Efficient Tree-Traversals: Reconciling Parallelism and Dense Data Representations

CHAITANYA KOPARKAR, Indiana University, United States
MIKE RAINEY, Carnegie Mellon University, United States
MICHAEL VOLLMER, University of Kent, United Kingdom
MILIND KULKARNI, Purdue University, United States
RYAN R. NEWTON, Purdue University, United States

Recent work showed that compiling functional programs to use dense, serialized memory representations for recursive algebraic datatypes can yield significant constant-factor speedups for sequential programs. But serializing data in a *maximally* dense format consequently serializes the processing of that data, yielding a tension between density and parallelism. This paper shows that a disciplined, practical compromise is possible. We present Parallel Gibbon, a compiler that obtains the benefits of dense data formats *and* parallelism. We formalize the semantics of the *parallel location calculus* underpinning this novel implementation strategy, and show that it is type-safe. Parallel Gibbon exceeds the parallel performance of existing compilers for purely functional programs that use recursive algebraic datatypes, including, notably, abstract-syntax-tree traversals as in compilers.

CCS Concepts: • **Software and its engineering → Formal language definitions**; **Parallel programming languages**; **Compilers**.

Additional Key Words and Phrases: Parallelism, Region Calculus, Compilers, Data Representation

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

1 INTRODUCTION

Representing tree-like data as *pointer-free, serialized byte arrays* can be extremely efficient for tree traversals, as it minimizes pointer-chasing and maximizes locality [Goldfarb et al. 2013; Makino 1990; Meyerovich et al. 2011]. Moreover, by using an in-memory representation also suitable for external transfer and storage [Varda 2015; Yang et al. 2015], programs can rapidly process data without the overhead of deserialization. Traditionally, any such tree-layout optimizations would be implemented manually by the programmer—for example, in a scientific application with balanced trees.

Recent work, however, has shown the benefits of *automatically* compiling tree traversals to use denser representations, even for source programs written in a general-purpose language. The Gibbon compiler for a subset of Haskell exemplifies this approach [Vollmer et al. 2019, 2017]. While

Authors' addresses: Chaitanya Koparkar, Indiana University, United States, ckoparka@indiana.edu; Mike Rainey, Carnegie Mellon University, United States, me@mike-rainey.site; Michael Vollmer, School of Computing, University of Kent, United Kingdom, m.vollmer@kent.ac.uk; Milind Kulkarni, Purdue University, United States, milind@purdue.edu; Ryan R. Newton, Purdue University, United States, rrnewton@purdue.edu.

2475-1421/2021/8-ART91
https://doi.org/10.1145/3473596

the dense data representation strategy works well for sequential programs, there is an intrinsic tension if we want to parallelize these tree traversals. As the name implies, efficiently *serialized* data must often be read serially. To change that, first, enough *indexing* data must be left in the representation so parallel tasks can "skip ahead" and process multiple subtrees in parallel. Second, the allocation areas must be bifurcated to allow allocation of outputs in parallel.

In this paper, we offer a solution to these challenges. We propose a strategy where form follows function: where data representation is random-access only insofar as parallelism is needed, and both data representation and control flow "bottom out" to sequential pieces of work. That is, granularity-control in the data mirrors traditional granularity-control in parallel task scheduling. We demonstrate our solution by extending the Gibbon compiler with support for parallel computation, introducing *Parallel Gibbon*. We also extend *LoCal*, Gibbon's typed intermediate language, adding parallelism and give an updated formal semantics (Section 3).

In addition to tree traversals, we show that Parallel Gibbon can efficiently compile other parallel programs, such as sort and search algorithms (Section 5) to match or exceed the performance of the best existing parallel functional compilers. We choose a *functional* focus for three primary reasons:

- Many tree traversals have different input and output types—as in a compiler pass that converts between intermediate languages—which necessitates *out-of-place* traversals even in an imperative language.
- Even pure programs can use mutable data, via linear types. (Gibbon uses these and eschews the *IO monad*.)
- The purely-functional parallel Gibbon programs considered in this work are intrinsically *data-race free*.

The last point is worth emphasizing: every time a language adds both parallel constructs and mutable data, it enables data-races and must define a memory model to give them meaning. In this work, we extend Gibbon with linearly-typed primitives for mutable data[1] (Section 4.6), while keeping the language race-free. We claim that linearly-typed mutable data, efficient data representation, and compiler-supported parallelism are a synergistic combination. In Parallel Gibbon programs, as in other purely functional parallel programs, parallelism annotations not only don't introduce races but also *do not affect program semantics*, meaning that these programs are *deterministic* as well as data-race free.

Ultimately, we believe that this work shows one path forward for high-performance, purely-functional programs. Parallelism in functional programming has long been regarded as theoretically promising, but has a spottier track record in practice, due to problems in runtime systems, data representation, and memory management. Parallel Gibbon directly addresses these sore spots, showing how a purely-functional program operating on fine-grained, irregular data can also run fast (sequentially) and parallelize efficiently. This complements more well-trodden areas of compiler research on parallelism, such as dense and sparse collective operations on arrays [Abadi et al. 2015; Anderson et al. 2017; Blelloch 1992; Paszke et al. 2019]. That is, the approach described in this paper—for general-purpose, recursive functional programs, including tree traversals—could be combined with targeted EDSLs or libraries implementing additional parallel programming idioms, such as Haskell's Accelerate [Chakravarty et al. 2011]. Both determinism and data-race-freedom would be *compositional* within the functional-parallelism setting. Indeed, we have taken the first steps in this direction, adding a small set of parallel array primitives to Gibbon (Section 4.6).

In this paper, we make the following contributions:

- We introduce the first compiler that combines parallelism with automatic dense data representations for trees. While dense data [Vollmer et al. 2019] and efficient parallelism [Raghunathan

[1]Leveraging the Linear Haskell [Bernardy et al. 2017] extensions now available in GHC 9

et al. 2016; Westrick et al. 2019] have been shown to independently yield large speedups on tree-traversing programs, our system is the first to combine these sources of speedup, yielding the fastest known performance generated by a compiler for this class of programs.
- We formalize the semantics of a *parallel location calculus* (Section 3) that underpins the compiler, including a proof of its type-safety (Section 3.4). To do so, we extend prior work on location calculi [Vollmer et al. 2019], which in turn builds on work in region calculi [Tofte and Talpin 1997].
- We evaluate our implementation (Section 4) on several benchmarks from the literature (Section 5). On a single thread, our implementation is 1.93×, 2.53×, and 2.14× faster than MaPLe [Westrick et al. 2019] (an extension of MLton), OCaml, and GHC, respectively. When utilizing 48 threads, our geomean speedup is 1.92×, 3.73× and 4.01×, meaning that the use of dense representations to improve sequential processing performance *coexists with scalable parallelism*. Most notably, the speedup on a five-pass compiler drawn from a university compiler course was 1.02×, 2.2× and 10.7× over those alternative languages.

2 OVERVIEW

We give a high-level overview of the ideas presented in this paper using a program given in Figure 1a (with a larger sample program given in the Appendix of the extended version [Koparkar et al. 2021]). Because the techniques we present for compiling tree-traversals are directly applicable to *compilers themselves*, we use a miniature compiler pass as our example. The example defines a datatype `Exp` which represents the abstract syntax of a language that supports integer arithmetic, and a function `constFold` that implements constant folding for this language. Constant folding is a common compiler optimization in which expressions with constant operands are evaluated at compile time, thus improving the run-time performance. But here we are trying to optimize the performance of *the constant folding pass itself* rather than the performance of the program produced by constant folding. `constFold` walks over the abstract-syntax-tree, and substitutes all expressions of the form `(Plus (Lit i) (Lit j))` with `(Lit (i+j))`. We only show a simplified `constFold` — for example it doesn't recur on the children of `Plus` before checking if they're literals — to keep it simple enough to serve as a running example.

The program in Figure 1a is written using the front-end language for Gibbon, a polymorphic, higher-order subset of Haskell, with strict rather than lazy evaluation. The `(||)` operator used on line 14 denotes a parallel tuple — it marks its operands to evaluate in parallel with each other. But with a purely functional source language, it is semantically equivalent to a sequential tuple, i.e., replacing all "||" occurences with "," yields a valid program. We will return to this topic in Section 4.

Gibbon uses LoCal (short for location calculus) as an intermediate representation (IR) with explicit byte-addressed, mostly-serialized data layout. To go from the vanilla Haskell front-end language to LoCal, it performs *location inference*, a variant of region inference [Tofte et al. 2004; Tofte and Talpin 1997], on the input programs. The LoCal IR code generated by Gibbon for the `constFold` function is shown in Figure 1b. In the following, we use it to sketch out how LoCal works.

