Hierarchical Memory Management for Parallel Programs
Ram A Raghunathan, Stefan A Muller, Umut A Acar, Guy A Blelloch

To cite this version:
Ram A Raghunathan, Stefan A Muller, Umut A Acar, Guy A Blelloch. Hierarchical Memory Management for Parallel Programs. Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, Sep 2016, Nara, Japan. 10.1145/3022670.2951935. hal-01416237

HAL Id: hal-01416237
https://inria.hal.science/hal-01416237
Submitted on 14 Dec 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Hierarchical Memory Management for Parallel Programs

Ram Raghunathan* Stefan K. Muller* Umut A. Acar*† Guy Blelloch*
†Carnegie Mellon University, USA †Inria, France (ram.r, smuller, umut, blelloch)@cs.cmu.edu

Abstract
An important feature of functional programs is that they are parallel by default. Implementing an efficient parallel functional language, however, is a major challenge, in part because the high rate of allocation and freeing associated with functional programs requires an efficient and scalable memory manager.

In this paper, we present a technique for parallel memory management for strict functional languages with nested parallelism. At the highest level of abstraction, the approach consists of a technique to organize memory as a hierarchy of heaps, and an algorithm for performing automatic memory reclamation by taking advantage of a disentanglement property of parallel functional programs. More specifically, the idea is to assign to each parallel task its own heap in memory and organize the heaps in a hierarchy/tree that mirrors the hierarchy of tasks.

We present a nested-parallel calculus that specifies hierarchical heaps and prove in this calculus a disentanglement property, which prohibits a task from accessing objects allocated by another task that might execute in parallel. Leveraging the disentanglement property, we present a garbage collection technique that can operate on any subtree in the memory hierarchy concurrently as other tasks (and/or other collections) proceed in parallel. We prove the safety of this collector by formalizing it in the context of our parallel calculus. In addition, we describe how the proposed techniques can be implemented on modern shared-memory machines and present a prototype implementation as an extension to MLton, a high-performance compiler for the Standard ML language. Finally, we evaluate the performance of this implementation on a number of parallel benchmarks.

Categories and Subject Descriptors D.3.4 [Programming Languages]: Processors — Memory management (garbage collection); D.1.3 [Programming Techniques]: Concurrent Programming — Parallel Programming; D.4.1 [Operating Systems]: Process Management — Scheduling

General Terms Languages, Performance

Keywords Languages, Memory Management, Parallelism, Scheduling

1. Introduction
In the past two decades, there has been a large body of work on both parallel garbage collection and parallel scheduling. In this paper, we propose an approach for coupling the two together, which is particularly useful in functional languages. The basic idea is to organize memory in a way that mirrors the structure of parallelism in the computation and takes advantage of the independence of parallel tasks to perform garbage collection. While this general idea may be applicable to the broader array of parallel functional programming languages, we focus here on strict languages, such as the ML family, extended with support for nested parallelism.

More specifically, instead of using a single shared heap [6, 9, 21, 26, 32] or a two-level heap consisting of local processor heaps and a global heap [5, 8, 23, 25, 31, 37], we suggest a hierarchy of heaps (tree of heaps) that is tied to the nesting of tasks. In nested parallel (fork-join) parallelism, the tasks form a natural nesting, and memory that is allocated and referenced has a structure that is related to this nesting, which we would like to leverage. In strict functional computations, pointers only point up the hierarchy, creating what we refer to as a disentangled heap. Furthermore, what is often understood as temporal locality in sequential programs becomes nested task locality in nested parallel computations. For example, in divide-and-conquer algorithms, each node in the recursion tree has some locality within its subtree [14, 45].

The motivation for organizing the heap into a hierarchy is three-fold. Firstly, it allows for simpler parallel collection since the nodes in the tree can be collected individually and separately, especially if they can be kept disentangled. Secondly, the hierarchical organization of heaps is better suited for supporting memory locality at multiple levels of a modern cache hierarchy, for which some of the higher levels are shared among cores on multiprocessors. By having the size of heaps grow going up the levels, and moving values up the heap as they are collected, the approach can be seen as the natural extension of a multi-level generational collector. Finally, the organization allows us to naturally tie memory management decisions to scheduler decisions. For example, a good time to collect is probably when a task finishes, or a good time to create a heap in the hierarchy is likely when stealing a task.

We propose a specific memory manager and garbage collection algorithm based on the hierarchical heaps and disentanglement. Rather than going directly to a low-level description of the algorithms, we formalize a functional, ML-like language with nested parallelism and a memory manager by presenting a reasonably high-level operational semantics that accounts for all the key aspects of the memory management and collection, while abstracting away from the details. The semantics models parallelism using non-deterministic interleaving of parallel steps. It captures a parallel fork by “activating” a parallel tuple. This is meant to capture the lower-level idea of scheduling a parallel task (e.g. stealing the task in a work-stealing scheduler), and allows for lazy task creation (i.e. if
there is plenty of parallelism in the computation, a parallel fork need never be activated). The semantics captures the hierarchical nature of the heap and creates new heaps exactly when a parallel tuple is activated. The model interleaves garbage collection steps with steps of the mutator. A node in the hierarchy is “locked” and cannot evaluate when it is being collected, but other nodes are allowed to evaluate concurrently. Our semantics captures the object-by-object copying nature of our collector, but ignores details of how memory is laid out or allocated.

Based on the semantics, we prove that the disentanglement property holds on programs written in our language, and that the garbage collector is correct (memory safe and meaning-preserving).

To remain as general as possible, the semantics does not specify a scheduler, and abstracts over many important details a real implementation must consider such as concurrency, contention, and performance. We briefly describe the key algorithms and data structures for realizing the semantics (Section 4) based on a popular work-stealing scheduler [3, 7, 15].

We have implemented a prototype of our collector as part of the MLton compiler [39] by adding it to Spoonhower’s parallel scheduler [47]. The implementation follows our semantics and design but has to consider a variety of other details. We present some preliminary performance results on a handful of benchmarks. The collector achieves good performance compared to both the existing collector on MLton and the Manticore runtime system [8], although the numbers are meant as a proof of concept rather than a careful analysis.

The contributions of this paper include the following.

- A hierarchical model of memory for a nested-parallel functional language.
- The precise formulation of the disentanglement property in the context of our nested-parallel functional language, and the proof that this language guarantees disentanglement.
- The formulation of a hierarchical garbage collection technique that allows portions of memory to be garbage collected concurrently with other tasks.
- The design of a runtime system that realizes the hierarchical memory management techniques.
- A prototype implementation as an extension to the MLton compiler for Standard ML, and an evaluation of its performance.

2. Overview

We present a high-level, informal overview of the ideas proposed in this paper. The rest of the paper makes these ideas precise.

We use the quicksort algorithm as an example and assume an ML-like, strict, purely functional language with nested parallelism, where the primitive par creates a tuple by allowing its components to be evaluated in parallel. Figure 1 shows the code for quicksort in such a language. For simplicity, we assume that the input contains no duplicates. The function qsort uses the first item as a pivot to partition the input list into less-than and greater-than parts, written 11 and 1g, sorts them recursively in parallel, and concatenates the sorted lists to compute the final sorted output.

