then it stays reachable until its heap merges with the heap of the source (of the down pointer). This assumption does not affect the correctness of our algorithm but it simplifies its analysis. With this assumption, the remembered set is not just conservative for locations pointed by down pointers, it is also exact, i.e, if a location pointed by a down pointer is in the remembered set then it is reachable.

The remembered set is also updated when some worker surrenders its heaps to worker p. Suppose tasks T_p and T_q are sibling tasks on workers p and q respectively and worker q surrenders to p after task T_q terminates. Before the surrender, the parent heap of siblings H_p and H_q is either on worker p or on worker q. Consider the case in which the parent heap is on worker p. By definition, heap H_q is a foreign heap. Since H_q is being surrendered to M_p, the pointers from it no longer need to be remembered. Thus, all locations that correspond to '*Steal*' or '*Pointer Deletion*' action by T_q are removed from the remembered set σ_p. Moreover, all the locations in the remembered set of worker q (σ_q) that correspond to down pointers from any heap of M_p are also deleted. The remaining locations of σ_q are added to σ_p. The other case where the parent heap is on worker q before the surrender is similar.

Cost Specification. Given a conservative remembered set for a heap set M and the root set ρ^t, the collection algorithm computes all the reachable locations in the heap set. Since the remembered set is conservative, the algorithm only traces pointers within the heap set. The number of locations and pointers in the heap set M are bounded by $|M|$. Thus, marking every reachable location takes $O(|M|)$ work and space using a procedure like depth first search. Moreover, a procedure that goes through every location in M and reclaims the unreachable ones takes $O(|M|)$ work. Thus, if W_c is the work done by the collection algorithm and M_c is its space complexity then,

(1) $W_c < c_1 \cdot |M|$
(2) $M_c < c_2 \cdot |M|$

The abstract description here skips some interesting practical details which we cover in the implementation section (Section 8). However, it shows that this specification is realistic.

The above constraints do not include the overhead of maintaining the remembered set σ_p. The up pointers added when a task is stolen are stored in closure representation and we only need P of them at any step. This overhead is bounded by $\kappa' \cdot P$, where κ' is the size of the largest closure. The number of locations added because of deleted pointers and new down pointers are loosely upper bounded by the size of the heap set. Both of these can be accounted for by adjusting the constant c_2 above. We briefly mention how we implement them in Section 8.

6 SPACE BOUND

We give a space bound by comparing the memory used in a parallel execution (with garbage collection) with the unordered reachable space in a sequential execution. To that end, we formalize a sequential execution and its space usage.

Sequential Cost Semantics. To define a sequential execution, we replace the rules that evaluate a parallel pair (ParL and ParR) by the inference rules in Figure 7. The rules ExL and ExR introduce

$$\frac{\rho \cup \{\mathsf{locs}(e_2)\} \vdash \mu\,;\mathcal{T}_1\,;R\,;e_1 \to^* \mu'\,;\mathcal{T}_1'\,;R'\,;\ell_1 \qquad \rho \cup \{\ell_1\} \vdash \mu'\,;\mathcal{T}_2\,;R'\,;e_2 \to^* \mu''\,;\mathcal{T}_2'\,;R''\,;\ell_2}{\rho \vdash \mu\,;g \oplus (\mathcal{T}_1 \otimes \mathcal{T}_2)\,;R\,;\langle e_1 \parallel e_2\rangle \to_L \mu''\,;g \oplus (\mathcal{T}_1' \otimes \mathcal{T}_2')\,;R''\,;\langle \ell_1 \parallel \ell_2\rangle}\ \textbf{ExL}$$

$$\frac{\rho \cup \{\mathsf{locs}(e_1)\} \vdash \mu\,;\mathcal{T}_2\,;R\,;e_2 \to^* \mu'\,;\mathcal{T}_2'\,;R'\,;\ell_2 \qquad \rho \cup \{\ell_2\} \vdash \mu'\,;\mathcal{T}_1\,;R'\,;e_1 \to^* \mu''\,;\mathcal{T}_1'\,;R''\,;\ell_1}{\rho \vdash \mu\,;g \oplus (\mathcal{T}_1 \otimes \mathcal{T}_2)\,;R\,;\langle e_1 \parallel e_2\rangle \to_R \mu''\,;g \oplus (\mathcal{T}_1' \otimes \mathcal{T}_2')\,;R''\,;\langle \ell_1 \parallel \ell_2\rangle}\ \textbf{ExR}$$

$$\frac{\begin{array}{c}\rho \vdash \mu\,;g \oplus (\mathcal{T}_1 \otimes \mathcal{T}_2)\,;R\,;\langle e_1 \parallel e_2\rangle \to_L \mu'\,;g \oplus (\mathcal{T}_1' \otimes \mathcal{T}_2')\,;R'\,;\langle \ell_1' \parallel \ell_2'\rangle \\ \rho \vdash \mu\,;g \oplus (\mathcal{T}_1 \otimes \mathcal{T}_2)\,;R\,;\langle e_1 \parallel e_2\rangle \to_R \mu''\,;g \oplus (\mathcal{T}_1'' \otimes \mathcal{T}_2'')\,;R''\,;\langle \ell_1'' \parallel \ell_2''\rangle \qquad R' \geq R''\end{array}}{\rho \vdash \mu\,;g \oplus (\mathcal{T}_1' \otimes \mathcal{T}_2')\,;R\,;\langle e_1 \parallel e_2\rangle \to \mu'\,;g \oplus (\mathcal{T}_1' \otimes \mathcal{T}_2')\,;R'\,;\langle \ell_1' \parallel \ell_2'\rangle}\ \textsc{PickL}$$

$$\frac{\begin{array}{c}\rho \vdash \mu\,;g \oplus (\mathcal{T}_1 \otimes \mathcal{T}_2)\,;R\,;\langle e_1 \parallel e_2\rangle \to_L \mu'\,;g \oplus (\mathcal{T}_1' \otimes \mathcal{T}_2')\,;R'\,;\langle \ell_1' \parallel \ell_2'\rangle \\ \rho \vdash \mu\,;g \oplus (\mathcal{T}_1 \otimes \mathcal{T}_2)\,;R\,;\langle e_1 \parallel e_2\rangle \to_R \mu''\,;g \oplus (\mathcal{T}_1'' \otimes \mathcal{T}_2'')\,;R''\,;\langle \ell_1'' \parallel \ell_2''\rangle \qquad R'' > R'\end{array}}{\rho \vdash \mu\,;g \oplus (\mathcal{T}_1' \otimes \mathcal{T}_2')\,;R\,;\langle e_1 \parallel e_2\rangle \to \mu''\,;g \oplus (\mathcal{T}_1'' \otimes \mathcal{T}_2'')\,;R''\,;\langle \ell_1'' \parallel \ell_2''\rangle}\ \textsc{PickR}$$

Fig. 7. Sequential Cost Semantics

new relations $\to_L$ and $\to_R$. The relation $\to_L$ ensures that the left component of the pair is evaluated completely before the right component steps. The relation $\to_R$ is analogous. These relations forbid the interleaving evaluation of the left and right components. Thus, if we fix the ordering between components, the evaluation becomes deterministic.

The sequential cost semantics introduces PickL and PickR to "pick" the order of execution with higher space usage. We refer to the space usage computed by the sequential cost semantics as the ***unordered reachable space*** or $\mathbf{R}^*$. We use this as a baseline to compare with parallel execution.