2.1 A Primer on Location-Calculus

LoCal is a type-safe IR that represents programs operating on densely encoded (serialized) data. All serialized values live in regions, which are unbounded memory buffers that never overlap and that store the raw data. All programs make explicit not only the region to which a value belongs to, but also a *location* at which that value is written. In our notation, a location l^r resides in region r. Locations are fine-grained indices into a region, but unlike pointers in languages like C, arbitrary arithmetic on locations is not allowed. Locations are only introduced relative to other locations, and they can be written to only once. Once allocated at a particular location, a value cannot be

```
1  data Exp = Lit Int
2           | Plus Exp Exp
3           | Sub Exp Exp
4           | Let Sym Exp Exp
5           ...
6
7  constFold :: Exp → Exp
8  constFold exp = case exp of
9    Lit i → Lit i
10   Plus e1 e2 →
11     case (e1, e2) of
12       (Lit i,Lit j) → Lit (i+j)
13       _ → let (e3,e4) =
14               ( constFold e1 ||
15                 constFold e2 )
16             in Plus e3 e4
17   Sub e1 e2 →
18     ...
```

(a) Constant folding written using the front-end language for Gibbon (Haskell).

```
1  constFold: ∀l1^r1 l2^r2. Exp@l1^r1 → Exp@l2^r2
2  constFold [l1^r1 l2^r2] exp = case exp of
3    Lit (i:Int@li^r1) → (Lit l2^r2 i)
4    Plus (e1: Exp@la^r1) (e2: Exp@lb^r1) →
5      case (e1,e2) of
6        (Lit(i:Int@lc^r1), Lit(j:Int@ld^r1))
7          → (Lit l2^r2 (i+j))
8        _
9          →
10         letloc l3^r2 = l2^r2 + 1 in
11         let e3 : Exp@l3^r2 =
12           constFold [la^r1 l3^r2] e1 in
13         letloc l4^r2 = after(Exp@l3^r2) in
14         let e4 : Exp@l4^r2 =
15           constFold [lb^r1 l4^r2] e2 in
16         (Plus l2^r2 e3 e4)
17   Sub (e1: Exp@la^r1) (e2: Exp@lb^r1) →
18     ...
```

(b) Figure 1a compiled into LoCal IR by Gibbon.

Fig. 1. Constant folding.

shared with another location (within the same region or across regions), and it has to be *copied* to allocate it at a different location. (In practice, the Gibbon compiler supports sharing using pointers, which we discuss in Section 4.3.)

A new location is either: at the start of a region, one unit past an existing location, or *after* all elements of a value rooted at an existing location. In the program given in Figure 1b, the location $l_3{}^{r_2}$ is one past the location $l_2{}^{r_2}$ (line 10) and $l_4{}^{r_2}$ is after every element of the value rooted at location $l_3{}^{r_2}$ (line 13). Any expression that allocates takes an extra argument: a location-region pair that specifies where the allocation should happen. The types of such expressions are decorated with these location-region pairs. For example, the `(Lit` $l_2{}^{r_2}$ `i)` data constructor (line 3) allocates a tag at location l_2 in region r_2, and has type `(Exp@`$l_2{}^{r_2}$`)`. Any scalar arguments passed to a data constructor, such as the unpacked integer `i` in this case, are allocated immediately after the tag. Functions may be polymorphic over any of their input or output locations, and these locations are provided at call-sites. In the example, the function `constFold` is polymorphic over an input location $l_1{}^{r_1}$ and an output location $l_2{}^{r_2}$, and values for these are given at all call-sites. In spite of the forall quantifier in its type signature, the input and output regions given at its call-site must be distinct ($r_1 \neq r_2$) to prevent overwrites. This property is checked by LoCal's type-system (described in Section 3.3), which makes multiple writes to any location illegal—with the use of a nursery environment—ensuring that function calls like (`constFold` [$l_x{}^{r_x}$ $l_x{}^{r_x}$] `x`) don't type check.

2.1.1 Sequential Execution Model. LoCal has a dynamic semantics which can run programs sequentially [Vollmer et al. 2019]. In this model, regions are represented as serialized heaps, where each heap is an array of cells that can store primitive values (data constructor tags, numbers, etc). A write operation, such as the application of a data constructor, allocates to a fresh cell on the heap, and a read operation reads the contents of a cell. Performing multiple reads on a single cell is safe, and the type-system ensures that each cell (location) is written to only once. At run time, locations in the source language translate to heap indices, which are the concrete addresses of

the cells where reads/writes happen. Computing addresses of locations which are at the start of a region, or one past another location is straightforward — the addresses get initialized to `0` and `(prev + 1)` respectively. But evaluating an `after` expression, to get an address one past the end of another, variable-sized value, requires more work.

A naive computational interpretation of this `after` is to simply scan over a value to compute its end. In LoCal's formal model, this is referred to as the *end-witness judgment*. Locations computed via `after` are used during both read and write operations. For example, when a LoCal **`case`** expression pattern matches on `(Plus e1 e2)`, it has to scan past `e1` in order to know the starting address of `e2`, which adds $O(n)$ amount of extra work in a *fully* serialized representation! In practice, if values are read in the same order in which they were serialized, a linear scan can be avoided by *tracking end-witnesses* that are naturally computed in the evaluation of the program, for example, by having every read return the address of the cell after it. Intuitively, we can imagine there being a single read pointer that is used to perform all reads in the program. It always points to the next cell to be read on heap, and each read advances it by one. When the program starts executing, the read pointer starts at the beginning of the heap and it chugs along in a continuous fashion. Allocating a serialized value can be thought of in a similar way — that there is a single allocation pointer that starts at the beginning of the heap, and moves along its length performing writes, as illustrated in Figure 2b. To avoid changing the asymptotic complexity of programs which read values out-of-order, the Gibbon compiler by default inserts some offset information — such as pointers to some fields of a data constructor — back into the representation. But it doesn't allow *out-of-order allocations*, which will be needed as we add parallelism to LoCal (Section 2.2).

2.1.2 Sequential Execution Model, Example. To make this execution model concrete, let us go over a step-by-step trace of the semantics executing `constFold` on `(Plus (Lit 20) (Plus (Lit 10) (Lit 12)))`. The execution trace is given in Figure 2. The store S maps regions to their corresponding heaps, and the location map M maps symbolic locations to their corresponding heap indices. The evaluation starts at `(constFold [`$l_1{}^{r_1}$ $l_2{}^{r_2}$`] e)`, and is given a store containing a fully allocated input region r_1 and an empty region r_2 to allocate the output, along with a location map containing the locations $l_1{}^{r_1}$ and $l_2{}^{r_2}$ initialized to the starting addresses of these regions. Since the input region has a `Plus` at the top, execution continues at line 5. The pattern match binds the locations l_a^0 and l_b^0 to the addresses of the sub-expressions `(Lit 20)` and `(Plus (Lit 10) (Lit 12))` respectively . Since both the sub-expressions are not constants, execution continues at line 10. Then, the output location of the first sub-expression, $l_3{}^{r_2}$, is defined to be one past $l_2{}^{r_2}$, and `constFold` is invoked recursively on this sub-expression. Step 4 *copies*[2] the first sub-expression by writing a tag L (short for `Lit`), followed by the integer 20 on the heap. Then, the output location of the second sub-expression, $l_4{}^{r_2}$, is defined to be one past every element of the first sub-expression, which occupies two cells after the 0^{th} cell. Thus, $l_4{}^{r_2}$ gets initialized with the address of the 3^{rd} cell. `constFold` is now invoked recursively for the second sub-expression. Following similar steps, the second sub-expression is allocated at $l_4{}^{r_2}$. Since the second sub-expression is a `Plus` with constant operands, it is transformed to `(Lit 22)`. Finally, Step 16 writes the tag P (short for `Plus`) which completes the construction of the full expression, `(Plus (Lit 20) (Lit 22))`.

[2]This value is copied because line 3 in Figure 1b has a data constructor `(Lit` $l_2{}^{r_2}$ `i)` on the right hand side of the case alternative. If we update the program to return the input expression `exp` directly, Gibbon would allocate a pointer and the value `(Lit 20)` would be *shared* between the input and the output regions. We discuss how sharing works in Section 4.3.