**Task Tree.** We can represent an evaluation of the quicksort algorithm with a *task tree* consisting of parallel tasks where each task corresponds to a parallel sub-computation as indicated by the par primitive. Figure 2 shows such a tree representation of the evaluation of the qsort function on the input list [5; 6; 3; 4; 1; 9]. The nodes of the task tree represent parallel tasks and the edges represent the control dependencies between them. For example, the root task, labeled T_A, which corresponds to the first call to the function, forks two new tasks T_B and T_C.

```plaintext
1 fun qsort l = 2 case l of 3      nil => nil 4 | h::nil => h 5 | h::t => let 6    T_B(qsort t) = partition h t 7      (sll, slg) = par (qsort l1, qsort lg) 8      in 9      append sll (h::slg) 10    end
```

![Figure 1. The code for quicksort.](image)

![Figure 2. An example run of quicksort. The nodes represent (parallel) tasks and the edges represent the control dependencies between them.](image)

**Memory as a Heap Tree and Disentanglement.** Classically, memory is viewed in programming languages as a flat, unstructured pool of objects, or a fixed hierarchy of several heaps, referred to as generations. In this paper, we propose an approach where we structure memory as a dynamic hierarchy by partitioning memory into heaps, each of which corresponds to a task in the task tree. To create this hierarchy, we dedicate a fresh heap to each forked task and allocate all memory requested by the task in its heap, which the task owns. When the task completes, we join its heap with that of its parent. As a result, we can organize the memory at any point during execution as a hierarchy, represented as a tree of heaps as follows. Let H_A and H_B be two heaps. If there is an edge between their owner tasks, T_A and T_B, respectively, then insert an edge between H_A and H_B. Otherwise, don’t insert any edge. We refer to the resulting tree on the heaps as a heap tree. The motivation behind our approach is to organize memory in a way that reflects the structure—specifically the dependencies—of the computation.

Figure 3 illustrates the task and the heap trees for our example. Each “qsort” node represents a task. The gray box around the node represents its heap. Tasks and the edges between them define the task tree. Heaps and the edges between them define the heap tree. The figure also shows the data stored in each heap. The root heap, H_A, contains the input lists L0 and the two lists obtained via partitioning, L1 and L2. The task T_B takes the input L1 from the root heap H_A and partitions it into two new lists L3 and L4. The task T_C takes the input L2 from the root heap H_A and produces the lists L5 and L6.

Having organized memory as a tree of heaps, we observe the following key disentanglement property. Let T_A and T_B be any two tasks in the task tree and let H_A and H_B be their heaps in the heap tree. If some object in H_A references (or points to) an object in H_B at any time during execution, then H_B is an ancestor of H_A in the heap tree. Intuitively speaking, this property holds because, in a purely functional language, a parallel task has access only to the objects that have been allocated before its execution or by the task itself, excluding the objects that are allocated by tasks that might have been executed in parallel (even if they are not actually executed in parallel). Such objects are allocated precisely by the tasks that are on the path from the task to the root of the task tree. Since all memory allocated by a task is placed in the heap of that task, and since there
is an isomorphism between the heap tree and the task tree as defined by the ownership relation, the disentanglement property holds. As a corollary, we can establish the following: if \( T_A \) is neither an ancestor nor a descendant of \( T_B \), then objects in \( H_A \) do not reference objects in \( H_B \) and objects in \( H_B \) do not reference objects in \( H_A \). In other words, if \( T_A \) and \( T_B \) are concurrent tasks that may be executed in parallel, then \( H_A \) and \( H_B \) are guaranteed to be “disentangled”. As suggested by disentanglement, in the quicksort example shown in Figure 3, all pointers in the memory point “up” in the heap tree: if an object in heap \( A \) points to another object in another heap \( B \), then \( A \) is an ancestor of \( B \).

In Sections 3.1 and 3.2, we make precise this high-level description of hierarchical heaps and disentanglement by presenting a semantics that restricts a task’s memory access to its own heap and ancestor heaps. We prove that a program evaluated with hierarchical heaps and restricted memory access has the same behavior as it would have under a standard operational semantics, thus establishing the disentanglement property.

**Garbage Collection.** When considered in the context of the heap tree, the disentanglement property leads us to observe that any leaf heap \( H \) in the heap tree can be collected by the task \( T \) that owns that heap independently of all the other heaps and tasks. This holds because, via disentanglement, \( H \) does not have any “roots” or pointers into it. Thus, garbage collection of \( H \) can proceed as all the other tasks are executing. A moving (copying) or a non-moving collection can be used. We can generalize this observation to any subtree in the heap tree as follows: the heaps in any subtree of the heap tree can be collected independently of all other heaps. As with a leaf of the heap tree, a subtree of heaps has no incoming pointers from outside the subtree, and thus the collection can proceed concurrently with all other tasks operating on the other heaps. This means that we can garbage collect a subtree of heaps by stopping only those tasks that own them as we allow other tasks to continue executing. Furthermore, the subtree can be garbage collected in parallel by taking advantage of disentanglement: all the leaves can be collected in parallel and, after their collections are completed, the parents of the leaves may be collected in parallel, and so on until the root is reached.

For example, in Figure 3, we can garbage collect heaps \( H_B, H_D, H_E, H_F, H_G \), and \( H_H \), which form a subtree rooted at \( H_B \), by stopping only the tasks \( T_B, T_D, T_E, T_F, T_G, T_H \), which own the heaps. To garbage collect the subtree, we can collect \( H_B, H_D, H_E, H_F, H_G \) in parallel, and then \( H_H \).

In Section 3.3, we make precise this informal description of garbage collection and establish its correctness by showing that collection does not free reachable locations (memory safety) or alter the behavior of the program (meaning preservation).

3. The Language

We present an ML-style functional calculus called \( \lambda^{HP} \) and an operational semantics that accounts for memory allocation and the scheduling of tasks, and enforces the property that tasks access memory only in their own heap and ancestor heaps. We then show that any \( \lambda^{HP} \) program is disentangled in that it may be run safely and that it is both deterministic and equivalent to a standard “flat” semantics where memory is treated conventionally. We then formalize a hierarchical memory manager on top of \( \lambda^{HP} \) and prove its correctness.

### 3.1 A Core Calculus for Disentanglement

**Abstract syntax.** Figure 4 shows the syntax of the language \( \lambda^{HP} \). The types include natural numbers (as the sole base type), functions and products. To present a precise accounting of memory operations, we distinguish between large values \( v \), which are always allocated in the heap, and small values consisting only of memory locations, written \( \ell \). Large values include natural numbers, named recursive functions and pairs of heap locations. Large values are not irreducible, but rather step to heap locations \( \ell \), which are the only

---

**Figure 3.** The heap and task tree for the quicksort example. Heaps are shown as gray boxes. As shown by the dashed arrows, pointers in heaps may point to higher, but not lower or cross, heaps.

**Figure 4.** Syntax of \( \lambda^{HP} \)

<table>
<thead>
<tr>
<th>Type</th>
<th>( T )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heap</td>
<td>( H )</td>
</tr>
<tr>
<td>Expression</td>
<td>( e )</td>
</tr>
<tr>
<td>Heap typing</td>
<td>( \Gamma \vdash H : \Sigma )</td>
</tr>
<tr>
<td>Path typing</td>
<td>( \Gamma \vdash H : \Sigma )</td>
</tr>
</tbody>
</table>

---

**Figure 5.** Statics of \( \lambda^{HP} \)

<table>
<thead>
<tr>
<th>Type</th>
<th>( T )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task</td>
<td>( T )</td>
</tr>
<tr>
<td>Expression</td>
<td>( e )</td>
</tr>
<tr>
<td>Heap typing</td>
<td>( \Gamma \vdash e : \tau )</td>
</tr>
<tr>
<td>Path typing</td>
<td>( \Gamma \vdash e : \tau )</td>
</tr>
</tbody>
</table>

---
Figure 6. Dynamics of $\lambda^H_P$

<table>
<thead>
<tr>
<th>Transition</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$H; e \rightarrow P; e'$</td>
<td>$H; e \rightarrow P; e'$; $H'$</td>
</tr>
<tr>
<td>$H; e \rightarrow P; e'$; $H'$</td>
<td>$H; e \rightarrow P; e'$; $H'$</td>
</tr>
<tr>
<td>$H; e \rightarrow P; e'$; $H'$</td>
<td>$H; e \rightarrow P; e'$; $H'$</td>
</tr>
</tbody>
</table>

Figure 7. Auxiliary dynamic judgments of $\lambda^H_P$

<table>
<thead>
<tr>
<th>Transition</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$H; e \rightarrow P; e'$</td>
<td>$H; e \rightarrow P; e'$</td>
</tr>
<tr>
<td>$H; e \rightarrow P; e'$</td>
<td>$H; e \rightarrow P; e'$</td>
</tr>
<tr>
<td>$H; e \rightarrow P; e'$</td>
<td>$H; e \rightarrow P; e'$</td>
</tr>
</tbody>
</table>

irreducible values of the language. Since small values consist only of locations, we use the term "value" to refer to a large value and "location" to refer to a small value.