Theorem 6.1 (Space Bound). *Given a determinacy-race-free program with unordered reachable space R^*, its parallel execution requires at most $(c_2{\cdot}\kappa){\cdot}R^*{\cdot}P$ memory, where c_2 is the space efficiency of the collector and $\kappa > 1$ is an adjustable parameter in the collection policy.*

Proof. In our collection policy, each worker maintains a counter and keeps the size of its heap set within κ times this counter. If the size exceeds this limit, then the worker collects. For worker p, $|M_p| < \kappa{\cdot}\lambda_p$, where λ_p denotes the counter and $|M_p|$ denotes the size of its heap set. By Lemma 6.2 (stated and proved below) the counter $\lambda_p \leq R^*$. Thus, the size of M_p is less than $\kappa{\cdot}R^*$ when the worker is not collecting. The collection algorithm requires at most $c_2{\cdot}|M_p|$ memory when it collects. Thus, the maximum space used by worker p is $c_2{\cdot}\kappa{\cdot}R^*$. □

Bounding the counter. After every collection, the worker updates its counter to the size of memory that survived the collection. Because the counter is not updated otherwise, its value is bounded by this size. Suppose worker p starts a collection at step t, with the root set ρ^t and remembered set σ_p^t. With a remembered set, the collection algorithm only uses the pointers within the worker's heap set to compute reachability. Let $E_p^+(\rho^t \cup \sigma_p^t)$ be the set of reachable locations. After every collection, the counter λ_p is set to $|E_p^+(\rho^t \cup \sigma_p^t)|$. We bound its size as follows:

Lemma 6.2. *At any step t of a parallel execution, the size of objects reachable from the root set and the remembered set is bounded by the unordered reachable space. Equivalently, $|E_p^+(\rho^t \cup \sigma_p^t)| \leq R^*$.*

Proof. In the sequential semantics (Figure 7), the rules ExL and ExR allow a choice on the evaluation order of parallel pairs. We define a particular order by specifying a precedence relation $(<_p)$ on every pair. Suppose at step t of some parallel execution $\mathcal{P}$, the worker p is executing task

T_A. Let the root to leaf path in the task tree be $\mathcal{T}_{\text{path}} = T_{A0}, T_{A1} \ldots T_{An}.$, where $T_{An} = T_A$. Also, let $T_{B1} \ldots T_{Bn}$ be the siblings of the tasks on $\mathcal{T}_{\text{path}}$. We define the precedence relation for tasks in $\mathcal{T}_{\text{path}}$ (and their siblings) as follows:

(1) If the task T_{Bi} has terminated by step t, then $T_{Bi} <_p T_{Ai}$.
(2) Otherwise, if T_{Bi} has not terminated by step t, then $T_{Ai} <_p T_{Bi}$

For the other siblings, let the relation $<_p$ be defined arbitrarily. This relation orders every parallel pair and defines a sequential execution $\mathcal{S}$. The sequential execution starts with the root task T_{A0}. Then, the order of execution for any two siblings T_1 and T_2, is given by the precedence relation. If $T_1 <_p T_2$ then T_1 is evaluated completely before T_2 is executed. Otherwise T_2 is evaluated first.

Since we only consider deterministic programs, there is some step (t') at which $\mathcal{S}$ executes the instruction that worker p executes at step t. Let $\mathcal{P}^t$ and $\mathcal{S}^{t'}$ denote the set of instructions that have been executed till step t and t' respectively. We show that the reachability in the heap set M_p is exactly the same after executing the instructions in $\mathcal{P}^t$ and $\mathcal{S}^{t'}$.

Let $\mathcal{T}_{Mp}$ be the set of tasks that correspond to heaps in M_p. Let $\mathcal{T}_{Sp}$ be the set of successor tasks that were stolen from worker p and, let $\mathcal{T}_{Dp} \supseteq \mathcal{T}_{Sp}$ be the set of all the tasks in $\mathcal{T}_{Sp}$ and their descendants. The tasks in $\mathcal{T}_{Sp}$ are siblings of tasks on $\mathcal{T}_{\text{path}}$. Because the heap hierarchy is disentangled and there are no cross pointers, i.e., the reachability in M_p only depends on pointers from ancestors and descendants. Since ancestor tasks are suspended, the reachability in M_p only depends on pointers created/deleted by two types of tasks: (i) Tasks in $\mathcal{T}_{Mp}$, and (ii) Tasks in $\mathcal{T}_{Dp}$. First, consider a task $T \in \mathcal{T}_{Mp}$. The following two cases arise:

(1) $T \in \mathcal{T}_{\text{path}}$. By definition, all tasks on the path are ancestors of T_A. Since at steps t and t', both executions are at the same instruction of task T_A, they have executed the same instructions of ancestors of T_A. Thus, exactly the same instructions of T are in $\mathcal{P}^t$ and $\mathcal{S}^{t'}$.
(2) $T \notin \mathcal{T}_{\text{path}}$. We use the following property of tasks in $\mathcal{T}_{Mp}$ [2]:

$$T \notin \mathcal{T}_{\text{path}} \text{ iff } T \text{ is a passive leaf and a sibling of some task on } \mathcal{T}_{\text{path}}$$

By this property, T is passive (has terminated) and is a sibling of some task $T' \in \mathcal{T}_{\text{path}}$. Because task T has terminated by step t, it is ordered before its sibling T', i.e., $T <_p T'$. Thus, the task T must have been evaluated completely in $\mathcal{S}^{t'}$ and has terminated by step t'.

Hence, for all the tasks of the first type, the same instructions have been executed in $\mathcal{P}^t$ and $\mathcal{S}^{t'}$.

Now consider a task $T \in \mathcal{T}_{Dp}$. We refer to tasks in $\mathcal{T}_{Dp}$ as *foreign*. Recall that the remembered sets are updated so that no action of a foreign task changes reachability in M_p (Section 5.2). Thus, at step t in the parallel execution $\mathcal{P}$, reachability is computed as if T has not executed at all. Let $T' \in \mathcal{T}_{Sp}$ be defined as follows:

$$T' = \left\{ \begin{array}{l} T, \text{ if } T \in \mathcal{T}_{Sp} \\ T', \text{ if } T \notin \mathcal{T}_{Sp} \text{ and } T' \text{ is that ancestor of } T \text{ which is in } \mathcal{T}_{Sp} \end{array} \right\}$$

Since T has not terminated by step t, T' has not terminated as well. By definition of the set $\mathcal{T}_{Sp}$, T' is a sibling of some task in $\mathcal{T}_{Mp}$. By construction, T' is ordered later than its sibling in the sequential execution and has not been executed (because the sibling has not terminated). Thus, in the sequential execution T has really not executed till step t'.