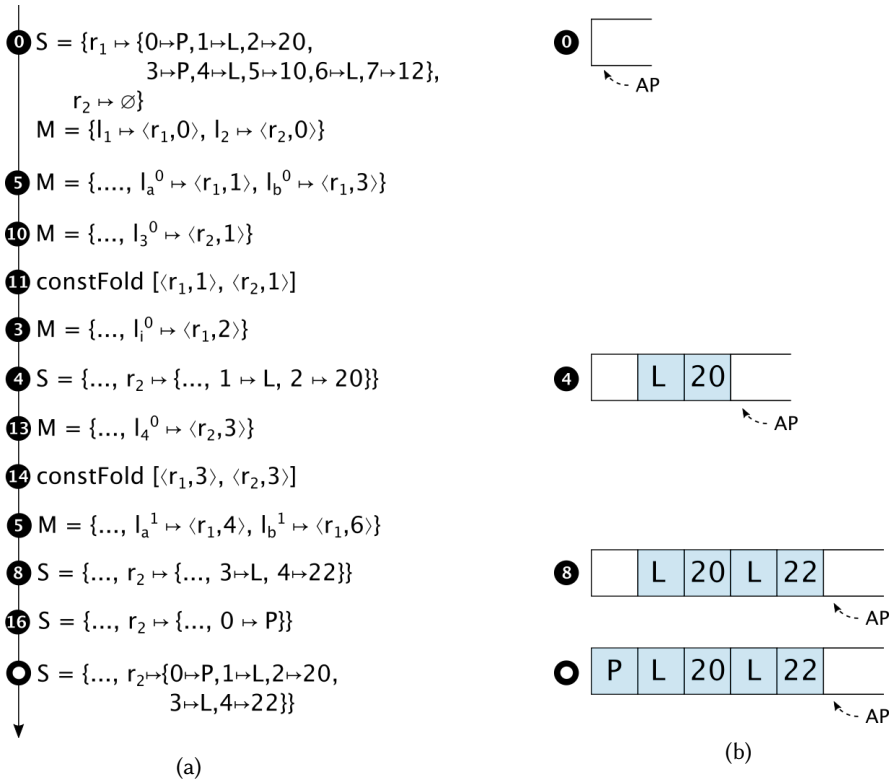


Fig. 2. (a) Sequential, step-by-step execution of the program from Figure 1b, and (b) the heap operations corresponding to the output region r_2. Each step is named after its line number in the program and only shows the changes relative to the previous step. *AP* is the allocation pointer. `P` is short for `Plus`, and `L` is short for `Lit`.

2.2 Parallelism in Location-Calculus

In this section, we outline various *latent* opportunities for parallelism that exist in LoCal programs (irrespective of annotation with "‖"). The first kind of parallelism is available when programs access the store in a read-only fashion, such as in an interpreter, for example.

```
interp : ∀ l^r . Exp @ l^r → Int
interp [l^r] t = case t of
  Lit (i : Int @ l_i^r) → i
  Plus (e1 : Exp @ l_a^r) (e2 : Exp @ l_b^r) →
    (interp [l_a^r] a) + (interp [l_b^r] b)
  ...
```

Even though the recursive calls in the `Plus` case can safely evaluate in parallel, there is a subtlety: parallel evaluation is efficient only if the `Plus` constructor stores offset information for its right child node. If it does, then the address of `e2` can be calculated in constant time, thereby allowing the calls to proceed immediately in parallel. If there is no offset information, then the overall tree traversal is necessarily sequential, because the starting address of `e2` can be obtained only after a full traversal of `e1`. As such, there is a tradeoff between space and time, that is, the cost of the space to store the offsets, versus the time of the sequential traversal forced by the absence of offsets.

Programs that write to the store also provide opportunities for parallelism. The most immediate such opportunity exists when the program performs writes that affect different regions. Such writes can happen in parallel because different regions cannot overlap in memory. There is another kind of parallelism that is more challenging to exploit, but is at least as important as the others: intra-region parallelism that can be realized by allowing different fields of the same constructor to be filled in parallel. This is crucial in LoCal programs, where large, serialized data (large trees or DAGs) frequently occupy only a small number of regions, and yet there are opportunities to exploit parallelism in their construction.

Consider the case in Figure 1b which recursively calls `constFold` on the sub-expressions of `Plus`. If we want to access the parallelism between the recursive calls, we need to break the data dependency that the right branch has on the left. The starting address of the right branch, namely ${l_4}^{r_2}$, is assigned to be end witness of the left branch by the **`after`** expression. But the end witness of the left branch is, in general, known only after the left branch is completely filled, which would effectively sequentialize the computation. One non-starter would be to ask the programmer to specify the size of the left branch up front, which would make it possible to calculate the starting address of the right branch. Unfortunately, this approach would introduce safety issues, such as incorrect size information, of exactly the kind that LoCal is designed to prevent. Instead, we explore an approach that is safe-by-construction and efficient, as we explain next.

3 REGION-PARALLEL LOCAL

To address the challenges of parallel evaluation—in concert with dense, mostly-serialized data representations—we start by presenting an execution model, LoCal$^{\text{par}}$, which can utilize *all* potential parallelism in LoCal programs. Parallelism in this formal model is generated *implicitly*, by allowing every let-bound expression to potentially evaluate in parallel with the body. Accordingly, the language omits explicit parallelism "hints" ($\|$). That is, you'll see in the next sections that implicitly parallel **`let`** has both a sequential and parallel evaluation rule. By modeling every possible parallelization, the formal model is general — it formalizes all possible valid parallel schedules, and all valid heap layouts. We return to the pragmatic issue of selecting *efficient* parallelizations, i.e. granularity control, in Section 4.1.

3.1 Region Memory and Parallel Tasks

In the formal model, while parallelism is implicit, there is still a restriction that at most one task allocates in a given region at a time. To realize intra-region parallelism, the model introduces fresh, intermediate regions as needed, that is, when the schedule takes a parallel evaluation step for a given let-bound expression, and the body tries to allocate in the same region. To demonstrate this, let us consider a trace of the region-parallel evaluation of the program from Figure 1b, corresponding to the schedule shown in Figure 3, where the recursive calls to `constFold` on lines 12 and 14 run in parallel with each other. The parallel fork point for the first recursive call occurs on the 11^{th} step of the trace. At this point, the evaluation of the let-bound expression results in the creation of a new child task, and the continuation of the body of the let expression in the parent task.

Each task has its own private view of memory, which is realized by giving the child and parent task copies of the store S and location map M. These copies differ in one way, however: each sees a different mapping for the starting location of `e3`, namely ${l_3}^{r_2}$. The child task sees the mapping ${l_3}^{r_2} \mapsto \langle r_2, 1\rangle$, which is the ultimate starting address of `e3` in the heap. The parent task sees a different mapping for ${l_3}^{r_2}$, namely $\langle r_2, \texttt{ivar}\ e3\rangle$. This address is an *ivar*: it behaves exactly like an I-Var [Arvind et al. 1989], and, in our example, stands in for the completion of the memory being filled for `e3`, by the child task. Any expression in the body of the let expression that tries to read from this location blocks on the completion of the child task.

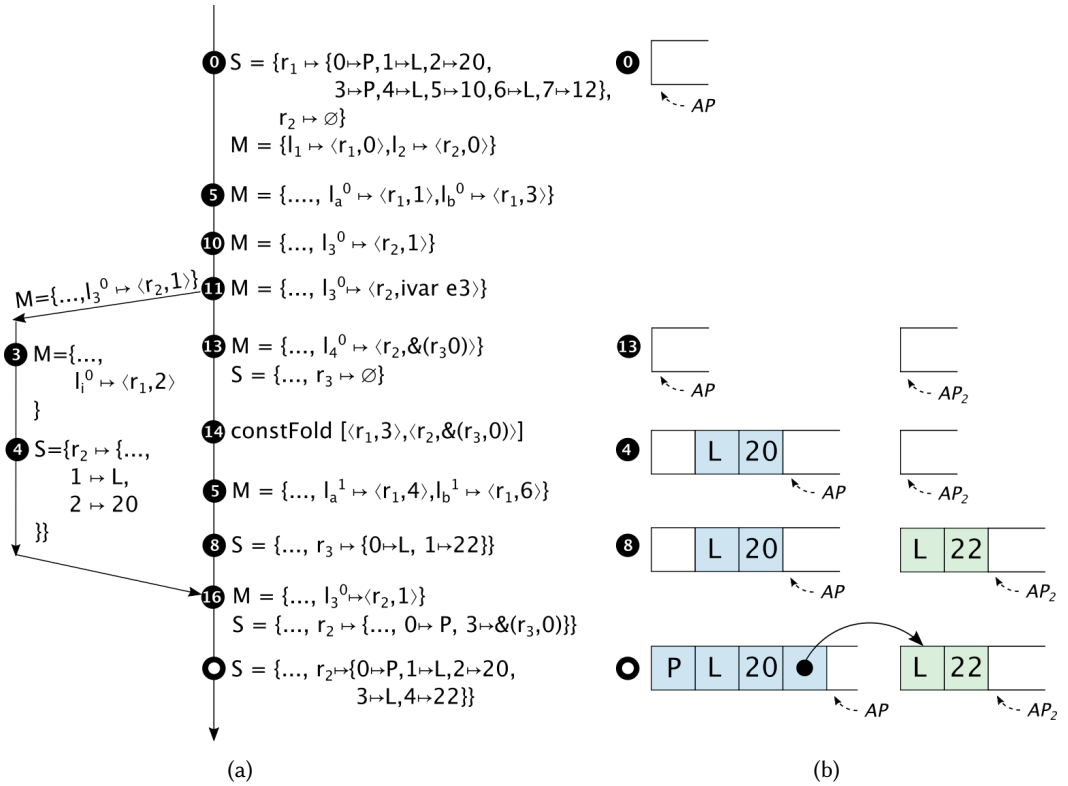


Fig. 3. (a) Parallel, step-by-step execution of the program from Figure 1b such that parallel allocations happen only in separate regions, and (b) the heap operations corresponding to the output region r_2. Each step is named after its line number in the program and only shows the changes relative to the previous step. `P` is short for `Plus`, and `L` is short for `Lit`.