In $\lambda^H_P$, we organize memory as a tree of heaps and restrict expressions to access only part of the memory (along a path going up the hierarchy). A heap, written $H$, (and variants), is simply a mapping from locations to (large) values. A heap path or simply a path, written as $P$, (and variants), is a list or stack of heaps. A path represents a leaf-to-root path in the heap tree; it is the collection of heaps to which an expression has access.

Formulas of $\lambda^H_P$ include locations and (large) values, introduction and elimination forms for the standard types such as function application, projection and pair creation, and parallel tuples $\langle e_1, e_2 \rangle$. Parallel tuples are evaluated sequentially by default, but may be activated for parallel evaluation. An active parallel tuple, $\langle T_1, T_2 \rangle$, consists of two parallel tasks, $T_1$ and $T_2$, which correspond to the components of the tuple.

A task, $H; e$, consists of an expression $e$ and a heap $H$. The heap $H$ serves as the portion of the memory that is private to the expression $e$, i.e., $H$ is visible only to the expression $e$ (including subexpressions of $e$). The reader may wonder why we introduce tasks as a level of indirection in the syntax since they can take only one form ($H; e$). When we formalize the memory manager, we will introduce another form of task, and the separation will make the presentation cleaner, so we adopt it now.

Statics. Figure 5 illustrates the static semantics of the calculus. Apart from the treatment of memory, the statics semantics of $\lambda^H_P$ is relatively standard. We first describe the typing of heaps and paths.

We ascribe type signatures to heaps and paths. A signature $\Sigma$ is a mapping from locations to types. The heap typing judgment

$\Sigma; v : \tau$; $H$ : $\Sigma$ ascribes the signature $\Sigma$ to heap $H$ under path signature $\Sigma'$, allowing the values in $H$ to refer to locations in $\Sigma'$. The empty heap $\emptyset$ is given the empty signature $\cdot$. The signature for a non-empty heap $H$ is defined inductively: each binding $\ell \rightarrow v$ extends the signature with $\ell : \tau$ if $v$ has type $\tau$ under the union of $\Sigma$ and the signature $\Sigma$ of $H$. By typing the contents of a heap under locations from both the path $\Sigma'$ and the heap $\Sigma$, we allocate a value in the heap to refer to locations in both the private and shared parts of the memory.

The signature of a path $P$, which is a list of disjoint heaps, is the union of the signatures of the individual heaps making up the path.

The expression typing judgment has the form $\Gamma_1 \vdash e : \tau$, indicating that $e$ has type $\tau$ under variable context $\Gamma$ and path signature $\Sigma$. Most of the typing rules seem standard: a location is well-typed if it is in the path signature and all the other expressions are typed in the usual way. There is an important point, however. An expression is typed only with respect to a path and not the whole memory, and the expression therefore may not have access to the whole memory. This point becomes evident when typing parallel tuples and tasks. The rule for active parallel tuples, $S$-ParA, requires that each constituent task be well-typed under the the same path signature. For each task $H; e$, the typing rule ensures that $e$ is well-typed under the union of the path signature, representing the shared memory, and the signature of $H$, representing the private memory.

Dynamics. The dynamics for $\lambda^H_P$, shown in Figure 6, use a small-step transition judgment $H; e \rightarrow P; e'$, which indicates that under heap $H$ and path $P$, $e$ steps to $e'$ and produces new heap $H'$. In the judgment, $H$ is the hierarchical heap that is private to this computation and will be used for lookup as well as allocation. The path $P$ is the shared memory for $e$, consisting of the hierarchical heaps on the path from $H$ to the root (excluding $H$).

The step relation uses the heaps on the path for lookups but not for allocation. In other words, the heaps on the path are read-only and are never modified; for this reason, we think of evaluation taking place in the context of a path, which we write as a subscript of the evaluation judgment. The impact of the distinction between heaps and paths can be seen in the rules for lookup and allocation.

The auxiliary lookup judgment $P; e \rightarrow P; e'$, defined in Figure 7, indicates that looking up the location $\ell$ in path $P$ results in $v$. The rules are defined inductively on the path $P$: the rule looks first in the heap at the head of the path (the private heap) and, if $\ell$ is not found, the rest of the path is scanned recursively. The allocation
We state and prove one of this paper’s key contributions: that this models the possibility that another processor may begin evaluating expressions to such a memory location in a sibling or descendant heap. Accesses do not cause a program to get stuck (such as by accessing a memory location that is internal to the task. Only the internal private heap may be accessed by ancestors. We have formulated the semantics of purely functional programs written in λ by-value calculus with left-to-right evaluation. Function application and projection evaluate the subexpressions down to heap locations, look up the locations in the path, and perform the operation. For technical reasons, the function application rule requires that the location ℓ be bound, but doesn’t immediately use the value.

Sequential and inactive parallel tuples evaluate the first component to a location, followed by the second. If both components of a parallel tuple are evaluated fully before the tuple is activated, it changes into a pair (ℓ₁, ℓ₂) (rule D-ParE). A parallel tuple may also be activated at any time, non-deterministically, by rule D-Activate. This models the possibility that another processor may begin evaluating one of the components in parallel. The activation rule changes the tuple into an active parallel tuple with two new tasks (each with a new, initially empty, private heap). Once a parallel tuple is activated, either component task may evaluate at any time using rules D-ParAS1 and D-ParAS2. (The rule D-ParAS2 does not require e₁ to be fully reduced to a location.)

The transition judgment for tasks, T ⇒ P T' indicates that, under path P, task T steps to T'. It doesn’t include a private heap since that is internal to the task. Only the internal private heap may be modified, so P is read-only. Rule D-TaskStep (Figure 7) allows e to step using the expression step judgment.

When a component of a parallel tuple is stepped (the premises of D-ParAS1 and D-ParAS2), the heap for the tuple, H, is consoed onto the current path and becomes part of the shared memory. This means that a heap becomes read-only when it is not at a leaf of the heap tree. When both components of an active parallel tuple are evaluated down to locations, the tuple steps to an ordinary pair using rule D-ParAE and merges the two local heaps into the parent heap.

3.2 Disentanglement

We state and prove one of this paper’s key contributions: that purely functional programs written in λHF have the disentanglement property. Recall that we have defined disentanglement informally as the property that parallel tasks can only access data allocated by ancestors. We have formulated the semantics of λHF based on this property by restricting a task to access only the locations that are allocated in its own and its ancestors’ heaps. To prove that an arbitrary λHF program has the disentanglement property, we first show that a well-typed program never gets stuck. This familiar type safety property guarantees that the restrictions placed on the memory accesses do not cause a program to get stuck (such as by accessing a memory location in a sibling or descendant heap).

We then show that a well-typed program yields the same result and exhibits the same termination behavior as it does under more traditional semantics. In other words, we show that, for a well-typed program, a run using hierarchical heaps and a run using conventional semantics either 1) produce the same result or 2) both do not terminate. This result also suffices to show that λHF programs are deterministic. We prove it by giving a standard operational semantics for terms which syntactically resemble λHF without heap locations or active parallel tuples. We define a transformation, called flattening, from arbitrary λHF expressions to such flat expressions, and show that the operational semantics of Figure 6 coincide with the storeless semantics up to flattening.

Type Safety. We first establish that well-typed terms do not become stuck by establishing progress and preservation. The proofs are straightforward, but two lemmas are required. The first states that if a path P is well-typed with a context that contains ℓ, then P contains a binding for ℓ with a value of the correct type.

Lemma 1. If P : Σ, ℓ : τ, then there exists a value v such that $\ell \xrightarrow{\text{loc}} v$ and $\Gamma, v : \tau$.

Proof. By construction.

The second lemma is an important property of heaps, which states that a merged heap is well-typed.