Hence for both types of tasks same instructions of $\mathcal{P}^t$ and $\mathcal{S}^{t'}$ are considered for reachability. The reachable memory in the heaps of M_p is thus identical in both. Note that R^* is an upper bound on the reachable space of sequential execution. Thus, $|E_p^+(\rho^t \cup \sigma_p^t)| \leq R^*$. □

[2]We present this property without proof. The proof follows directly from description in Section 4

7 WORK BOUND

The collection algorithm performs $c_1 \cdot |M|$ units of work to collect $|M|$ amount of memory. We prove the following bound on the work done in all the collections:

Theorem 7.1 (Work Bound). *If a program's execution requires W units of work, then the work done in garbage collection is upper bounded by $k' \cdot (W + P \cdot R^*)$, where $k' = c_1 \cdot \frac{\kappa}{\kappa - 1}$, κ is set by the collection policy, c_1 is the work efficiency of the collector and R^* is the unordered reachable space.*

Proof. Let $|M_p^i|$ be the size of heap set before the i*th* collection on worker p. If λ_p^{i+1} is the size of heap set after the collection, then the memory reclaimed is $(|M_p^i| - \lambda_p^{i+1})$. The memory reclaimed by all the collections can not be greater than the memory allocated by the program. Thus, if worker p performs n_p collections and α is the total memory allocated then:

$$\sum_p \sum_{i=1}^{i=n_p} (|M_p^i| - \lambda_p^{i+1}) \leq \alpha \tag{1}$$

In our collection policy, a worker starts a collection only when the size of its heap set grows beyond κ times its counter, i.e, the worker p starts the i*th* collection because $|M_p^i| \geq \kappa * \lambda_p^i$. Moreover, by Lemma 6.2, the value of the counter λ_p does not exceed R^*. Thus, it follows that:

$$\sum_p \sum_{i=1}^{i=n_p} \lambda_p^{i+1} \leq \sum_p \sum_{i=1}^{i=n_p} \lambda_p^i + \sum_p \lambda_p^{(n_p+1)} \leq \sum_p \sum_{i=1}^{i=n_p} \frac{|M_p^i|}{\kappa} + P \cdot R^*$$

After substituting this in Equation 1, it follows that:

$$\sum_p \sum_{i=1}^{i=n_p} |M_p^i| \leq (\alpha + P \cdot R^*) \cdot \frac{\kappa}{\kappa - 1} \leq (W + P \cdot R^*) \cdot \frac{\kappa}{\kappa - 1}$$

We assume that allocation of one unit of memory requires one unit of work and thus, $\alpha \leq W$. The total memory traced in all collections is $(\sum_p \sum_{i=1}^{i=n_p} |M_p^i|)$. Thus, the total work done in collections is upper-bounded by $c_1 \cdot (\sum_p \sum_{i=1}^{i=n_p} |M_p^i|) \leq c_1 \cdot (W + P \cdot R^*) \cdot \frac{\kappa}{\kappa - 1}$. □

8 IMPLEMENTATION

We implement our techniques on top of MaPLe (MPL) [Westrick et al. 2020]. MPL extends the MLton compiler (a compiler for Standard ML) to support nested parallelism by providing a primitive **par**: `(unit → α) * (unit → β) → α * β` which takes two functions and executes them in parallel to return their results. Our new implementation, which we call MPL* [3], differs mainly in how it garbage-collects heaps. That is, while MPL is unable to collect shallow suspended heaps, our MPL* is able to perform garbage collection on all heaps.

Scheduling Tasks and Heaps. For load-balancing tasks across worker threads (OS threads), MPL* uses a work-stealing scheduler with private deques. Whenever a new task begins execution, the scheduler creates a new heap for it. The implementation is faithful to the pseudo-code described in Section 5, except for surrender. The pseudo-code assigns sibling heaps to the same worker as soon as one of them becomes passive. We implement a lazy form of surrendering in which the heaps are transferred only after both siblings are ready to join.

The task abstraction is a one-shot continuation with an additional data structure that stores its depth, its parent task and its heap. A heap is a doubly-linked list of fixed-size blocks (chunks) in

[3] https://github.com/MPLLang/mpl/tree/popl21-artifact

a global/uniform address space. This makes it possible to merge (and surrender) heaps without copying any data; their block-lists are just linked together.

Remembered Sets. As described in Section 5.2, each worker's remembered set maintains pointers from descendants at other worker's heaps. We instead maintain (up) pointers from every descendant heap, irrespective of which worker it is on. This simplification allows us to garbage-collect each internal heap independently. It amounts to snapshotting internal heaps at the time of the fork, i.e., no object of an internal heap becomes unreachable while its descendants execute. To that end, we maintain a remembered set for each heap (instead of maintaining one per worker). The remembered set of a heap consists of three components: up pointers from its descendants, internal pointers deleted by descendants, and down pointers from ancestors.

Each time a task forks, its subtasks' continuations are copied and added to its heap's remembered set. The continuations contain all the up pointers to the parent heap. Moreover, we track every update to mutable data using a write-barrier. An update may delete a pointer from $x[i]$ (field i of object x) to y. To prevent updates from changing reachability in internal heaps, the write-barrier checks if x and y are in the same internal heap and, adds an entry (x, i, y) to the remembered set. The write-barrier also checks if the update results in a down pointer. If an update creates a pointer from $x[i]$ to y, the barrier checks if x is in an ancestor heap. If so, it adds the entry (x, i, y) to the remembered set of the heap containing y. The write-barrier uses blocks to identify the depth of an object. Since the block sizes are fixed, the block of an object can be identified by zeroing the low-order bits of its memory address. Each block is associated with a *descriptor* that points to the meta-data of the heap it belongs to. The heap's meta-data tracks its depth.

Garbage Collection. For performance reasons, we use different algorithms for different heaps. Because disentanglement forbids cross pointers, the objects in leaf heaps are not referenced by other leaves. Internal heaps whose siblings are still in the deque also satisfy this property. We refer to such heaps as ***private*** heaps because the concurrently executing leaves do not have pointers into them. This property allows us to garbage-collect private heaps with a moving/copying collector. For better efficiency, the collector starts with a ***promotion phase*** that copies objects referenced by down pointers to the corresponding ancestor heaps. A promotion may create new down pointers, so the phase is repeated until no down pointers remain. The performance benefit of promoting objects is two-fold: (i) A promoted object is not copied until the ancestor becomes a leaf and, (ii) No down pointers remain in the remembered set. After the promotion phase, the tracing phase performs a Cheney-style collection. It copies the reachable objects to new locations and updates all references to them. Because the objects are private, we do not synchronize for updating the references at any step. MPL's runtime performs this copying collection on private heaps. MPL* extends it with the collection of non-private internal heaps.

Internal heaps that are not private are collected using the mark and sweep algorithm. Because remembered sets snapshot the internal heaps, the collection can be done independently for every heap. In our implementation, the worker pauses its task until it finishes the collection of all the heaps. However, we specialize the root heap for better parallel performance: if the root heap belongs to a worker's heap set, it does not collect it right away. Instead, the worker creates another task for collecting the root heap and adds it to its deque. The task's continuation is designed to contain all the (collector) roots for the heap. The continuation can be stolen by another worker to collect the heap concurrently. While the worker (that stole) collects the root heap, the root task may become a leaf and start adding new objects to the root heap. To avoid synchronization between the allocating worker and the collecting worker, we create a secondary root heap for new allocations. The secondary root is the only child of the primary root. However, unlike other parent and children heaps, we do not maintain down pointers or up pointers between the root heaps. This is because the

secondary root is not subject to collection until the primary root is collected. After the collection at primary root completes, the root heaps are merged.