The only exception to this blocking-rule is a letloc `after` expression, which is handled differently. Such an expression occurs at line 13, just after the parent continues after the fork point. At this step, the parent task uses an **`after`** expression to assign an appropriate location for the starting address of `e4`, one past every byte occupied by `e3`. If we synchronize with the child task here, the computation will effectively be sequential. In order to avoid that, the starting address of `e4` is assigned to be $l_4^{r_2} \mapsto \langle r_2, \&(r_3, 0)\rangle$. This address is an *indirection* pointing to the start of fresh region r_3, and causes the parent task to allocate `e4` in the region r_3 instead of r_2, which is being allocated to by the child task, thus maintaining the single-threaded-per-region allocation invariant. The parent and child tasks have, in effect, two different allocation pointers for what will functionally be the same region (after joining). The use of `e3` on line 16 forces the parent task to join with its child task. In particular, $\langle r_2, \texttt{ivar}\ e3\rangle$ is substituted by $\langle r_2, 1\rangle$, the starting address of `e3`, in the expression and the location map M. Also, all the new entries in the location map M and store S of the child are merged into the corresponding environments in the parent. Finally, the regions r_2 and r_3 are linked with a pointer, corresponding to the indirection pointer that was added for the starting address of `e4`.

$K \in$ Data Constructors, $\tau_c \in$ Type Constructors,
$x, y, f \in$ Variables, $l, l^r \in$ Symbolic Locations,
$r \in$ Regions, $i, j \in$ Concrete Region Indices,

Top-Level Programs	top	$::= \overrightarrow{dd}\ ;\ \overrightarrow{fd}\ ;\ e$
Datatype Declarations	dd	$::= \texttt{data}\ \tau_c = \overrightarrow{K\ \overrightarrow{\tau}}$
Function Declarations	fd	$::= f : ts; f\ \overrightarrow{x}\ = e$
Located Types	$\hat{\tau}$	$::= \tau@l^r$
Types	τ	$::= \tau_c$
Type Scheme	ts	$::= \forall_{\overrightarrow{l^r}}.\ \overrightarrow{\hat{\tau}} \rightarrow \hat{\tau}$
Extended Region Indices	$i\diamond, j\diamond$	$::= i \mid \texttt{ivar}\ x \mid \&(r, i)$
Concrete Locations	cl	$::= \langle r, i\diamond \rangle^l$
Values	v	$::= x \mid cl$
Expressions	e	$::= v$
		$\mid f\ [\overrightarrow{l^r}]\ \overrightarrow{v}$
		$\mid K\ l^r\ \overrightarrow{v}$
		$\mid \texttt{let}\ x : \hat{\tau} = e\ \texttt{in}\ e$
		$\mid \texttt{letloc}\ l^r = le\ \texttt{in}\ e$
		$\mid \texttt{letregion}\ r\ \texttt{in}\ e$
		$\mid \texttt{case}\ v\ \texttt{of}\ \overrightarrow{pat}$
Pattern	pat	$::= K\ \ (\overrightarrow{x : \hat{\tau}})\ \rightarrow\ e$
Location Expressions	le	$::= \texttt{start}\ r$
		$\mid l^r + 1$
		$\mid \texttt{after}\ \hat{\tau}$
Store	S	$::= \{\ r_1 \mapsto h_1,\ \ldots\ , r_n \mapsto h_n\ \}$
Heap Values	hv	$::= K \mid \&(r, i)$
Heap	h	$::= \{\ i_1 \mapsto hv_1,\ \ldots\ , i_n \mapsto hv_n\ \}$
Location Map	M	$::= \{\ l_1^{r_1} \mapsto cl_1,\ \ldots\ , l_n^{r_n} \mapsto cl_n\ \}$
Sequential States	t	$::= S; M; e$
Parallel Tasks	T	$::= (\hat{\tau}, cl, t)$
Task Set	$\mathbb{T}$	$::= \{\ T_1, \ldots, T_n\ \}$

Fig. 4. Grammar of LoCal$^{\text{par}}$.

3.2 Syntax and Operational Semantics

In this section, we present the formal semantics of our parallel location calculus, LoCal$^{\text{par}}$. This semantics has also been mechanically tested in PLT Redex [Felleisen et al. 2009]. The grammar for the language is given in Figure 4. Again, all parallelism in this model language is introduced implicitly, by evaluating `let` expressions. There is no explicit syntax for introducing parallelism in

our language, and consequently the language is, from the perspective of a client, exactly same as the sequential language [Vollmer et al. 2019].

The parallel operational semantics does, however, differ from the sequential semantics, most notably from the introduction of a richer form of indexing in regions. Whereas in sequential LoCal a region index consists simply of a non-negative integer, it is enriched to an extended region index $i\diamond$ in LoCal$^{\text{par}}$. It consists of either a concrete index i, an `ivar` x, or an indirection pointer $\&(r, i)$. A concrete index is a non-negative integer that specifies the final position of a value in a region. An ivar is a synchronization variable that is used to coordinate between parallel tasks. For example, the `ivar` $e3$ in the sample trace in Figure 3, is used to synchronize with the child task that is allocating `e3`. An indirection $\&(r, i)$ points to the address i in the region r, and is used to link together different *chunks* of the same logical region, which may have been introduced to enable intra-region parallel allocation. For example, in the sample trace in Figure 3, a pointer $\&(r_3, 0)$ written at the end of the value `e3` links it with the value `e4`, which is allocated to a separate region r_3. And a concrete location cl is enriched to a pair $\langle r, i\diamond \rangle$, of a region r, and an extended region index $i\diamond$. The state configurations of LoCal$^{\text{par}}$ appear at the bottom of Figure 4. Just like in sequential LoCal, a sequential state of LoCal$^{\text{par}}$, t, contains a store S, location map M, and an expression e. But by using enriched concrete locations, the location map also has the ability to contain indirection pointers. A value that can be written to a heap, hv, is similarly enriched to allow indirection pointers.

3.2.1 Sequential Transitions. A subset of the sequential transition rules are given in Figure 5. The rules are close to the original sequential rules, except for some minor differences. For the rule D-DataConstructor, we need to handle the case where an indirection is assigned to the source symbolic location l^r. For this purpose, we use a metafunction $\hat{M}$ (formally defined in the Appendix of the extended version [Koparkar et al. 2021]) that can dereference indirection pointers when looking up its address in the location map M. With respect to the rule D-LetLoc-After-NewReg, we now allow the concrete location assigned to the source location ${l_0}^r$ to hold an ivar. The purpose of this relaxation is to allow an expression downstream from a parallelized **`let`** binding to continue evaluating in parallel with the task that is evaluating the let-bound expression. The task evaluating the **`after`** expression continues by using an indirection pointing to the start of a fresh region r'. The effect is to make $\langle r', 0 \rangle$ the setting for the allocation pointer for the task. If the source location ${l_0}^r$ is assigned to hold a concrete index i, the rule D-LetLoc-After yields an address by using the the end-witness judgment. The remaining rules are similar to sequential LoCal, and are available in the Appendix of the extended version [Koparkar et al. 2021].

3.2.2 Parallel Transitions. We generalize a sequential state to a parallel task T by adding two more fields: a located type and a concrete location, which together describe the type and location of the final result allocated by the task. A parallel transition in LoCal$^{\text{par}}$ takes the form of the following rule, where any number of tasks in a task set $\mathbb{T}$ may step together.

$$\mathbb{T} \Longrightarrow_{rp} \mathbb{T}'$$

In each step, a given task may make a sequential transition, it may fork a new parallel task, it may join with another parallel task, or it may remain unchanged.