Lemma 2. If H : Σ and H' : Σ', then (H ∪ H') : Σ ∪ Σ'.

Proof. By induction on the derivation of H'. If H' = 0, then the result is trivial. Otherwise, H' = H[S₁ → v] and Σ' = Σ'', ℓ : τ. By induction, (H ∪ H'') : Σ ∪ Σ''. By the heap typing rules and weakening, H ∪ H' = (H ∪ H'')[ℓ → v] : Σ ∪ Σ'', ℓ : τ = Σ ∪ Σ'.

We now state and prove the preservation and progress lemmas.

Lemma 3 (Preservation). If $\Gamma, e : \tau$ and H : Σ₁ and P : Σ₂ and H, e → P e'; H', then H' : Σ₁ where Σ₁ is an extension of Σ₁, and $\Gamma, e' : \tau$.

Proof. By induction on the derivation of H; e → P e'; H'.

Lemma 4 (Progress). If $\Gamma, e : \tau$ and H : Σ₁ and P : Σ₂, then either e is a location or there exist e' and H' such that H; e → P e'; H'.

Proof. By induction on the derivation of $\Gamma, e : \tau$.

Correspondence with flattened semantics. Figure 8 defines the flattening transformation, which converts a λHF expression into a flat expression with no nested tasks and no free heap locations (but which is still a valid expression in the λHF syntax). The judgment $|e|_P \xrightarrow{\ell} \hat{v}$ indicates that expression e flattens to $\hat{v}$ under path P. The transformation is defined inductively. There are two important rules to note. Locations are looked up in the path and the resulting value is recursively flattened (this will terminate since heaps and heap paths have no cycles). The flattened value is substituted for the location. The rule for active parallel tuples recursively flattens the two subexpressions, adding the local heaps to the path, and then converts the active tuple into an inactive tuple, thus flattening the hierarchy.

We make three observations about the flattening transformation:

- If e contains no active parallel tuples or heap locations (it may be a source program that has not yet begun executing), then $|e|_P \xrightarrow{\ell} e$.
- If $\ell \in \text{dom}(P)$ and $|e|_P \xrightarrow{\ell} \hat{v}$, then $\hat{v}$ is simply the value at $\ell$ “lifted” so that it is closed with respect to the heap.
- Flattening is deterministic: if $|e|_P \xrightarrow{\ell_1}$ and $|e|_P \xrightarrow{\ell_2}$, then $\ell_1 = \ell_2$.

Flattened expressions may be run using the simple operational semantics of Figure 9. No path or heap is required, since these expressions have no heap locations. There is now no notion of “active” parallel tuples, and instead (formerly “inactive”) parallel tuples may evaluate in parallel using rules F-ParS1 and F-ParS2. While these parallel evaluation rules allow for nondeterministic interleavings, evaluation is nevertheless deterministic. As usual, we show this using an intermediate result, the diamond lemma, which states that if an expression steps to two different expressions in one step, they can be “brought back together” in one step. We do not immediately prove confluence, the generalization of the diamond property to multi-step evaluation. This will be part of the disentanglement theorem.

Lemma 5 (Diamond Lemma). If e → e₁ and e → e₂, then there exists $e'$ such that $e₁ → e'$ and $e₂ → e'$.
Lemma 9. Suppose that \( \hat{e} \to \hat{e}' \) and \( \|e\|_{H'} \to e' \). Then there exist \( H' \) and \( e \) such that \( H'; e \to_p e'; H' \) and \( \|e\|_{H';p} \to e' \).

Proof. By induction on the derivation of \( \hat{e} \to \hat{e}' \).

Disentanglement. Finally, we state and prove the disentanglement theorem, which formalizes the two properties (type safety and correspondence with flattened semantics) outlined at the beginning of this section. If a source program \( e \) is well-typed and it evaluates to \( e' \), then (1) \( e' \) is not stuck and (2) for any flattened evaluation of \( e \) to \( e'' \) (\( e' \) and \( e'' \) need not be related), \( e' \) and \( e'' \) may be brought back together in the style of confluence.

Theorem 1 (Disentanglement). Suppose \( e \) is a source program with no active parallel tuples. If \( \cdot\vdash e; \tau \) and \( \Theta; e \to^* e'; H' \), then

1. Either \( e' \) is a location or there exists \( e'' \) such that
   \[ H'; e' \to^* e'' ; H'' \]

   \[ e'' \to^* e'' ; e''' \]

   and \( \|e''''\|_{H''';p} \to \hat{e}'''' \).

Proof. 1. We show by induction on the derivation of \( \Theta; e \to^* e'; H' \) that \( \vdash H'; \Sigma ; \tau \) and \( \vdash e'''' ; \tau \). From this, the result follows from Lemma 4. If \( e'''' = e \) and \( H'''' = \emptyset \), then \( \vdash H'''' ; \vdash e'''' ; \tau \) and the typing result on \( e'''' \) was assumed. Otherwise, \( \vdash \Theta; e \to^* e'''' ; H_0 \) and \( H_0; e'''' ; \vdash_1 e'''' ; H'''' \). By induction, there exists \( \Sigma_0 \) such that \( \vdash \Theta; \Sigma_0 \vdash_1 e'''' ; H'''' \). By the typing result for \( e'''' \) follows from Lemma 3.

2. We have \( \|e''''\|_{H''''} \to e'''' \). By inductive application of Lemma 8, we have \( e'''' \to^* e''' \) and \( \|e''''\|_{H''''} \to e''' \). By a standard inductive application of Lemma 5, there exists \( e''' \) such that \( e'''' \to^* e''' \) and \( e''' \to^* e'''' \). By an inductive application of Lemma 9 to the latter derivation, there exist \( e'''' \) and \( H'''' \) such that \( H'; e'''' \to^* e''''' ; H'''' \) and \( \|e''''''\|_{H''''} \to \hat{e}'''''' \).

Part (2) of Theorem 1, as well as its proof, is illustrated graphically in Figure 11. This statement is surprisingly powerful. In particular, it immediately implies that if \( e \) terminates with flattened semantics, it terminates with hierarchical semantics, and vice versa, that both semantics are deterministic and that they both produce the same result up to flattening. Each of these statements is simply an instantiation of Theorem 1.

Corollary 1. If \( e \to^* \hat{v} \), then there exist \( v \) and \( \hat{H} \) such that \( \Theta; e \to^*_v e; H \) and \( \|e\|_{\hat{H}} \to \hat{v} \).

3.3 Hierarchical Garbage Collection

We describe a hierarchical garbage collector, dubbed HGC, that takes advantage of disentanglement to perform concurrent garbage collection at the granularity of tasks. The collector HGC can collect any heap in the heap hierarchy by stopping the task that owns the heap along with its descendant tasks, while all other (non-descendant) tasks continue executing. Any number of independent heaps, which are not descendants and ancestors of each other, can be collected in parallel. The collector HGC is thus able to perform garbage collection without incurring additional synchronization.
The collector we formalize is a semispace copying collector, though component, \( F \) second component, to-heap \( H \) collection. The tuple represents a heap which is in an intermediate shown at the top of Figure 12. The new form, Syntax.