9 EVALUATION

To evaluate our implementation of the proposed techniques, we present multiple comparisons. First, we present a high-level "sorting competition" between MPL* (our implementation) and two state-of-the-art parallel systems with automatic memory management: Java and Go. Next, we consider a suite of parallel benchmarks from numerous problem domains, including graph analysis, numerical algorithms, computational geometry, raytracing, image and audio manipulation, text processing, sorting, etc. On these benchmarks, we compare the performance of MPL* against MLton and MPL to determine the overheads, scalability, and space benefits of our techniques. These comparisons collectively demonstrate that

(1) In comparison to state-of-the-art procedural languages with automatic memory management, MPL* can scale better while using comparable amount of space.
(2) MPL* has low overhead, in terms of both time and space, in comparison to a fast, well-established sequential baseline.
(3) MPL* scales well up to high core counts.
(4) In comparison to MPL, our MPL* has a small penalty in time but uses significantly less space: up to -99% on one core, and up to -86% on 70 cores.

9.1 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, running Ubuntu version 16.04.6. For each benchmark, we run 10 trials and report averages, where each trial yields one time measure and one space measure. Each trial runs the benchmark 10 times back-to-back (in the same program instance) and measures the cumulative time taken for these 10 runs. We measure the space usage of one trial with *maximum residency*, as reported by Linux with `/usr/bin/time -v`. This is a system-independent measure, allowing us to compare space usage across systems, despite differences in memory management (e.g. heap architecture).

To account for warmup and initialization costs, we begin each trial with 5 warmup runs of the benchmark, which are excluded from the timing results. Note however that the space numbers reported include the costs of initialization and warmup.

In the sorting competition, for Java we use OpenJDK `1.8.0_222` with `-XX:+UseParallelGC`, and for Go we use version `1.8.1` with default settings.

For MPL, MPL*, Java, and Go, we write T_p and R_p respectively to denote the time and space usage on p processors. For MLton, we write T_s and R_s. All timings are in seconds and all space numbers are in gigabytes (GB).

9.2 Parallel ML Benchmark Overview

Here we describe the 15 benchmarks used for the MLton and MPL comparisons in Section 9.4. Note that the reported timings and max residency results are cumulative over multiple runs of each benchmark, as described in Section 9.1.

Centrality computes single-source betweenness centrality, based on the Ligra implementation [Shun and Blelloch 2013]. The input is a randomly generated power-law graph [Chakrabarti et al. 2004] with approximately 16.7M vertices and 199M edges, symmetrized.[4] ***Dedup*** computes the set of unique words of an input text by first separating the text into words, and then deduplicating

[4]A symmetrized graph is an undirected graph where each edge is represented as two directed edges.

the words by hashing. The input text is approximately 60MB with 6.3M words and 99K unique words. ***Dense-Matrix Multiplication (dmm)*** multiplies two 1024×1024 dense matrices of 64-bit floating-point elements using the simple $O(n^3)$-work algorithm. ***Grep*** is an implementation of the Unix `grep` utility. The input text is 60MB with 6.3M lines, and the search pattern appears on 138K lines. ***Mergesort (msort)*** sorts 10M 64-bit integers. The input is uniformly random, generated by a hash function. ***Nearest-Neighbors (nn)*** computes all nearest neighbors within a set of 2D points (i.e. for each point, the nearest other point within the set) by constructing an intermediate quad-tree and then querying it in parallel. The input is 1M points distributed uniformly randomly within a square. ***Palindrome*** finds the longest (contiguous) substring which is a palindrome using a polynomial rolling hash. The input is 1M characters. ***Primes*** generates all prime numbers that are less than 100M (approximately 5.8M primes) with a parallel sieve. ***Quickhull*** computes the convex hull of 10M uniformly random points distributed within a circle. ***Random*** generates 1B pseudo-random 64-bit numbers with a hash function. ***Raytracer (ray)*** computes an image of 1000×1000 pixels by ray-tracing. ***Reverb*** applies an artificial reverberation effect to an audio file. The input is approximately 4 minutes long with a sample rate of 44.1 kHz at 16 bits per sample. ***Seam-carve*** is a parallel implementation of the seam-carving [Avidan and Shamir 2007] technique for content-aware rescaling. This benchmark removes 100 vertical seams from a panoramic image of approximately 1.5M pixels. ***Suffix-array*** computes the suffix array of a uniformly random input text of 10M characters. ***Tokens*** separates a text into tokens, using whitespace as delimiters. The input text is approximately 60MB with 6.3M tokens.

9.3 Sorting Competition

In this section we present the results of a "sorting competition" between MPL*, Java, and Go. We chose Java and Go because both are state-of-the-art parallel systems with automatic memory management. Other comparisons would also be possible: for example, MPL has previously been compared against both Cilk and Haskell in a similar experiment [Westrick et al. 2020]. We do not include these comparisons because Cilk does not have automatic memory management, and Haskell was found to not scale up to high core counts.

To conduct this experiment, we obtained highly optimized parallel sorting implementations for each system. The Java implementation is the standard `java.util.Arrays.parallelSort`, written by Doug Lea for the Java Fork/Join library [Lea 2000]. The Go implementation is a highly optimized samplesort (based in part on the PBBS samplesort [Blelloch et al. 2010; Shun et al. 2012]). For MPL*, we provide a mergesort. The input (identical in all cases) is an array of 100M 32-bit integers generated by a hash function, and we require that the input is not modified: the sorted result must be returned as a freshly allocated array. Note that the results are cumulative over multiple runs of the sorting routine, as described in Section 9.1.

The results are shown in Figures 8 and 9. Figure 8 gives the time and space usage of each system, and Figure 9 shows the speedups relative to the fastest sequential time (i.e. Java). In general, we observe that all systems perform within a factor two of one another in terms of both space and time. Go—although slowest on one core—has a memory footprint half the size of both MPL* and Java, which use approximately the same amount of memory. In terms of parallelism, Java only scales linearly up to approximately 20 processors, whereas both MPL* and Go scale nearly linearly up to 70 cores. Our MPL* is the fastest on large core counts by a wide margin. On 70 processors, MPL* is 35% faster than Go and nearly twice as fast as Java.

9.4 Comparison with MLton and MPL

In this section, we compare our MPL* against MPL and MLton using the benchmarks detailed in Section 9.2. We use MLton as a sequential baseline to compute overheads, speedups, and space

	Time		Space	
	T_1	T_{70}	R_1	R_{70}
MPL* (Ours)	**219**	**4.08**	**6.65**	**8.82**
Java	139	7.52	6.28	7.90
Go	272	6.23	3.77	3.37

Fig. 8. Sorting competition.

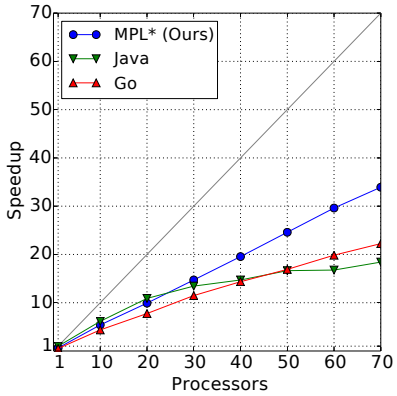

Fig. 9. Sorting competition speedups (Java T_1 used as baseline).