The parallel transition rules are given in Figure 6. In these rules, we model parallelism by an interleaving semantics. Any of the tasks that are ready to take a sequential step may make a transition in rule D-Par-Step. A parallel task can be spawned by the D-Par-Let rule, from which an in-flight **`let`** expression breaks into two tasks. The child task handles the evaluation of the **`let`**-bound expression e_1, and the parent task handles the body e_2. To represent the future location of the let-bound expression, and to create a data dependency on it, the rule creates a fresh ivar,

[D-DataConstructor]
$$S; M; K\ l^r\ \overrightarrow{v} \Rightarrow S'; M; \langle r', i' \rangle$$
where $S' = S \cup \{\, r' \mapsto (i' \mapsto K) \,\};\ \langle r', i' \rangle = \hat{M}(l^r)$

[D-LetLoc-After]
$$S; M; \texttt{letloc}\ l^r = \texttt{after}\ \tau@{l_0}^r\ \texttt{in}\ e \Rightarrow S; M'; e$$
where $\langle r, i \rangle = \hat{M}({l_0}^r);\ \tau; \langle r, i \rangle; S \vdash_{ew} \langle r, j \rangle$
$M' = M \cup \{\, l^r \mapsto \langle r, j \rangle \,\}$

[D-LetLoc-After-NewReg]
$$S; M; \texttt{letloc}\ l^r = \texttt{after}\ \tau@{l_0}^r\ \texttt{in}\ e \Rightarrow S'; M'; e$$
where $\langle r, \texttt{ivar}\ x \rangle^{l_0} = \hat{M}({l_0}^r); r'$ fresh
$S' = S \cup \{\, r' \mapsto \varnothing \,\}$
$M' = M \cup \{\, l^r \mapsto \langle r, \&(r', 0) \rangle \,\}$

[D-Case]
$$S; M; \texttt{case}\ \langle r, i \rangle^{l^r}\ \texttt{of}\ [\ldots, K\ \overrightarrow{(x : \tau@l^r)}\ \rightarrow\ e, \ldots] \Rightarrow S; M'; e'$$
where $e' = e[\langle r, \overrightarrow{w} \rangle^{\overrightarrow{l^r}} / \overrightarrow{x}]$
$M' = M \cup \{\, \overrightarrow{l_1^r} \mapsto \langle r, i+1 \rangle, \ldots, \overrightarrow{l_{j+1}^r} \mapsto \langle r, \overrightarrow{w_{j+1}} \rangle \,\}$
$\overrightarrow{\tau_1}; \langle r, i+1 \rangle; S \vdash_{ew} \langle r, \overrightarrow{w_1} \rangle$
$\overrightarrow{\tau_{j+1}}; \langle r, \overrightarrow{w_j} \rangle; S \vdash_{ew} \langle r, \overrightarrow{w_{j+1}} \rangle$
$K = S(r)(i);\ j \in \{\, 1, \ldots, n-1 \,\};\ n = |\overrightarrow{x : \hat{\tau}}|$

[D-Let-Expr]
$$\frac{S; M; e_1 \Rightarrow S'; M'; e_1' \qquad e_1' \neq v}{S; M; \texttt{let}\ x : \hat{\tau} = e_1\ \texttt{in}\ e_2 \Rightarrow S'; M'; \texttt{let}\ x : \hat{\tau} = e_1'\ \texttt{in}\ e_2}$$

[D-Let-Val]
$$S; M; \texttt{let}\ x : \hat{\tau} = v_1\ \texttt{in}\ e_2 \Rightarrow S; M; e_2[v_1/x]$$

Fig. 5. Selected dynamic semantics rules (sequential transitions).

which is passed to the body of the **let** expression. This same ivar is also the target concrete location of the child task, thereby indicating that it produces this value.

A task can satisfy a data dependency in a rule such as D-Par-Case-Join, where a **case** expression is blocked on the value located at `ivar` x_c, by joining with the task producing the value. Because each task has a private copy of the store and location map, the process of joining two tasks involves merging environments. The merging of the task memories is performed by the metafunctions *MergeS* and *MergeM*, defined formally in the Appendix of the extended version [Koparkar et al. 2021]. We merge two stores by merging the heaps of all the regions that are shared in common by the two stores, and then by combining with all regions that are not shared. We merge two heaps by taking the set of all the heap values at indices that are equal, and all the heap values at indices in only the first and only the second heap. The merging of location maps follows a similar pattern, but is slightly complicated by its handling of locations that map to ivars. In particular, for any

[D-Par-Step]

$$\frac{S; M; e \Rightarrow S'; M'; e'}{T_1, \ldots, (\hat{\tau}, cl, S; M; e), \ldots T_n \Longrightarrow_{rp} T_1, \ldots, (\hat{\tau}, cl, S'; M'; e'), \ldots T_n}$$

[D-Par-Let-Fork]

$T_1, \ldots, (\hat{\tau}, cl, S; M; e), \ldots T_n \Longrightarrow_{rp} T_1, \ldots, (\hat{\tau}_1, cl'_1, S; M; e_1), \ldots T_n, (\hat{\tau}, cl, S; M_2; e'_2)$

where $e = (\texttt{let}\ x : \hat{\tau}_1 = e_1\ \texttt{in}\ e_2);\ \hat{\tau}_1 = \tau_1 @ {l_1}^{r_1}$
x_1 fresh; $cl'_1 = \langle r_1, \texttt{ivar}\ x_1 \rangle;\ e'_2 = e_2[cl'_1/x]$
$M = \{\, {l_1}^{r_1} \mapsto cl_1 \,\} \cup M'$
$M_2 = \{\, {l_1}^{r_1} \mapsto cl'_1 \,\} \cup M'$

[D-Par-Case-Join]

$T_1, \ldots, T_c, \ldots, T_n \Longrightarrow_{rp} T_1, \ldots, T'_c, \ldots T_n,$
where
$T_c = (\hat{\tau}_c, cl_c, S_c; M_c; e_c)$
$e_c = \texttt{case}\ \langle r, \texttt{ivar}\ x_c \rangle^{l_c}\ \texttt{of}\ \overrightarrow{pat}$
$T_p \in \{\, T_1, \ldots, T_n \,\} = (\tau_p @ {l_p}^r, \langle r, \texttt{ivar}\ x_c \rangle, S_p; M_p; \langle r, i_p \rangle)$
$M_3 = MergeM(M_p, M_c);\ S_3 = MergeS(S_p, S_c)$
$e'_c = \texttt{case}\ \langle r, i_p \rangle^{l_p}\ \texttt{of}\ \overrightarrow{pat}[i_p/\texttt{ivar}\ x_c]$
$T'_c = (\hat{\tau}_c, cl_c, S_3; M_3; e'_c)$

[D-Par-DataConstructor-Join]

$T_1, \ldots, (\hat{\tau}, cl, S; M; e), \ldots, T_n \Longrightarrow_{rp} T_1, \ldots, T', \ldots, T_n$

where $e = K\ l^r\ \overrightarrow{v};\ \langle r, \texttt{ivar}\ x_j \rangle = \overrightarrow{v_j};\ T_c \in \{\, T_1, \ldots, T_n \,\}$
$T_c = (\tau_c @ {l_c}^r, \langle r, \texttt{ivar}\ x_j \rangle, S_c; M_c; \langle r, i_c \rangle^{l_c})$
$M' = MergeM(M_c, M);\ S' = MergeS(S_c, S)$
$n = |\overrightarrow{v}|;\ \overrightarrow{v'} = [\overrightarrow{v_1}, \ldots, \overrightarrow{v_{j-1}}, \langle r, i_c \rangle^{l_c}, \overrightarrow{v_{j+1}}, \ldots, \overrightarrow{v_n}]$
$\tau_j = TypeOfField(K, j);$
$S'' = LinkFields(S', M, \tau_j, \langle r, i_c \rangle^{l_c})$ if $j \neq n$ else S'
$e' = K\ l^r\ \overrightarrow{v'};\ T' = (\hat{\tau}, cl, S''; M'; e')$

Fig. 6. Dynamic semantics rules (parallel transitions).

location where one of the two location maps holds an ivar and the other one holds a concrete index, we assign to the resulting location map the concrete index, because the concrete index contains the more recent information. After merging the environments, all occurrences of $\texttt{ivar}\ x_c$ are eliminated in the continuation, and are replaced by the index i_p, that represents the starting index of the value produced by the task T_p. Join points in LoCal$^{\text{par}}$ are, in general, deterministic, because they only *increase* the information held by the parent task.

The rule D-Par-DataConstructor-Join handles the case where a data constructor is blocked on the value of its j^{th} field, and it joins with the task producing that value. It is similar to D-Par-Case-Join, and also requires merging environments. But depending on the schedule of execution, if the $(j+1)^{th}$ field of this constructor was computed in parallel with the j^{th} field, they will both be allocated to separate regions, due to the way the rule D-LetLoc-After-NewReg works. These fields have to be reconciled to simulate a single region. For this purpose, we use a metafunction *LinkFields* — defined formally in the Appendix of the extended version [Koparkar et al. 2021] — which stitches

$$\text{[T-Task]}\quad \frac{\Gamma;\Sigma;C;A;N \vdash A';N';e:\hat{\tau}}{\Gamma;\Sigma;C;A;N \vdash_{task} A';N';(\hat{\tau},cl,S;M;e)} \qquad \text{[T-Taskset-Empty]}\quad \mathbb{\Gamma};\mathbb{\Sigma};\mathbb{C};\mathbb{A};\mathbb{N} \vdash_{taskset} \mathbb{A};\mathbb{N};\varnothing$$

$$\text{[T-Taskset]}\quad \frac{\begin{array}{c}(\hat{\tau},cl,S;M;e) = T_i \quad \Gamma = \mathbb{\Gamma}(cl) \quad \Sigma = \mathbb{\Sigma}(cl) \quad C = \mathbb{C}(cl) \quad A = \mathbb{A}(cl) \quad N = \mathbb{N}(cl) \\ \Gamma;\Sigma;C;A;N \vdash_{task} A';N';T_i:\hat{\tau} \quad \mathbb{A}' = \mathbb{A} \cup \{\, cl \mapsto A' \,\} \quad \mathbb{N}' = \mathbb{N} \cup \{\, cl \mapsto N' \,\} \\ \mathbb{\Gamma};\mathbb{\Sigma};\mathbb{C};\mathbb{A}';\mathbb{N}' \vdash_{taskset} \mathbb{A}'';\mathbb{N}'';\{\, T_1,\ldots,T_n \,\}\end{array}}{\mathbb{\Gamma};\mathbb{\Sigma};\mathbb{C};\mathbb{A};\mathbb{N} \vdash_{taskset} \mathbb{A}'';\mathbb{N}'';\{\, T_1,\ldots,T_i,\ldots,T_n \,\}}$$

Fig. 7. Typing rules for a parallel task T, and a set of parallel tasks $\mathbb{T}$.

together these fields by writing an indirection pointing to the start of region containing the $(j+1)^{th}$ field at an address one past the end of the j^{th} field. Thus, when all fields of a data constructor are synchronized with, all fields allocated to different regions are linked together by indirection pointers, forming a linked-list.