While they analyze only stop-the-world dynamic rules to perform garbage collection on tasks. Several tasks to represent a task which is locked for collection, and several garbage collectors. For notational convenience, we implicitly extend the domain of forwarding maps by mapping locations that are not in the domain to themselves:

\[ \ell \mapsto \{ e \} \]

Forwarding maps have been replaced with every free location \( \ell \) has been replaced with \( \ell \). The forwarding map models forwarding pointers, which are commonly used to perform this function in practical copying garbage collectors. For notational convenience, we implicitly extend the domain of forwarding maps by mapping locations that are not in the domain to themselves:

\[ F(\ell) := \begin{cases} F(\ell) & \ell \in \text{dom}(F) \\ \ell & \ell \notin \text{dom}(F). \end{cases} \]

We will also use the notation \( F(e) \) to denote an expression in which every free location \( \ell \) has been replaced with \( F(\ell) \). We extend \( F \) to operate on heaps in the following way:

\[ F(H) := \{ F(\ell) \mapsto F(v) \mid \ell \mapsto v \in H \} \]

and on signatures in the following way:

\[ F(\Sigma) := \{ F(\ell) \mapsto F(v) \mid \ell \mapsto v \in \Sigma \} \]

A particularly common usage of the forwarding map is the notation \( [\ell \mapsto \ell'](e) \) (and similar for heaps and signatures), in which \( e \) is forwarded with the singleton forwarding map forwarding \( \ell \) to \( \ell' \). In addition, we use \( F[\ell \mapsto \ell'] \) to indicate the extension of the map \( F \) with a mapping from \( \ell \) to \( \ell' \).
Statics. The typing rule for the new form of task is given in Figure 12. The rule uses the rules for typing heaps to assign a signature to the concatenation of the from- and to-heaps. Since values in $H_1$ might refer to locations that have already been copied, we forward the values of $H_1$ before concatenating with $H_2$. The expression must be well-typed under the concatenation of the global signature $\Sigma$ and this new signature $\Sigma$. In addition, the typing rule also enforces a key invariant of garbage collection: that the scan set $S$ is equal to the set $FL(H_1; e)$ of locations that appear free in either $e$ or a binding of $H_1$ and are not bound in $H_2$. Formally,

$$FL(H_1; e) \equiv FL(e) \cup \bigcup_{(\ell \in \text{dom}(H_2))} FL(H_1(\ell)) \setminus \text{dom}(H_2)$$

Dynamics. Note that if a collection is in progress and the task is of the form $H \leftarrow T$, no rule allows $e$ to step, so no changes are required to the semantics to ensure that $e$ is in fact locked during collection. Instead, we introduce a new transition rule for tasks, D-GCStep, which allows the task to proceed with collection. The auxiliary judgment $T \leadsto e T'$ indicates that a step of garbage collection transforms $T$ into $T'$.

Our garbage collection algorithm uses a small-step semantics, where one step (atomically) copies a single location. A full garbage collection will therefore involve many individual steps, which can be freely interleaved with evaluation or collection on other processors. Figure 12 gives the transition rules for collection. Rule HGC-StartGC locks the task for collection and sets up the tuple by using the existing heap, $H$, as the from-heap, initialising the to-heap and the forwarding map to be empty and the scan set to be $FL(e)$, the roots of the expression. Rule HGC-Copy performs one step of collection. The rule takes a location from $S$ which is present in $H_f$, copies its value $v$, and adds to $S$ the locations of $v$ which have not already been copied. Rule HGC-EnqGC requires that $\text{dom}(H_f) \cap S = \emptyset$, i.e. that there are no remaining locations to be collected. If this is the case, $e$ is unlocked and the to-heap becomes the new heap.

Rule HGC-StartGC can apply to a task nondeterministically at any time, and so our semantics does not specify when collections can or should occur. Also note that a collection in the semantics can collect any heap in the hierarchy at any time. An entire subtree becomes the new heap. Thus, we effectively model parallel collection.

3.3.1 Safety and Correctness Proofs

We will now proceed to prove that our collection algorithm is memory safe, that is, it never frees a location that will be dereferenced later. Because our calculus is typed and dereferencing a freed location is a stuck state, memory safety is an immediate corollary of type safety. Thus, we only need to update the progress and preservation proofs to handle the new rules. Several lemmas are required. One important lemma is that garbage collection preserves typing, in that if $e$ is well-typed under heap $H \top H'$ and $H$ is discarded by garbage collection, $e$ is also well-typed under heap $H'$. The signature of the higher levels of the hierarchy, notated $\Sigma_1$ in the lemma, does not change. Both $H$ and $H'$ are typed under $\Sigma_1$. We also allow the expression to refer to additional locations in another signature, $\Sigma_2$, This contains locations at levels lower than the level being collected. In using this lemma to prove preservation, we will only be interested in the case where $\Sigma_2$ is empty because we want to show type preservation of the expression whose heap is being collected. However, $\Sigma_2$ is necessary for the proof of the lemma, when inducting on expressions containing nested heaps.

Lemma 10. If $\Gamma \vdash_{\Sigma_1} e : \tau$ and $\Gamma \vdash_{\Sigma_2} e : \tau$ and $FL(H'; e) \cap \text{dom}(H) = \emptyset$, then $\Gamma \vdash_{\Sigma_1, \Sigma_2} e : \tau$.

Proof. By induction on the derivation of $\Gamma \vdash_{\Sigma_1, \Sigma_2} e : \tau$. □

Lemma 11 states that typing of an expression is preserved by performing renamings on both the expression and signature.

Lemma 11. If $\Gamma \vdash e : \tau$, then $\Gamma \vdash_{\Sigma_1, \Sigma_2} e : \tau$.

Proof. By induction on the derivation of $\Gamma \vdash e : \tau$. □

We now restate the preservation and progress theorems and prove them for the full language, including garbage collection.

Theorem 2 (Preservation). If $\Gamma \vdash_{\Sigma_1, \Sigma_2} e : \tau$ and $e : \tau$, then $\Gamma \vdash_{\Sigma_1, \Sigma_2} e : \tau$.

Proof. We consider the cases for garbage collection $T \leadsto e T'$ where, without loss of generality, $e = \bullet T_1, T_2 \triangleright$ and $T_1 = T_2$.

- HGC-StartGC. Then $T = H \cdot e_1$. By typing inversion, $H : \Sigma$ and $\Gamma \vdash_{\Sigma_1, \Sigma_2} e_1 : \tau_1$. We have that $H(\ell) \equiv H = \Sigma$ so by task typing, $\Gamma \vdash_{\Sigma_1, \Sigma_2} (H; FL(e_1); \emptyset; \emptyset) \cdot e_1 : \tau_1$. By definition, $FL(e_1) = FL(\emptyset; e_1)$.

- HGC-Copy. Then $T = (H_1[\ell \mapsto v]; S \cup \{\ell\}; H_2; F \cdot e_1).$ By typing inversion, $F(H_1[\ell \mapsto v]) \equiv H : \Sigma$ and $\Gamma \vdash_{\Sigma_1, \Sigma_2} e_1 : \tau_1$. We have $F(H_1[\ell \mapsto v]) \equiv H' \equiv \ell' \cdot (F(H_1[\ell \mapsto v]) \equiv \ell' \cdot (F(H_1[\ell \mapsto v]) = \ell' \cdot (H_2[\ell \mapsto v]) \equiv H_2') \equiv \ell' \cdot (H_2[\ell \mapsto v]) \equiv H_2$). Since, and since $\ell' \equiv \ell' \cdot (\text{dom}(H_2) = \emptyset$). Lemma 11 gives $\Gamma \vdash_{\Sigma_1, \Sigma_2} (H_2; F \cdot e_1) : \tau_1$. It is a straightforward exercise to show that $S = FL(H_1[\ell \mapsto v]) = (H_1[\ell \mapsto v])$.

- HGC-EnqGC. Then $T = (H_1; S; H_2; F \cdot e_1)$. By typing inversion, $F(H_1) \equiv H : \Sigma$ and $\Gamma \vdash_{\Sigma_1, \Sigma_2} e_1 : \tau_1$ and $S = FL(H_1(\cdot) e_1)$, so $\text{dom}(H_1(\cdot) e_1) \cap S = \emptyset$. If $H_1, S$, by Lemma 10, $\Gamma \vdash_{\Sigma_1, \Sigma_2} e_1 : \tau_1$. By the typing rules, we have $\Gamma \vdash_{\Sigma_1, \Sigma_2} H_1(\cdot) e_1 : \tau_1$.

□

Theorem 3 (Progress). If $\Gamma \vdash_{\Sigma_1, \Sigma_2} e : \tau$ and $\Gamma \vdash_{\Sigma_1} e : \tau$, then $\Gamma \vdash_{\Sigma_1, \Sigma_2} e : \tau$ and $\Gamma \vdash_{\Sigma_1, \Sigma_2} e : \tau$.