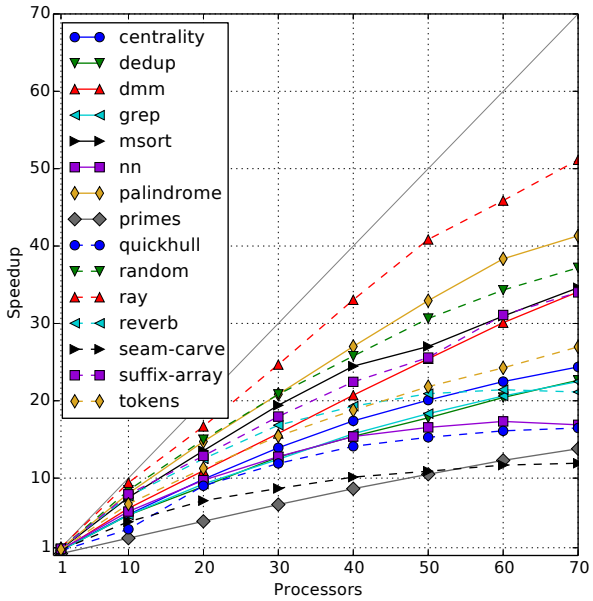

Fig. 10. Speedups of MPL* in comparison to MLton.

	T_s	T_1		Overhead		T_{70}		Speedup	
	MLton	MPL	**MPL* (Ours)**	MPL	**MPL* (Ours)**	MPL	**MPL* (Ours)**	MPL	**MPL* (Ours)**
centrality	116	163	164 (+1%)	1.41	1.41	4.60	4.75 (+3%)	25	24
dedup	20.4	30.1	31.1 (+3%)	1.48	1.52	0.87	0.90 (+3%)	23	23
dmm	18.7	28.7	29.3 (+2%)	1.53	1.57	0.54	0.55 (+2%)	35	34
grep	5.61	9.10	8.73 (-4%)	1.62	1.56	0.24	0.25 (+4%)	23	22
msort	15.8	19.3	19.3 (+0%)	1.22	1.22	0.45	0.46 (+2%)	35	34
nn	11.2	13.6	14.5 (+7%)	1.21	1.29	0.53	0.66 (+25%)	21	17
palindrome	16.1	16.8	17.9 (+7%)	1.04	1.11	0.37	0.39 (+5%)	44	41
primes	17.7	76.1	73.0 (-4%)	4.30	4.12	1.28	1.28 (+0%)	14	14
quickhull	9.95	12.5	16.2 (+30%)	1.26	1.63	0.53	0.60 (+13%)	19	17
random	45.0	58.1	52.6 (-9%)	1.29	1.17	1.11	1.21 (+9%)	41	37
ray	26.7	25.8	25.7 (+0%)	0.97	0.96	0.51	0.52 (+2%)	52	51
reverb	27.5	32.2	29.9 (-7%)	1.17	1.09	1.09	1.30 (+19%)	25	21
seam-carve	119	161	158 (-2%)	1.35	1.33	9.23	9.99 (+8%)	13	12
suffix-array	45.7	52.8	52.3 (-1%)	1.16	1.14	1.29	1.34 (+4%)	35	34
tokens	5.98	7.34	7.75 (+6%)	1.23	1.30	0.21	0.22 (+5%)	28	27

Fig. 11. Times (seconds), overheads, and speedups of MPL and MPL* in comparison to MLton. Percent differences of MPL* are relative to MPL.

blowup factors. All results are shown in Figures 10, 12, and 11. Figure 10 plots the speedups of MPL* across multiple processors counts. Figure 11 gives timings, overheads, and speedups. Figure 12 shows space (max residency) results and blowups. In Figures 11 and 12, the percent difference of MPL* with respect to MPL is shown in the T_p and R_p columns for MPL*. Negative percent differences indicate improvements (in time or space).

	R_s	R_1		Blowup$_1$		R_{70}		Blowup$_{70}$	
	MLton	MPL	**MPL* (Ours)**	MPL	**MPL* (Ours)**	MPL	**MPL* (Ours)**	MPL	**MPL* (Ours)**
centrality	25	36	17 (**-53%**)	1.4	0.7	38	20 (**-47%**)	1.5	0.8
dedup	3.6	11	3.7 (**-66%**)	3.1	1.0	10	9.7 (**-3%**)	2.8	2.7
dmm	0.10	0.20	0.13 (**-35%**)	2.0	1.3	0.49	0.46 (**-6%**)	4.9	4.6
grep	2.6	5.3	1.2 (**-77%**)	2.0	0.5	3.3	2.8 (**-15%**)	1.3	1.1
msort	2.2	2.9	1.3 (**-55%**)	1.3	0.6	3.6	2.4 (**-33%**)	1.6	1.1
nn	1.3	4.1	1.2 (**-71%**)	3.2	0.9	4.4	3.8 (**-14%**)	3.4	2.9
palindrome	0.20	0.38	0.15 (**-61%**)	1.9	0.7	0.67	0.65 (**-3%**)	3.4	3.2
primes	1.5	2.6	0.86 (**-67%**)	1.7	0.6	3.0	1.8 (**-40%**)	2.0	1.2
quickhull	3.5	4.0	3.0 (**-25%**)	1.1	0.9	4.9	5.3 (+8%)	1.4	1.5
random	121	120	48 (**-60%**)	1.0	0.4	120	48 (**-60%**)	1.0	0.4
ray	0.40	1.0	0.27 (**-73%**)	2.5	0.7	1.8	1.5 (**-17%**)	4.5	3.8
reverb	8.3	45	4.2 (**-91%**)	5.4	0.5	48	6.8 (**-86%**)	5.8	0.8
seam-carve	0.41	47	0.35 (**-99%**)	114.6	0.9	75	15 (**-80%**)	182.9	36.6
suffix-array	6.3	13	2.4 (**-82%**)	2.1	0.4	14	3.4 (**-76%**)	2.2	0.5
tokens	3.8	3.9	1.6 (**-59%**)	1.0	0.4	4.3	3.4 (**-21%**)	1.1	0.9

Fig. 12. Max residencies (GB) and space blowups. Percent differences of MPL* are relative to MPL.

Speedups. The ***speedup*** of MPL* (and similarly for MPL) on p processors is given by T_s/T_p. This quantity summarizes the benefit of parallelism as an improvement factor relative to a fast sequential baseline. A speedup of $T_s/T_p = p$ would indicate perfect speedup, i.e. full utilization of all p processors. Perfect speedup is uncommon, even for embarrassingly parallel benchmarks, due to overheads of parallelism and memory bottlenecks on modern multicores. Typically, we expect to see speedups scale linearly with the number of processors but then plateau as the memory bandwidth of the machine is reached, particularly for "memory-bound" benchmarks.

In Figure 10, we observe two primary behaviors, as expected. Most benchmark can be classified as either compute-bound or memory-bound: the compute-bound benchmarks (e.g. ray, palindrome, dmm, primes) all scale approximately linearly, whereas the memory-bound benchmarks (e.g. reverb, nn, quickhull, seam-carve) each initially scale linearly and then plateau as the memory bandwidth of the machine is reached. In all cases, as the number of processors increases, the speedup either stays approximately the same or increases.

In Figure 11, we see that on 70 cores, MPL* achieves between 12× and 51× speedup over MLton. Across the board, these speedups are similar to those obtained by vanilla MPL: on 70 cores, MPL* is at most 25% slower than MPL, with 12 out of 15 benchmarks within a ±10% difference. The fact that MPL* is just as fast as MPL on high core counts suggests that the possible $O(R^*P)$ additional work due to GC (Theorem 7.1) could be loose in practice.