3.3 Type System

Our type system for LoCal$^{\text{par}}$ requires some substantial extensions to the original type system given by Vollmer et al. [2019]. These extensions address the need to handle multi-task configurations, which require a number of new typing environments and rules. Before we present these extensions, we recall the typing rule for the configuration of a single task, which is mostly unchanged from the original.

$$\Gamma;\Sigma;C;A;N \vdash A';N';e:\hat{\tau}$$

The context for this judgment includes five different environments. First, Γ is a standard typing environment. Second, Σ is a store-typing environment, mapping *materialized* symbolic locations to their types. That is, every location in Σ *has been written* and contains a value of type $\Sigma(l^r)$. Third, C is a constraint environment, keeping track of how symbolic locations relate to each other. Fourth, A maps each region in scope to a location, and is used to symbolically track the allocation and incremental construction of data structures; Finally, N is a nursery of all symbolic locations that have been allocated, but not yet written to. Locations are removed from N upon being written to, as the purpose is to prevent multiple writes to a location. Both A and N are threaded through the typing rules, also occurring in the output of the judgment, to the right of the turnstile.

To generalize our typing rules to handle multi-task configurations, we introduce new environments for variables $\mathbb{\Gamma}$, store typing $\mathbb{\Sigma}$, allocation constraints $\mathbb{C}$, allocation pointers $\mathbb{A}$, and nurseries $\mathbb{N}$. These environments extend their counterparts in the sequential LoCal type system, and are needed to track state on a per-task basis. Figure 7 gives the precise typing rules to type check a parallel task T, and a set of parallel tasks $\mathbb{T}$. A parallel task T is well-typed if its target expression e is well-typed, using the original LoCal typing rules, and a task set $\mathbb{T}$ is well-typed if all tasks in it are well-typed. LoCal$^{\text{par}}$'s complete typing rules are given in the Appendix of the extended version [Koparkar et al. 2021].

3.4 Type Safety

Compared to the original type-safety result proved for single-task LoCal, ours generalizes to parallel evaluation by requiring that, for any given multi-task configuration, either the program has fully evaluated or at least one task can take a step. As usual, we prove this theorem by showing progress

and preservation. The main complication relates to the property that parts of the overall store are now spread across the individual stores of the tasks, whereas in the original proof there is only one store. In particular, our proof must establish that the store of each task remains well formed, even while that task waits on a data dependency, and moreover after the task joins with another task and their stores are merged. The complete proof is available in the Appendix of the extended version [Koparkar et al. 2021]. Here we summarize key invariants.

Many such invariants are specified by our well-formedness rule, which applies to a set of tasks executing in the parallel machine. The full rule is given in the Appendix of the extended version[Koparkar et al. 2021].

$$\mathbb{\Sigma}; \mathbb{C}; \mathbb{A}; \mathbb{N} \vdash_{wf_{tasks}} \mathbb{T}$$

This judgment specifies two new invariants that must hold for all tasks $T \in \mathbb{T}$. The first enforces that all ivars get filled with an appropriate value. In particular, if an expression being evaluated by a task references an ivar, then there must be exactly one other task in the task set which supplies a *well-typed* value for it. The second invariant consists of the well-formedness judgment that verifies certain properties hold for each store of a given task. This judgment generalizes a similar rule used in the original proof by its use of the overall task set $\mathbb{T}$. We discuss in detail the necessary extensions in the sequel.

With these new typing judgments in hand, we can now state the type-safety theorem, shown below. This theorem states that, if a given task set $\mathbb{T}$ is well typed and its overall store is well formed, and if $\mathbb{T}$ makes a transition to some task set $\mathbb{T}'$ in n steps, then either all tasks in $\mathbb{T}'$ are fully evaluated or $\mathbb{T}'$ can take a step to some task set $\mathbb{T}''$.

Theorem 3.1 (Type Safety).

$$\begin{aligned} &\textit{If } \varnothing; \mathbb{\Sigma}; \mathbb{C}; \mathbb{A}; \mathbb{N} \vdash_{taskset} \mathbb{A}'; \mathbb{N}'; \mathbb{T} \ \wedge \ \mathbb{\Sigma}; \mathbb{C}; \mathbb{A}; \mathbb{N} \vdash_{wf_{tasks}} \mathbb{T} \\ &\textit{and } \mathbb{T} \Longrightarrow^{n}_{rp} \mathbb{T}' \\ &\textit{then, either } \forall T \in \mathbb{T}'.\ TaskComplete(T) \\ &\textit{or } \exists\, \mathbb{T}''.\ \mathbb{T}' \Longrightarrow_{rp} \mathbb{T}''. \end{aligned}$$

Well-formedness of the Store. Our store well-formedness judgment extends the judgment of the sequential LoCal typing system to establish a global criterion for well-formedness, checking in particular that parts of regions that are distributed across each task-private store, given by $M; S$, are well-formed:

$$\Sigma; C; A; N; \mathbb{T} \vdash_{wf} M; S$$

This judgment, defined formally in the Appendix of the extended version [Koparkar et al. 2021], is one of the most challenging parts of our extension, because it must be strong enough to ensure safe merging of stores when tasks meet at join points. Like in sequential LoCal, it specifies three categories of invariants.

The first category enforces that allocations occurring across the task-private stores are accounted for. In particular, for each symbolic location in the store-typing environment, $(l^r \mapsto \tau) \in \Sigma$, a value must be allocated to the appropriate store. There are two possible ways in which this allocation may occur: (1) sequentially, in the current task, or (2) in parallel, in a different task. In the sequential case, l^r's address in the location map M must be a concrete index, and it must have an end-witness. This technical point ensures that the store never contains partially allocated values. In the parallel case, l^r's address in M must be an ivar, and there must be exactly one other task, $T_{oth} \in \mathbb{T}$, that supplies a well-typed value for it. Moreover, l^r's address in T_{oth}'s location map must be a concrete index, and if T_{oth} has finished evaluating, this value must have an end-witness. This property ensures that

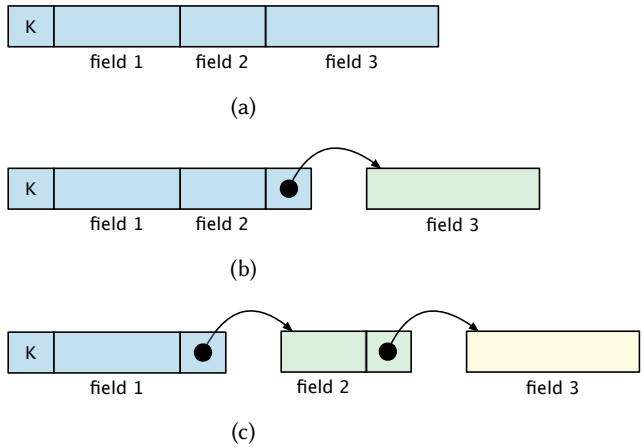


Fig. 8. The heap layout for a data constructor if: (a) all fields are allocated sequentially, (b) only the second and third fields are allocated in parallel with each other, and (c) all fields are allocated in parallel.

when these tasks merge, the resulting store has *complete* values allocated at the expected addresses and the expected types.

The second category enforces that allocations occur in the sequence specified by the constraint environment C. In particular, if there is some location l in the domain of C, then the location map and store must have the expected allocations at the expected types. The most interesting rule here is that for the `after` constraint, since it involves potential parallel allocations. For instance, if $(l \mapsto (\texttt{after}\ \tau@l')) \in C$, then the values at locations l and l' may be allocated sequentially, in the same task, or in parallel, in different tasks. The sequential case is straightforward. For the parallel case, there are two possibilities — (1) the task allocating the value at location l' may be still in-flight, or (2) it may have already synchronized with the current (parent) task. In the first case, we ensure the presence of an appropriate indirection in the location map ($l \mapsto \langle r, \&(r_{\text{fresh}}, 0)\rangle$) and of a fresh region in the store ($r_{\text{fresh}} \in S$). Otherwise, we ensure that a link between the values at locations l' and l exists, which is accomplished by the metafunction *LinkFields*.