Proof. We consider the case of $e = \bullet T_1, T_2 \triangleright$ where either $T_1 = (H_1; S; H_2; F \cdot e_1)$ or $T_2 = (H_2; S'; H_2'; F' \cdot e_2)$ or both. Without loss of generality, suppose $T_1 = (H_1; S; H_2; F \cdot e_1)$. If $\text{dom}(H_1(\cdot) S) = \emptyset$, then apply HGC-EnqGC. If $\ell \in \text{dom}(H_1(\cdot) S)$, then apply HGC-Copy. In either case, $T_1$ steps with D-GCStep. □

Together, the progress and preservation theorems imply the standard type safety property: a well-typed term will not become stuck. Formally, if $e \leadsto e' \leadsto \ldots$, then, for some $e$ and $e'$, we have $\emptyset; e \leadsto e' \leadsto \emptyset; H$. In particular, evaluation of $e$ will never dereference a freed memory location. This completes the proof of memory safety. This is only half of the correctness of the garbage collector; we have not yet shown that collection is meaning-preserving, i.e. that the behavior of a garbage-collected program is identical to an execution without collection. This property is straightforward to prove using the same flattening machinery that was introduced in the previous section. The proof simply requires extending the definitions and proofs related to flattening to account for garbage collection. We
first extend the definition of flattening to cover tasks which are in the process of collection:
\[
\|e_1\|_{H_1; H_2; P} \sim \hat{e}_1 \quad \|e_2\|_{F(H'_2; H'_1; P)} \sim \hat{e}_2
\]

The two other rules for parallel tuples in which one task is of the form \(H \cdot e\) and the other is of the form \(H'_1; S; H'_2; F\) \([e]\) are omitted but are defined in the natural way. We also require two more lemmas regarding flattening, which will be used in the extension of the simulation proof.

**Lemma 12.** If \(\text{dom}(H) \cap \text{dom}(P) = \emptyset\) and \(\text{dom}(F) \subset \text{dom}(H)\) and \(\|e\|_{H; P} \sim \hat{e}\) then \(\|F(e)\|_{H_1; H_2; P} \sim \hat{e}\).

**Proof.** By induction on the derivation of \(\|e\|_{H; P} \sim \hat{e}\). The interesting case is the case for a location \(\ell\). □

**Lemma 13.** If \(\text{dom}(H_1) \cap \text{FL}(e) = \emptyset\) and \(\|e\|_{H_1; H_2; P} \sim \hat{e}\) then \(\|e\|_{H_2; P} \sim \hat{e}\).

**Proof.** By induction on the derivation of \(\|e\|_{H_1; H_2; P} \sim \hat{e}\). The interesting case is the case for a location \(\ell\). □

Lemma 14 is simply a restatement of Lemma 8, which showed that a step of the hierarchical semantics can be simulated by zero or one steps of the flattened semantics. The lemma now includes garbage collection steps.

**Lemma 14.** Suppose that \(H; e \rightarrow_{\text{HC}} e'; H'\) and \(H :: P : \Sigma\) and \(F \vdash e : \tau \|e\|_{H; P} \sim \hat{e}\). Then either \(\|e'\|_{H; P} \sim \hat{e}\) or \(\|e'\|_{H; P} \sim \hat{e}\) and \(\hat{e} \rightarrow_{\text{HC}} e'.\)

**Proof.** The new cases are those that instantiate D-ParAS1 or D-ParAS2 with one of the GC rules. We show that if \(T \rightarrow_{\text{HC}} T'\), then for any \(T_2\), if \(\|T, T_2\|_{H; P} \sim \hat{e}\) then \(\|T', T_2\|_{H; P} \sim \hat{e}\).

- **HGC-Copy.** Then \(T = \langle H_1[t \mapsto v]; S \parallel (t; H; F) :: [e]\). Let \(H_T = F(H_1[t \mapsto v]) \cup H_1\)
  \(H_T = F(H'_1) \cup [t \mapsto H_1[t \mapsto F(v)]\)
We have \(\|e\|_{H_T; P} \sim \hat{e}_1\) and wish to show \(\|e_1\|_{H_1; H_2; P} \sim \hat{e}_1\).
  This follows from Lemma 12 since \(\|\ell \mapsto\{\|L_1\(H)\) = [t \mapsto e'(H_1[t \mapsto v]) \cup H_1\]
  \(\|\ell \mapsto e'(H'_1) \cup H_1[t \mapsto F(v)]\)
  \(\|e(T) \cup [t \mapsto H_1[t \mapsto F(v)]\)
  \(= H_T\)
- **HGC-StarGC.** Then \(T = H_1 \cdot e_1\) and \(\|e_1\|_{H_1; H_2; P} \sim \hat{e}_1\). If \(F = \emptyset\), then \(\|F(H_1)\|_{H_1} = H_1\), so \(\|H_1[H; \text{FL}(e_1); F; t; e_2] \|_{H; P} \sim \hat{e}\)
- **HGC-ExtGC.** Then \(T = \langle H_1; S; H; F; e \cdot e_1\) and \(\|e_1\|_{H_1; H_2; P} \sim \hat{e}_1\)
By typing, \(S = \text{FL}(H_1; e_1)\), so \(\text{dom}(H_1) \cap \text{FL}(e_1) = \emptyset\) and Lemma 13 gives \(\|e_1\|_{H_1; H_2; P} \sim \hat{e}_1\) and \(\|H_1[H; e_1; e_2] \|_{H; P} \sim \hat{e}\).

Finally, Theorem 4 states the correctness of HGC: the hierarchical semantics produces a value from an expression if and only if the flattened semantics produces a value from the same expression, and the two values agree up to flattening. It was earlier shown that the flattened semantics is consistent with the non-collected hierarchical semantics, so this is sufficient to show that garbage collection does not change the meaning of a hierarchical program.

**Theorem 4.** Let \(e\) and \(\tau\) be such that \(\vdash e : \tau\).

- If \(e \rightarrow \hat{e}'.\) then there exist \(\ell\) and \(H\) such that \(\ell; e \rightarrow_{\text{HC}} \ell; H\) and \(\|\ell; e\|_{H; P} \sim \hat{e}'\)
- If \(\ell; e \rightarrow_{\text{HC}} \ell; H\), then there exists \(\hat{v}\) such that \(e \rightarrow \hat{v}\) and \(\|\ell; e\|_{H; P} \sim \hat{v}\).

**Proof.** These are simply items 1 and 2 of Corollary 1, and are instances of Theorem 1, which is easily proven for the full language including garbage collection by using Lemma 14 instead of Lemma 8 in the proof. □

4. **The Design**

There are three primary challenges to the design of a practical implementation of the hierarchical semantics. These challenges primarily stem from the fact that a practical design must handle concurrency to realize parallelism.

- **Scheduling:** the semantics models scheduling using a non-deterministic interleaving of threads; a practical design must use a specific scheduler.
- **Algorithms for the heap hierarchy:** a practical design must use efficient and correct data structures and algorithms to represent the heap hierarchy.
- **Hierarchical garbage collection:** a practical design must ensure correct and performant garbage collection.

4.1 **Scheduling**

For this design, we use a work-stealing scheduler as described by Blumofe et al. [15]. This scheduler assigns every processor a double-ended queue, or “deque”, containing tasks associated with that processor. As a computation spawns new tasks, they are pushed onto the bottom end of the processor’s deque. Once a task is completed, the processor fetches the next task by popping the task from the bottom end of the deque. If the processor finds the deque to be empty, the processor steals a task from a randomly chosen victim processor by popping from the top of the victim’s deque.

The semantics presented in Section 3.1 accounts for some of the scheduler actions, such as the creation of parallel tasks, by distinguishing between active and inactive parallel tuples. There is a subtle but a crucial difference, however, between the management of tasks in the semantics and in work stealing. The semantics allows a parallel pair to be activated at any time by creating a task for each component and by pairing each task with its own heap. In work stealing, however, task creation is a lazy and asynchronous operation: a task is created only when one of the components of a parallel tuple is stolen, as the other component continues to execute serially. This subtle difference creates a challenge for identifying the heap for a task.