For MPL*, the lowest speedup (12× on 70 cores) is seam-carve, which is expected due to three factors. First, seam-carving not highly parallel: in an image of width w and height h, seam-carving has $O(wh)$ work and $O(h)$ span, leaving only $O(w)$ parallelism, which for typical images is small. Second, seam-carving is memory-bound: it only does a small amount of compute (a few arithmetic instructions) per memory access. Third, seam-carving has a high allocation and reclamation rate: this particular implementation is "pure" in the sense that removing one seam does not modify the input image, so in total the benchmark allocates approximately 100 copies of the input image, which stresses the memory management system. In light of these bottlenecks, a speedup of 12× for seam-carving is admirable. Another case of low speedup is primes, which is explainable entirely due to its overhead (discussed below): in fact, MPL* has an excellent "self-speedup" of $T_1/T_{70} = 57$.

Overheads. To summarize the cumulative impact of parallelization as a slowdown factor, we compute the ***overhead*** of MPL* as T_1/T_s where T_1 is the time on 1 processor and T_s is MLton's sequential runtime. Overheads larger than 1 indicate that MPL* is that many times slower than MLton. We also compute the overhead of vanilla MPL for comparison.

In almost all cases, we observe that MPL* is less than 60% slower than MLton. The one exception is the primes benchmark, where MPL* is more than 4× slower than MLton on one processor (and despite this, MPL* still manages to achieve 14× speedup over MLton). Note that MPL also has a similar overhead on this benchmark, suggesting that there are differences in the underlying MLton and MPL compilers influencing the performance of this benchmark.

In general, the differences between MPL* and MPL are small on one processor: MPL* is anywhere from 9% faster to 30% slower than MPL, and all benchmarks except for one show less than a ±10% difference. Overall, the time costs of our extensions to MPL appear to be relatively low.

Space Efficiency. In Figure 12, we immediately observe that MPL* offers significant space improvements over MPL. On a single processor, these improvements range from 25% to 99%. On 70 processors, MPL* uses up to 86% less memory.

To better understand the relationship between sequential and parallel execution, we also consider here the *space blowup* of the parallelism. Denoted B_p, the ***space blowup*** of MPL* (and similarly for MPL) on p processors is defined as $B_p = R_p/R_s$ where R_p is the max residency of MPL* on p processors, and R_s is the max residency of MLton. The blowup summarizes the space overhead of parallelism, which broadly consists of two factors: (1) differences in memory management, e.g. concurrent versus stop-the-world GC, and (2) the inherent additional space required to execute a program in parallel.

We can relate blowup to Theorem 6.1 by taking R_s as an approximation of kR^* where k is the "sloppiness" factor of MLton's GC implementation.[5] MLton will use up to a factor k more space than the program needs, so the approximation $R_s \approx kR^*$ should be accurate as long as each benchmark needs R^* space sequentially. We believe this is the case for the following reasons. First, note that for MLton, we replaced parallel pairs with sequential pairs. This induces a left-then-right sequential schedule, which is one of the possible schedules for witnessing R^* maximum reachable memory. But this immediately raises the question: is it possible that some *other* sequential schedule has a much larger maximum reachability? For well-parallelized programs, this seems unlikely, because parallelization relies upon good load-balancing of all costs, including allocation. Therefore, it seems reasonable to assume that all sequential schedules of a well-parallelized program need approximately R^* space.

Taking $R_s \approx kR^*$, Theorem 6.1 suggests that we should expect for MPL* to have approximately at most linear blowup, i.e. $B_p \lessapprox p$. Indeed, in Figure 12, we observe in almost all cases that $B_p \leq p$. (The only exception is dmm with $B_1 = 1.3$, where the total memory used is small, so the difference could be due to unavoidable additive overheads such as the extra memory needed for the scheduler.) There is only one benchmark where the parallel blowup is large: seam-carve, with $B_{70} \approx 37$. Seam-carve is a challenging benchmark for a number of reasons (described above in the *Speedups* paragraph); in particular, it is memory intensive, with a high rate of allocation. Note that the space blowup on seam-carve is nevertheless within expected bounds. Furthermore, MPL* sees one of its largest space improvements over vanilla MPL on this benchmark (between 80% and 99% improvement).

Surprisingly, we observe that the blowup on 70 processors is often much less than 70. Out of 15 benchmarks, 9 of them have $B_{70} < 2$, and 5 benchmarks even have $B_{70} < 1$. On these 5 benchmarks (centrality, random, reverb, suffix-array, and tokens), MPL* is more efficient than MLton in terms of both space and time: it uses less space overall while being up to 21 and 37 times faster.

[5]For MLton with default settings, $k \approx 16$.

To summarize, the space measurements show that

(1) MPL* offers significant space improvements over MPL.
(2) MPL* uses only a small amount of extra space to achieve significant parallel speedup over MLton. On some benchmarks, MPL* consistently uses less space than MLton.
(3) Theorem 6.1 appears to hold in practice, even despite the simplifications made for a practical implementation. It is also possible that the $O(R^*P)$ space bound is very loose in practice.

10 RELATED WORK

Cost Semantics. To establish our bounds, we use a cost semantics that keeps track of work, space usage, and yields a task tree of the computation. The task tree allow us to reason about the intrinsic properties of the computation (threads/concurrency created and their relationships). For our bounds, we use a notion of reachability that accounts for the different orders in which parallel pairs may be executed in a sequential computation. This notion is quite interesting: it does not account for all interleavings of the parallel pairs, but just two specific ones, where the left completes before the right starts, and the right completes before the left starts.

Cost semantics have proved to be an effective tool for reasoning about non-trivial properties of the computation. The idea of cost semantics goes back to the early 90s [Rosendahl 1989; Sands 1990a] and has been shown to be particularly important in high-level languages such as lazy (e.g., Sands 1990a,b; Sansom and Peyton Jones 1995) and parallel languages (e.g., Acar et al. 2018, 2016a; Blelloch and Greiner 1995, 1996; Spoonhower et al. 2008). Aspects of our cost semantics resemble prior cost semantics used in parallel languages [Acar et al. 2018, 2016a; Blelloch and Greiner 1996; Spoonhower et al. 2008], though the specifics such as our use of task trees and our specific notion of reachability measure differ.

Scheduling. Nearly all modern parallel programming languages today rely on a scheduler to distribute threads over hardware resources so that the programmer does not have to control it manually. This is important as, manual thread scheduling can be very challenging, especially for multiprogrammed environments.

Early results on scheduling goes back to seventies and to the work of mathematician Brent [Brent 1974]. Brent's result was later generalized to greedy schedulers [Arora et al. 2001; Eager et al. 1989]. Blumofe and Leiserson [Blumofe and Leiserson 1999] and later Arora, Blumofe, and Plaxton [Arora et al. 1998]. proved that randomized work stealing algorithm can generate efficient greedy schedules "on-the-fly", also on multiprocessor systems. More recent work extended these techniques to account for the cost of thread creation [Acar et al. 2018, 2016a; Tzannes et al. 2014] and responsiveness or interactivity [Muller et al. 2020; Muller and Acar 2016; Muller et al. 2017, 2018a] Our implementation is based on a variant of the work stealing algorithm based on private deques [Acar et al. 2013].