The final category enforces that each location is written to only once. This is done by checking that the domain of the store-typing environment and the nursery are disjoint: $dom(\Sigma) \cap N = \varnothing$.

3.5 Controlling Fragmentation

A consequence of LoCal$^{\text{par}}$ introducing fresh regions is that the schedule of evaluation dictates the way a value is laid out on the heap, as shown in Figure 8. Every choice to parallelize an intra-region allocation implies the creation of a new region and a new indirection, thereby introducing fragmentation. Thus, in addition to the usual task-scheduling overheads, in our system, a schedule that parallelizes too many allocations also leads to fragmentation. Conversely, effort at amortizing the overhead of parallelism simultaneously amortizes the overhead of indirections and region fragmentation. We return to this topic and address fragmentation along with parallelism granularity management in Section 4.1.

4 IMPLEMENTATION

Gibbon is a whole-program[3] micropass compiler that compiles a polymorphic, higher-order subset of (strict) Haskell[4]. The Gibbon front-end uses standard whole-program compilation and monomorphization techniques [Chlipala 2015] to lower input programs into a first-order, monomorphic representation. On this representation, Gibbon performs *location inference* to convert it into a LoCal program, which has region and location annotations. Then a big middle section of the compiler is a series of LoCal->LoCal compiler passes that perform various transformations. Finally, it generates C code.

Our parallelism extension operates in the middle end, with minor additions to the backend code generator and the runtime system. We add a collection of LoCal->LoCal compiler passes that transform the program so that reads and allocations can run in parallel. At run time, we make use of the Intel Cilk Plus language extension [Blumofe et al. 1995] (and its work-stealing scheduler) to realize parallel execution. Our implementation closely follows the formal model described in Section 3.2, but with explicit parallelism annotations.

4.1 Granularity Control

Before going further into the details of what we implemented, we first explain our choices in what we do and *do not* implement. As we saw in Figure 1a, we use manual annotations for the programmer to mark parallelism opportunities. This is the norm in both current and past parallel programming practice: from MultiLisp [Halstead 1985] to OpenMP [Menon and Dagum 1998], Cilk [Blumofe et al. 1995], Java fork-join [Lea 2000], etc. Recall also that with a purely functional source language, parallel-tuple annotations change *performance only*, not program semantics, so a programmer need not worry about safety when inserting annotations[5]. Task granularity thresholds can be fine-tuned by using the same reasoning as in other parallel systems — switch to sequential for small problem sizes. But there's also the issue of fragmentation (Section 3.5), i.e. amortizing the overhead of pointers in the representation as well as parallel tasks in the control flow. One might wonder how these interact.

4.1.1 How to Optimize Granularity in Gibbon? Relatively small chunks of sequential data can effectively amortize the cost of creating regions and indirections. For instance, serializing just the bottom two levels in our binary tree examples eliminates 75% of pointers and ensures pointers use only 11% of memory. The task size to amortize parallel scheduling overheads, on the other hand, is usually much larger. Therefore, it's best the Parallel Gibbon programmer thinks about parallelism granularity exclusively, and the data representation can comfortably follow from that. In other words, a manual (or automatic) solution to task granularity, also gives "for free", an efficient, mostly-serialized data representation with amortized indirections.

4.1.2 Why not Automatic Granularity? Automatic task-parallel granularity control is an active research area [Acar et al. 2019, 2018]. Combining Parallel Gibbon's automatic control over data representation, *together* with automatic granularity control, is a promising avenue for future work.

[3]Gibbon's automatic selection of data representation works best if it can see the whole program, much like the data-representation optimizations in MLton. One way to get around this issue would be to make the programmer responsible for choosing the representation, by using appropriate annotations in datatype definitions. Another option is to conservatively insert random-access information in all datatypes that flow into code within other compilation units. Our current implementation does not offer these options, and only supports whole program compilation.

[4]Note that we are not the first to propose a strict variant of Haskell, not only do many of its cousins like Idris take a strict approach, but GHC itself supports a module-level strict mode.

[5]This same property holds for inserting parallel annotations in pure GHC Haskell code, which has been used to modest benefit in past experimental work [Harris and Singh 2007], but is not commonplace practice.

This goes doubly so if approaches like *Heartbeat scheduling* [Acar et al. 2018] mature to the point of offering robust backends and runtime systems that compilers like Parallel Gibbon may target, and very recent work offers a step in that direction [Rainey et al. 2021].

For this paper, however, it would be confounding to address automating granularity *simultaneously* with compacting data representation and assigning regions. In our experiments (Section 5), we hold task granularity constant across different implementations of the same benchmarks, focusing only on the impact of each compiler's code generation and data representation choices. Parallel Gibbon, as well as all of its competitors, use explicit parallelism annotations, and schedule the same set of tasks at runtime for the same program inputs, unless mentioned otherwise.

4.2 Desugaring Parallel Tuples

As shown in Figure 1a, in the front-end language, we use the standard parallel tuples to express parallelism, like other eager, parallel functional languages [Reppy et al. 2009; Westrick et al. 2019]. A parallel tuple `(e1 || e2)` marks the expressions `e1` and `e2` to evaluate in parallel with each other. To more closely match Cilk, we desugar these parallel tuples into a spawn/sync representation in the compiler IR:

```
let x = spawn e1 in
let y = e2 in
let _ = sync in
(x,y)
```

Using this representation simplifies the subsequent conversion to LoCal, in which additional steps like allocating regions or binding locations may be required before getting to `e1` or `e2`. Generating the corresponding `letregion`/`letloc` bindings, such that they have the correct scope, is easier with a spawn/sync representation. Also, we preserve these parallelism annotations in the LoCal code we generate. In contrast with the formal model (with implicit parallelism, Section 3), `let y = e2` is always sequential, whereas `let y = spawn e2` essentially corresponds to a *potentially* parallel let binding, though the decision is ultimately dynamic.

We do not support first-class futures, or tasks that communicate through channels or other mutable data structures, and thus the task-parallelism opportunities available in Parallel Gibbon remain effectively *series-parallel*. But this is sufficiently expressive for writing a large number of parallel algorithms. Note that the formal model can express some *local*, non-escaping futures that are not strictly series-parallel by using parallel lets that are forced out of order. This pattern of parallelism does not provide much additional expressive power over that provided by parallel tuples, so we do not give up much by not exposing this capability in the front-end language.

4.3 Indirection Pointers

In the implementation, we need a runtime representation of optional pointers to include in the data that corresponds to the indirections in the semantics (Section 3.1). Fortunately, in the Gibbon compiler there is already a pointer mechanism that is sufficient for our purposes. This exists because of how Gibbon's regions grow—rather than copying data into a larger buffer, Gibbon accumulates a linked list of contiguous chunks, doubling the size on each extension. The last filled cell in a chunk is an indirection into the next chunk. Indirections also enable Gibbon to allocate a value that is *shared* between multiple locations (within the same region or across regions) without requiring a full copy, which is crucial for ensuring asymptotic complexity conservation of programs. Variable aliases indicate this sharing to the compiler. For example, the aliased variable `x` in the expression `(let x = mkBigExpr in Plus x x)` indicates that a single, shared value must be allocated for the left and right subtrees of `Plus`. Gibbon rewrites this expression to `(let x = mkBigExpr in Plus x (IndPtr x))`,

where the right subtree is an indirection pointing to the data allocated for the left. Similarly, the identity function (`id x = x`) becomes (`id x = IndPtr x`). However, shared scalar values such as numbers and booleans are always copied, because it is more efficient to do so. In Parallel Gibbon, we reuse this indirection pointer mechanism to implement intra-region parallel allocations.

4.4 Parallel Reads

Using static analysis, Gibbon can infer if a datatype requires offsets, and it can transform the program to add offsets to datatypes that need them. In sequential programs, these are used to preserve asymptotic complexity of certain functions. In Parallel Gibbon, we use these offsets to enable parallel reads. We update that static analysis, and add offsets if a program performs parallel reads, i.e. via a clause in a case expression that accesses a data-constructor's fields in parallel.

4.5 Parallel Allocations

The implementation of intra-region parallel allocations closely follows the design described in Section 3. A program transformation pass generates code that allocates fresh regions and writes indirection pointers at appropriate places. But the metafunctions *MergeS* and *MergeM* which merge task memories at join points have a different run time behavior compared to their formal definition. The implementation does not have a direct notion of a store. At run time, a region variable r is only a structure containing a pointer to the start of a memory buffer and some metadata necessary for garbage collection. Two memory buffers are merged (linked) simply by writing a single indirection pointer in one of them. This operation is relatively cheap compared to the set union used in the formal definition. Similarly, the implementation does not have a direct notion of a location map, and therefore there is no run time operation equivalent to *MergeM*. At run time, all location variables become absolute pointers into the heap.