For example, consider a processor \(Q\) executing a parallel tuple \(e_1, e_2\). To execute the tuple, the processor starts executing the left component \(e_1\). Sometime after \(e_1\) is reduced to \(e'_1\), \(e_2\) may be stolen and converted to a task \(T_2\) so that it can be executed by the thief processor. Since the algorithm has not converted \(e_2\) to a task before \(e_1\) was stolen, we don’t have a heap for it, and thus don’t know the objects allocated during the execution of \(e_1\) thus far.

To solve this problem, we assign to each parallel pair of the form \(e_1, e_2\) a level that corresponds to its nesting depth. For example, if the expression \(e = e_1, e_2\) has level \(m\), then \(e_1\) and \(e_2\) have level \(m + 1\). We then use levels to identify the heap of a task by tracking the levels at which steals take place.

4.2 **Hierarchical Heaps**

In our design, the abstract heaps of the semantics are implemented using *pages* and *chunks*, described below. These smaller units of memory are collected in data structures which we call *superheaps,*
The computation starts with a root superheap. During the execution, a processor works on one superheap. When a steal happens, we remember the level of the stolen task at the superheap. The allocation operation is simply a bump allocation within the current chunk. When the chunk is full, a new one is fetched from the chunk pool and assigned a level. Looking up an object is simply a pointer dereference.

### Heap operations
To realize the semantics, the implementation of heaps needs to support four key operations: creating heaps, merging two heaps, allocating a new location in a heap, and looking up the value of a location in a heap. The design allows creating a heap by marking the level for the stolen task in the victim processor as activated and creating a new superheap for the stolen task. Merging two heaps corresponds to simply unioning their chunks. The allocation operation is simply a bump allocation within the current chunk. When the chunk is full, a new one is fetched from the chunk pool and assigned a level. Looking up an object is simply a pointer dereference.

### 4.3 Garbage collection
The semantics allow for collection of any heap in the memory hierarchy. The heaps that are leaves of the heap hierarchy, however, are special in the sense that they can be collected without communicating with any other processor, since they are disentangled. In our design, we therefore distinguish between two forms of collection. Local collection performs garbage collection on a leaf in the heap hierarchy; non-local collection collects all of the heaps in a subtree of the hierarchy.

#### Local Collection.
A processor starts a local collection by locking its deque, ensuring that no tasks can be stolen during collection. A steal of a task from the deque corresponds in the semantics to the activation of a parallel tuple contained in the current expression, so locking the expression in rule HGC-StartGC corresponds to locking the deque.

After locking its deque, the processor proceeds to run a standard algorithm such as Cheney (copying) collection [20] adapted for hierarchical heaps. Such a collection would scan the root set (the collecting processor's stack) for all pointers within the scope of collection. A pointer is considered to be within scope if it points to an object residing in one of the levels being collected. All roots in scope of the collection are copied to a new collection of chunks. The in-scope check also appears in the semantics, since rule HGC-Copy only copies locations that appear in both the scan set and the heap being collected.

After all roots are copied, copy collection continues normally, level-by-level in descending order. Any pointers to objects in scope of the collection but in lower levels are copied and scanned once collection gets to that level. Note that by the disentanglement property, no pointers from lower levels to higher levels can exist, so once a level is completely scanned, it does not need to be revisited. Once copy collection completes, all chunks in the levels being collected are released to the chunk pool, the processor's deque is unlocked and execution resumes (corresponding to transition rule HGC-EndGC).

#### Non-Local Collection.
As specified in Section 3.3, the GC semantics allows non-deterministic collection of any heap in the hierarchy. Here, we describe a specific non-local GC algorithm that collects all heaps in a given subtree at each non-local collection. Other algorithms are also possible.

A non-local collection starts by first choosing a subtree of the heap hierarchy to collect and identifying the processors participating in the collection. All participating processors then synchronize; each locks its deque and starts collection. All other, non-participating, processors continue to work on their execution as their accessible data will not be affected by the collection.

Participating processors then, in parallel, scan their root sets for all pointers within the scope of collection, just like for local collection. These pointers are all copied to the corresponding target space. After all the roots are copied, collection proceeds level-by-level in parallel across all superheaps involved in the collection. However, each superheap is collected sequentially on its own.
Therefore, once collection on all of a parent superheap’s children is completed, one processor will continue collection on the parent superheap while the other processors remain idle. In this way, collection proceeds up the hierarchy of heaps until complete, at which point all the processors involved in the collection unlock their dequeues and resume execution.

Open issues. Our design leaves open several important questions: 1) when to garbage collect, 2) whether to perform a local or non-local collection and 3) if non-local, which heaps, or subtree, to collect. The answers to these questions are not as straightforward as in traditional collection algorithms, where relatively simple amortization arguments can guide them. We leave the investigation of these questions to future work. In the implementation described below, we choose a simple technique adopted from traditional garbage collection literature.

5. Implementation and Evaluation

We describe a prototype implementation and preliminary evaluation of the proposed hierarchical memory manager.

5.1 Implementation

We implement our hierarchical memory manager on top of Daniel Spoonhower’s parallel extension of the MLton compiler [47]. As a whole-program optimizing, high-performance compiler for Standard ML, MLton [39] offers an excellent starting point for parallel computing. Spoonhower extended MLton to support a number of parallelism primitives including nested (fork-join) parallelism and futures. Spoonhower implemented many different schedulers; we use his implementation of a standard work-stealing scheduler.

Our implementation, which we call mlton-parmem, closely follows the design described in Section 4 but of course has to consider many more details. One such detail is the treatment of stacks. MLton executes all code with a traditional call stack, which is allocated on the heap. We allocate computation threads’ initial stacks in their associated heap. When the stack runs out of space, a new stack is allocated in the same heap at the current level of the computation. Upon first inspection, this model seems to blantly violate disentanglement as a stack allocated in a lower level may have pointers in a higher level by virtue of computation at that level. However, since the stack cannot be pointed to from any object, there is no issue of having parallel reads from another processor. In addition, since the stack is explicitly scanned as a root set for collection, we keep track of all the entangled pointers at collection time.

Currently, our implementation only supports local collection, which collects unactivated levels in descending order until the first activated level. For example, in Figure 13, local collection will only collect level 8 as all lower levels (drawn higher in the figure) have been activated. Our algorithm for deciding when to collect is as in traditional collection algorithms, where relatively simple collection algorithms can guide them. We leave the investigation of these questions to future work. In the implementation described below, we choose a simple technique adopted from traditional garbage collection literature.

5.3 Measurements

For our experiments, we use a 64-core AMD machine with 128 gigabytes of memory, but due to technical limitations of the underlying MLton codebase, we are not able to utilize more than 32 cores. We use interleaved NUMA allocation in all of our experiments. For the grain sizes we use, our benchmarks create anywhere from several thousand to hundreds of thousands of tasks.

In addition to our mlton-parmem compiler, we run our benchmarks on the manticore compiler, the mlton compiler for Standard ML and Spoonhower’s parallel extension of MLton, which we call mlton-spoonhower. The memory manager of mlton-spoonhower allows for parallel allocation, but performs stop-the-world sequential collection. We choose this compiler as a naive baseline for comparing parallel garbage collectors.

The Manticore compiler for Parallel ML [28] offers support for an ML-like language extended with several paradigms for parallelism, including fork-join parallelism and speculation. Manticore uses an advanced NUMA-aware memory manager based on the Doligez-Leroy-Gonthier and Appel semi-generational collectors. It is a reasonably well-developed system with particular innovations on memory management and has been shown to deliver good performance [8, 11, 13, 28, 44]. Since it is based on ML, Manticore is a good basis for comparison for the approach suggested in this paper.

We use the mlton compiler timings as the sequential baseline for all our speedup calculations. We derive the sequential versions of our
5.4 Performance and Scalability

Figure 14 shows some of the key measurements across our benchmarks using the following quantities.