The space consumption of various scheduling algorithms have also been studied [Agarwal et al. 2007; Blelloch et al. 1999; Blumofe and Leiserson 1998; Narlikar and Blelloch 1999], as well as their locality properties [Acar et al. 2015, 2002; Blelloch et al. 2011; Blelloch and Gibbons 2004; Chowdhury and Ramachandran 2008; Lee et al. 2015; Spoonhower et al. 2009]. But, none of these works consider garbage collection and the impact of thread scheduling on garbage collection. For example Blumofe and Leiserson [Blumofe and Leiserson 1999] establish space bounds similar to ours but assume a restricted form of "stack-allocated computations" that use work stealing, where all memory is allocated on the stack, and all memory allocated by a function call is freed upon returning from that call. Stack allocation is a rather unrealistic assumption for most programming languages, because even non-managed languages such as C/C++ permit heap allocated objects. They assume instead that programs follow a specific allocation strategy, typically "stack allocation", where objects that are allocated by deeper calls cannot be returned without being copied explicitly.

One of our key contribution is the bounds accounting for heap allocated objects and garbage collection.

Memory Management. Since its early days in the Lisp language, automatic memory management has come a long way and has become a popular and a prominent feature of modern programming languages. The book by Jones et al. [Jones et al. 2011] discuss many garbage-collection techniques incorporating parallelism, concurrency, and real-time features. There is, however, relatively little work on the problem of parallel memory management for functional languages that support nested parallelism, where programs may create many (e.g., millions of) fine grained threads, which are scheduled by a (usually highly nondeterministic) scheduler.

Although our results are related to the work in the broader area of memory management, they are also different in an important aspect. Our primarily concern is the partitioning of memory between processors dynamically in a parallel computation so that each processor can decide individually and independently when to perform garbage collection (without synchronizing with others). For the actual garbage collection, we allow each processor to use any garbage collection algorithm as long as it meets certain basic criteria (Section 5.2). In our implementation, we use sequential copying and in-place concurrent collectors. But other collection algorithms, including multiprocessor, incremental algorithms could be used instead.

In the rest of this section, we describe some of the closely related work in the area of functional programming and in approaches that aim to provide provable guarantees.

There are many provably space and work efficient algorithms for garbage collection for uniprocessor computing models. Similar provable algorithms for multiprocessors or parallel systems are more scarce. One notable exception is the algorithm of Blelloch and Cheng [Blelloch and Cheng 1999; Cheng and Blelloch 2001], which is able to achieve tight space and time bounds. The algorithm, however, is primarily meant for real-time garbage collection and has several shortcomings, including its complex synchronization and load-balancing techniques, and its relatively liberal space usage [Bacon et al. 2003]. Follow-up work has overcome some of these limitations, though sometimes by assuming the uniprocessor model [Bacon et al. 2003; Pizlo et al. 2008]. These real-time algorithms may be used in conjunction with our heap scheduling algorithm in real-time applications.

Within the world of parallel functional programming, we can distinguish between two main architectural approaches to memory management, none of which has (until this paper) was able to establish tight space and work bounds.

The first approach uses 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 and Jones 2011]. This design is employed by the Doligez-Leroy-Gonthier (DLG) parallel collector [Doligez and Gonthier 1994; Doligez and Leroy 1993] and the Manticore garbage collector [Auhagen et al. 2011; Le and Fluet 2015]. They enforce 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. The Glasgow Haskell Compiler (GHC) uses a garbage collector [Marlow and Jones 2011] that follows a similar architecture but also employs techniques similar to Domani et al. [Domani et al. 2002]. The collector allows pointers from global to local heaps and relies on a read barrier to promote (copy) data to the global heap when accessed. Recent work on a multicore memory manager for OCaml uses several techniques to reduce the cost of promotions [Sivaramakrishnan et al. 2020]. None of these approaches can guarantee space and work/time bounds, because they rely on absence of pointers

from shared to private heaps; as a result common scheduling actions, such as migrating a thread or returning the result of a child task, can require copying (promoting) the objects reachable from a migrated thread to the shared heap.

The second approach is due to more recent work on disentanglement [Acar et al. 2015; Guatto et al. 2018; Raghunathan et al. 2016; Westrick et al. 2020]. In that work, the authors associate heaps with tasks rather that system-level threads or processors and organize the memory as a dynamic hierarchy that can be arbitrarily deep and grows and shrinks as the computation proceeds. Pointers between heaps that have ancestor-descendant relationships are allowed but cross pointers between concurrent heaps are not allowed. Therefore, disentangled parallel programs 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. The absence of cross-pointers usually require no significant loss of generality, because shared objects can be allocated in an ancestor heap, and many programs, including all determinacy-race-free programs, are disentangled [Westrick et al. 2020]. The primary focus of work on disentanglement so far has been to develop the dynamic memory architecture consisting of a tree of heaps and do not offer any guarantee on space usage.

Nearly all of the work on parallel functional languages organizes memory as a hierarchy of heaps; this general idea goes back to 1990s [Alpern et al. 1990; Krishnamurthy et al. 1993; Numrich and Reid 1998; Yelick et al. 1998]. More recent work includes Sequioa [Fatahalian et al. 2006] and Legion [Bauer et al. 2012]. These techniques are also remotely related to region-based memory management in the sequential setting [Fluet et al. 2006; Grossman et al. 2002; Hanson 1990; Ross 1967; Schwartz 1975; Tofte and Talpin 1997; Walker 2001], which allows objects to be allocated in specific regions, which can be deallocated in bulk.

Parallelism: Procedural and Functional Approaches. Many parallel programming languages based on procedural, object-oriented, and functional programming languages have been developed. 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], though it is difficult (if not impossible) to implement efficient parallel algorithms, such as the algorithms that we consider in our evaluation, by using safe primitives only. 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. The Go language is designed from grounds up with concurrency in mind.

Because these systems support memory effects or destructive updates, programs written in them are vulnerable to determinacy or data races [Allen and Padua 1987; Emrath et al. 1991; Mellor-Crummey 1991; Netzer and Miller 1992; Steele Jr. 1990]. Such races are notoriously difficult to avoid and can be harmful [Adve 2010; Bocchino et al. 2011, 2009; Boehm 2011]. Data races may be detected or even be eliminated 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], and 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]).

Functional programming languages typically offer substantial control over side effects, usually via powerful type systems [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], which help programmers to avoid race conditions. Notable functional parallel languages include several forms of a Parallel ML language [Acar et al. 2015; Fluet et al. 2008, 2011; Guatto et al. 2018; Raghunathan et al. 2016; Westrick et al. 2020], 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 and Jones 2011].

11 DISCUSSIONS

All the techniques proposed in this paper assume disentanglement. Because disentanglement is implied by determinacy-race-freedom [Westrick et al. 2020], it can be checked by using a known race-detector. But disentanglement is more general: programs with determinacy- and data-races are disentangled, as long as concurrently executing threads do not see each other's memory allocations. It turns out many interesting parallel programs do exactly that: they use data races but remain disentangled. For example, parallel graph algorithms use data races for improved efficiency but remain disentangled. A natural question thus is how to check for disentanglement. At this time, there are no known efficient algorithms or implementations for disentanglement checking.