- Serial run time, $T_s$, is the time in seconds for a serial run with mlton.
- Uniprocessor run time, $T_1$, is the uniprocessor run-time of the parallel code reported in seconds;
- The overhead $O$ is the overhead calculated as $T_1/T_s$.
- 32-processor run time, $T_{32}$, is the 32-processor run-time of the parallel code reported in seconds;
- The speedup $S$ is the 32-core speedup calculated as $T_s/T_{32}$.

Across all the benchmarks, we notice that the overhead compared to mlton is comparable for both mlton-spoonhower and mlton-parmem and range between −15% and 51%. These measurements suggest that there is some overhead to parallelism but using a hierarchical memory manager is not significantly more expensive than the naive parallel memory manager employed in mlton-spoonhower. In fact, overheads of mlton-parmem are slightly better in 3 benchmarks than those of mlton-spoonhower. The manticore compiler incurs significantly larger overheads compared to mlton. This is consistent with earlier observations with the Manticore compiler [12], which focuses on performant parallel execution but not fast sequential execution. Our observed overhead of between 1.19 and 2.95 is in line with expectations [12].

In terms of scalability, which we can observe by comparing the Speedup “$S$” columns across compilers, our mlton-parmem compiler delivers the best speedups for all benchmarks except Raytracer. Figure 15 shows the speedup curves for individual benchmarks. In all benchmarks, we observe that mlton-parmem scales nicely up to 32 processors. While mlton-spoonhower also scales reasonably in some benchmarks, it consistently fails to scale as well as mlton-parmem. Indeed, in both SMM and DMM, we see the speedup of mlton-spoonhower starting to plateau towards 32 processors. In MemStress, mlton-spoonhower fails to achieve much more than 3× speedup regardless of the number of processors it utilizes. The manticore compiler scales very well, but also trails mlton-parmem, except in Raytracer. As programs compiled with manticore typically suffer from large overhead over the baseline, mlton, they require more processors in order to overcome the
overhead. Indeed, manticore requires about 3 processors just to
match the performance of mlton on the MemStress benchmark. In
Raytracer, the overhead of manticore against mlton poses less of a
problem and allows manticore to exhibit high scalability.

These results suggest that the proposed parallel hierarchical
memory manager can be implemented in the context of a high-
performance compiler such as MLton and can deliver good per-
formance. The overheads and the speedups that our mlton-parmem
compiler delivers suggest that the basic idea behind the approach
of coupling parallel scheduling and memory management is sound.
We note, however, that our comparative study does not carefully
account for many important factors that can have significant effects
on performance including various internal “hardwired” constants
such as block and page sizes and amortization constants that vary
between compilers. We leave a more careful implementation and a
more detailed experimental study for future work.

6. Related Work
There has been significant work on parallel garbage collection and
on scheduling for parallel programs, which we discuss below.

Garbage Collection. The work on parallel garbage collection
dates back at least to the multilisp [31] parallel collector. The
multilisp collector kept separate heaps for each processor and
had no shared heap. A variety of parallel collectors use a shared
heap, sometimes with a local block-allocated pool for allocation [6,
9, 21, 26, 32]. This requires significant synchronization among
the processors, and does not give the locality advantages of a
generational collector.

The Doligez-Leroy-Gonthier (DLG) collector [23, 24] intro-
duced the idea of having local nurseries (heaps) for each processor
in addition to a global heap. Memory is allocated in the local heaps
and promoted to the global heap on a local collection, or in various
other situations, such as when writing into state that is in the global
heap. The approach has a similar advantage as a single-level genera-
tional sequential collector—by the time a local heap is promoted to
the global heap, much of what has been allocated is already garbage.
Also if the local nursery fits in cache, its locations can be kept “warm”
in cache so that allocation is cheaper. Finally, the local collections
can be done independently. Several other parallel collectors have
refined the idea of local nurseries and a global heap [5, 8, 25, 37, 51].
These collectors differ in precisely how the two levels are divided
and what collection approach is used at each level (for example, the
original DLG collector allows the global heap to be collected concur-
tently using Dijkstra’s concurrent mark-sweep algorithm [22]. The
ABFR collector [8], on the other hand, uses a form of stop-the-world
copying collection on the global heap.

To handle mutable data, these two-level collectors ensure a
property similar to disentanglement by requiring that most or all
(depending on the strategy used by the individual collector) mutable
data be stored in the global heap. Thus, mutation-heavy programs
will require a great deal of global collection. In all of these collectors,
the decision to collect is based purely on when memory is full, or
perhaps when mutations happen across boundaries, but is not related
to scheduling decisions or other aspects of the tasks themselves.
In fact, they are agnostic to the tasks running on the processors.

Marlow et al. use a generational block-structured parallel col-
lector [38] with support for multiple generations. However, the
hierarchy does not form a tree, but rather is a sequence of global
heaps of varying size.

Pizlo et al. describe a hierarchical heap structure [43], which
has some similarities to ours. It is, however, designed for real-
time collectors, and in particular as a replacement for the Real-
time Specification for Java (RTSJ). The key difference is that the
hierarchical structure is meant to be (mostly) static and user-defined,

while ours is meant to be highly dynamic and invisible to the user.
The idea in their work is that real-time threads can use heaplets at
the leaf, and that each heaplet can support its own style of GC.

Bocchino et al. suggest using regions for parallel memory alloca-
tion [16]. Regions can be allocated in a tree hierarchy dynamically
based on the structure of the parallelism. Their motivation, however,
is very different. In particular they use typing rules on the regions
to statically prevent any race conditions. They have no discussion
of garbage collection within or across regions.

Morrisett et al. used an operational semantics for abstracting
garbage collection in a way similar to our operational semantics [40].
However, it was in a purely sequential context, and is more abstract
than ours, e.g. not even capturing the distinction between copying
and mark-sweep collection.

The results presented here build on our recent paper [4], which
proposed coupled computation and memory management.

Scheduling. Nearly all modern parallel programming languages
rely on a run-time scheduler to map tasks to processors. Determining
an optimal schedule is actually NP-hard [50], but a classic result by
Brent [17] shows that a 2-factor approximation can be computed by
using a greedy-scheduling principle. Brent’s theorem, however,
does not take into account the cost of the scheduling algorithm itself.
Over the years, many efficient scheduling algorithms implementing
Brent’s greedy principle have been developed and bounds have
been proven on runtime and space use [3, 15, 29, 30, 41], and
locality [1, 14]. These schedulers perform well in practice [3, 29, 48]
when the granularity of parallel tasks can be controlled so as to
prevent the creation of tiny tasks [2, 12, 49] Much of this work on
schedulers, however, does not try to couple the structure of memory
with that of the schedule as we do here.

7. Conclusion
This paper presents a memory management technique for parallel
functional programs that is based on the general strategy of closely
coupling the structure of the memory with that of the computation.
As we show, such close coupling enables using the invariants of the
semantics of the language, such as the disentanglement property, in
managing memory.

There are many interesting directions for future research. Here,
we consider fork-join programs, which include a relatively large
class of computations. Extending the technique to include other
forms of parallelism such as async-finish and futures may be inter-
esting and desirable. Another natural direction for future research
would be to extend the techniques to allow some mutable state. We
present an approach to realizing the semantics in practice by consid-
ering a work-stealing scheduler, by describing the data structures for
implementing hierarchical heaps, and by describing an algorithm
for performing garbage collection. The initial experimental results
for our extension of MLton suggest that the proposed techniques
can be implemented efficiently. The implementation, however, is
incomplete—it employs local collection only—and is not carefully
optimized. A natural next step would be to finish a complete and
optimized implementation that can perform both local and nonlocal
collections. Finally, it would be interesting to investigate if the tech-
niques described here can be applicable to other parallel languages
and systems [19, 28, 29, 33–36, 46].

Acknowledgments
This research is partially supported by the National Science Founda-
tion under grant numbers CCF-1320563 and CCF-1408940, by the
European Research Council under grant number ERC-2012-STG-
308246, and by Microsoft Research. We also thank the reviewers
for their insightful comments and in particular the idea of using
flattening to prove correctness.
References