In this paper, we considered fork-join programming model. This model has proved a good fit for compute intensive applications, ranging from scientific computing to graph processing and machine learning. There are other applications, however, that benefit from more expressive forms of parallelism. For example, interactive parallel applications are more naturally expressed with futures [Acar et al. 2016b; Halstead 1984] and would be difficult to express by using fork-join only [twi 2011; fac 2015; Muller et al. 2020; Muller and Acar 2016; Muller et al. 2017, 2018a,b]. It would be interesting to extend these techniques to more general models of computation including futures. Such an advance would require generalizing the disentanglement theory to futures first. This in-and-of-itself seems nontrivial, because futures allow for more complex synchonizations between concurrent threads, which may break disentanglement. Assuming that it is possible to extend disentanglement for futures, the memory management and heap scheduling techniques would also need to be extended accordingly. In addition to futures, another form of parallelism that is popular is the "async-finish" style, which can be viewed as a generalization of fork-join that allows for any number of parallel computations to join (instead of just two). This is a mild generalization and there does not appear to be significant difficulties in extending our techniques to support asynch-finish style.

12 CONCLUSION

We present techniques for provable space and work efficient memory management for nested-parallel languages. Our techniques apply both to purely functional and imperative programs that use destructive updates as long as they are disentangled. The key technical innovation behind our techniques is to partition the memory into smaller heaps and schedule (assign) these heaps to processors in such a way that each processor may collect its own partition of heaps independently of the others by using one of many garbage collection algorithms available in the literature. Our techniques are quite general and permit many different implementations. We present such an implementation and show that it delivers good performance.

ACKNOWLEDGMENTS

This work was partially supported by the National Science Foundation under grant numbers CCF-1408940 and CCF-1901381. We also thank our shepherd, Simon Peyton Jones, for making numerous suggestions to improve the paper.

REFERENCES

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

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

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

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

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

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

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

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

Shivali Agarwal, Rajkishore Barik, Dan Bonachea, Vivek Sarkar, R. K. Shyamasundar, and Katherine A. Yelick. 2007. Deadlock-free scheduling of X10 computations with bounded resources. In *SPAA 2007: Proceedings of the 19th Annual ACM Symposium on Parallelism in Algorithms and Architectures, San Diego, California, USA, June 9-11, 2007*. 229–240.

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. 1998. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures* (Puerto Vallarta, Mexico) *(SPAA '98)*. ACM Press, 119–129.

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.

Shai Avidan and Ariel Shamir. 2007. Seam carving for content-aware image resizing. In *ACM SIGGRAPH 2007 papers*. 10–es.

David F. Bacon, Perry Cheng, and V.T. Rajan. 2003. A Real-Time Garbage Collecor with Low Overhead and Consistent Utilization. In *Conference Record of the Thirtieth Annual ACM Symposium on Principles of Programming Languages (ACM SIGPLAN Notices)*. ACM Press, New Orleans, LA.

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 Blelloch and John Greiner. 1995. Parallelism in sequential functional languages. In *Proceedings of the 7th International Conference on Functional Programming Languages and Computer Architecture (FPCA '95)*. ACM, 226–237.

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

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

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

Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Harsha Vardhan Simhadri. 2011. Scheduling irregular parallel computations on hierarchical caches. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures* (San Jose, California, USA) *(SPAA '11)*. 355–366.

Guy E. Blelloch and Phillip B. Gibbons. 2004. Effectively sharing a cache among threads. In *SPAA* (Barcelona, Spain).

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

Guy E. Blelloch, 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* (Newport, Rhode Island, USA) *(SPAA '97)*. 12–23.

Guy E Blelloch, Phillip B Gibbons, and Harsha Vardhan Simhadri. 2010. Low depth cache-oblivious algorithms. In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*. 189–199.

Guy E. Blelloch and John Greiner. 1996. A provable time and space efficient implementation of NESL. In *Proceedings of the 1st ACM SIGPLAN International Conference on Functional Programming*. ACM, 213–225.

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* (Orlando, Florida, USA) *(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*.

Richard P. Brent. 1974. The parallel evaluation of general arithmetic expressions. *J. ACM* 21, 2 (1974), 201–206.

Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A recursive model for graph mining. In *SIAM SDM*.

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

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

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

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

Rezaul Alam Chowdhury and Vijaya Ramachandran. 2008. Cache-efficient dynamic programming algorithms for multicores. In *Proc. 20th ACM Symposium on Parallelism in Algorithms and Architectures* (Munich, Germany). ACM, New York, NY, USA, 207–216.

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

Derek L. Eager, John Zahorjan, and Edward D. Lazowska. 1989. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computing* 38, 3 (1989), 408–423.

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

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* (Tampa, Florida) *(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.

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* (Austin, Texas, United States) *(LFP '84)*. ACM, 9–17.

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

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* (Baltimore, Maryland, USA) *(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* (Portland, Oregon, USA) *(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* (San Diego, California, USA) *(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* (Edinburgh, United Kingdom) *(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* (San Diego, California, USA) *(POPL '14)*. ACM, New York, NY, USA, 257–270.

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

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

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

I-Ting Angelina Lee, Charles E. Leiserson, Tao B. Schardl, Zhunping Zhang, and Jim Sukha. 2015. On-the-Fly Pipeline Parallelism. *TOPC* 2, 3 (2015), 17:1–17:42.

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* (Orlando, Florida, USA) *(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.

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

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

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

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

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

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

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

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

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

Girija J. Narlikar and Guy E. Blelloch. 1999. Space-Efficient Scheduling of Nested Parallelism. *ACM Transactions on Programming Languages and Systems* 21 (1999).

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 5.0 2018. *OpenMP Application Programming Interface, Version 5.0.* Accessed in July 2018.

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

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

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

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.

Filip Pizlo, Erez Petrank, and Bjarne Steensgaard. 2008. A study of concurrent real-time garbage collectors. *ACM SIGPLAN Notices* 43, 6 (2008), 33–44.

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

Raghavan Raman, Jisheng Zhao, Vivek Sarkar, Martin 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* (Tucson, Arizona) *(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).

Mads Rosendahl. 1989. Automatic complexity analysis. In *FPCA '89: Functional Programming Languages and Computer Architecture*. ACM, 144–156.

D. 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/

David Sands. 1990a. *Calculi for Time Analysis of Functional Programs.* Ph.D. Dissertation. University of London, Imperial College.

David Sands. 1990b. Complexity Analysis for a Lazy Higher-Order Language. In *ESOP '90: Proceedings of the 3rd European Symposium on Programming*. Springer-Verlag, London, UK, 361–376.

Patrick M. Sansom and Simon L. Peyton Jones. 1995. Time and space profiling for non-strict, higher-order functional languages. In *Principles of Programming Languages* (San Francisco, California, United States). 355–366.

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

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

68–70.

KC Sivaramakrishnan, Stephen Dolan, Leo White, Sadiq Jaffer, Tom Kelly, Anmol Sahoo, Sudha Parimala, Atul Dhiman, and Anil Madhavapeddy. 2020. Retrofitting Parallelism onto OCaml. *arXiv preprint arXiv:2004.11663* (2020).

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.

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

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

Daniel Spoonhower, Guy E. Blelloch, Robert Harper, and Phillip B. Gibbons. 2008. Space Profiling for Parallel Functional Programs. In *International Conference on Functional Programming*.

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

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.

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.

Alexandros Tzannes, George C. Caragea, Uzi Vishkin, and Rajeev Barua. 2014. Lazy Scheduling: A Runtime Adaptive Scheduler for Declarative Parallelism. *TOPLAS* 36, 3, Article 10 (Sept. 2014), 51 pages.

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

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. In *Proceedings of the First workshop on Semantics, Program Analysis and Computing Environments for Memory Management (SPACE'01)*. London. http://www.diku.dk/topps/space2001/program.html#DavidWalker

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

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

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