But there still exists an issue with fragmentation. With granularity control — in the form of judicious use of parallel tuples — we can restrict excessive creation of fresh regions, but the number of regions created will still always be equal to the number of parallel tasks spawned by the program. This can still cause fragmentation because all spawned tasks might not actually *run* in parallel.

The key insight is to make the number of fresh region allocations equal to the number of *steals*, not spawns. That is, because our implementation uses a work-stealing scheduler, but the general idea applies to other schedulers as well: fragmentation should be proportional to parallelism in the dynamic schedule, not the static potential for parallelism. Our implementation creates fresh regions for intra-region parallel allocations only if they really run in parallel. We accomplish this by using the Cilk Plus API to implement a hook to detect when steals occur. Before reaching a parallel fork point (spawn), the runtime system stores the ID of the worker executing the current code. Next, the corresponding ID is immediately fetched in the continuation of the fork point. If the IDs match, it indicates that a steal did not occur. This optimization enables parallel allocations with minimal fragmentation.

4.6 Parallel Arrays

Programs need arrays as well as trees. We extend Gibbon with array primitives such as `alloc`, `length`, `nth`, `slice`, and `inplaceUpdate`, and use them to build a small library of parallel array operations with good work and span bounds. The ability to *safely* mutate an array in-place allows us to implement optimizations that go beyond what is commonly allowed in a purely functional language. This is enforced using the new Linear Haskell extensions [Bernardy et al. 2017], for example, the signature of an $O(1)$ array mutation is:

```
inplaceUpdate :: Int → a → Array a ⊸ Array a
```

Using these primitive operations, collective operations on arrays are implemented as recursive divide-and-conquer functions in Parallel Gibbon that use parallelism annotations[6]. For example, our parallel `map` first allocates an array to store the output, and then updates it in parallel with `inplaceUpdate`. But all such potentially-racy operations are hidden behind a pure interface. Also, an `Array` in Parallel Gibbon can only store primitive values such as numbers, booleans, and n-ary tuples of such values. In the future, we plan to explore ways to support data-parallel operations on serialized algebraic data.

4.7 Memory Management

In the formalism, regions are modeled as *unbounded* memory buffers that offer a byte-indexed storage for primitive values (data constructor tags, numbers, etc.). Practically, we start by allocating a single contiguous *chunk* of memory of *bounded* size. When this chunk is exhausted, a new one which is double in size is allocated and linked with the previous one using a pointer. This policy is used up to an upper bound (1GB) after which constant sized (1GB) new chunks are allocated. Thus, a single region is really a linked-list of chunks: a small initial chunk, with subsequent ones doubling in size. The small initial chunk is beneficial when a region contains a small value, and the doubling policy reduces the overall `malloc` overhead when allocating large values.

Our garbage collection strategy is based on regions and lifetimes, and also reference counting. In classic region calculi, regions can be immediately deallocated at the end of their lexical scope. However, we allow indirection pointers to point across *different* regions (chunks), which is crucial to support parallelism, (and also to maintain the asymptotic complexity of certain sequential programs). Thus, a region can stay alive beyond its lexical scope, for example if a pointer to it is captured by another region which is still in scope.

We use reference counting to deallocate such regions. When a region is initialized—with a `letregion`—its reference count is set to 1, and it is decremented when the region goes out of scope. At this stage, if its reference count hits zero, it is deallocated by freeing all of its chunks. When a chunk is freed, the reference counts of the regions it points to are also decremented, which may cause some of these regions to be freed as well. But these reference counts are *per-region*, rather than *per-chunk*. Hence, even if a single chunk in a region is truly alive, all of its other chunks are also considered alive, and cannot be freed. Also, there isn't a supplemental garbage-collector for long-lived regions at this time, as in later versions of MLKit [Tofte et al. 2004]. In the future, we plan to make improvements in this area. Note, however, that it is impossible to create heap cycles in Gibbon, because it's a pure strict language. (In-place modification through linear types doesn't change this, and besides, Parallel Gibbon's arrays do not contain pointers or sum types.)

5 EVALUATION

In this section, we evaluate our implementation using a variety of benchmarks from the existing literature, as well as a new compiler benchmark. To measure the latent overheads of adding parallelism, we compare our single-thread performance against the original, sequential LoCal, as implemented by the Gibbon compiler. Sequential Gibbon is also a good baseline for performing speedup calculations since its programs operate on serialized heaps, and as shown in prior work, are significantly faster than their pointer-based counterparts. Note that prior work [Vollmer et al. 2017] compared sequential constant factor performance against a range of compilers including GCC and Java. Since Sequential Gibbon outperformed those compilers in sequential tree-traversal workloads, we focus here on comparing against Sequential Gibbon for sequential performance.

[6]These combinators offer variants to explicitly control sequential chunk size, or to use the common heuristic of splitting into a number of tasks that is a multiple of the number of cores (provided as a global constant).

We also compare the performance of our implementation to other languages and systems that support efficient parallelism for recursive, functional programs — MaPLe [Westrick et al. 2019], Multicore OCaml [Leroy et al. 2020; Sivaramakrishnan et al. 2020a], and GHC. MaPLe (an extension of MLton) is a whole-program, optimizing compiler for Standard ML [Milner et al. 1997]; it supports nested fork/join parallelism, and generates extremely efficient code.

The experiments in this section are performed on a 48 core machine (96 hyper-threads) made up of 2 × 2.9 GHz 24 core Intel Xeon Platinum 8268 processors, with 1.5TB of memory, and running Ubuntu 18.04. The shared memory on this machine is divided into two NUMA nodes such that CPUs `0-23` and `48-71` use `node-0` as their local memory node, and `24-47` and `72-95` use `node-1`. In our experiments we only use 48 threads (no SMT), evenly distributed across both NUMA nodes (`numactl --physcpubind=48-95`). All experiments are performed using the default memory allocation policy which always allocates memory on the current NUMA node. We observed that using a round-robin memory allocation policy (option `--interleave=0,1`) did not affect performance, and therefore we do not report those results.

Each benchmark sample is the median of 9 runs. To compile the C programs generated by our implementation we use GCC 7.4.0 with all optimizations enabled (option `-O3`), and the Intel Cilk Plus language extension [Blumofe et al. 1995] (option `-fcilkplus`) to realize parallelism. To compile sequential LoCal programs we use the open-source Sequential Gibbon compiler, but we modify it to include arrays with in-place mutation using linear types, just like Parallel Gibbon. For MaPLe, we use version `20200220.150446-g16af66d05` compiled from its source code. For OCaml, we use the Multicore OCaml compiler [Sivaramakrishnan et al. 2020a] (version 4.10 with options `-O3`), along with the `domainslib` [7] library for parallelism. We use GHC 8.6.5, with options `-threaded -O2`, along with the monad-par [Marlow et al. 2011] library for parallelism.

5.1 Benchmarks

We use the following set of 10 benchmarks. For GHC, we use *strict datatypes* in benchmarks, which generally offers the same or better performance, and avoids problematic interactions between laziness and parallelism. All programs use the same algorithms and datatypes (including mutable arrays, which are provably race-free in Gibbon and GHC), have identical granularity control thresholds, and are run with the same inputs. This way, each pairing of program and input creates a deterministic task graph — which does not change when varying the number of threads — and the evaluation focuses on data representation and code generation, rather than on decomposing and scheduling parallel tasks.

- **fib**: Compute the 48th fibonacci number with a sequential cutoff after depth=18: a simple baseline for scaling.
- **buildtreeHvyLf**: This is an artificial benchmark that is included here to measure parallel allocation under ideal conditions. It constructs a balanced binary tree of height 18, and computes the 20th fibonacci number at each leaf, with sequential cutoff after depth=12.
- **buildKdTree** and **countCorrelation** and **allNearest**: `buildKDTree` constructs a kd-tree [Friedman et al. 1977] containing 1M 3-d points in the Plummer distribution. The sequential cutoff is at a node containing less than 32K points. `countCorrelation` takes as input a kd-tree and a list of 100 3-d points, and counts the number of points which are correlated to each one. The chunk-size for the parallel-map is 4, and the sequential cutoff for `countCorrelation` is at a node containing less than 8K points. `allNearest` computes the nearest neighbor of all 1M 3-d points. The chunk-size for the parallel-map is 1024.

[7] https://github.com/ocaml-multicore/domainslib

- **barnesHut**: Use a quad tree to run an nbody simulation over 1M 2-d point-masses distributed uniformly within a square. The chunk-size for the parallel-map is 4096.
- **coins** This benchmark is taken from GHC's NoFib [8] benchmark suite. It is a combinatorial search problem that computes the number of ways in which a certain amount of money can be paid by using the given set of coins. The input set of coins and their quantities are `[(250,55),(100,88),(25,88),(10,99),(5,122),(1,177)]`, and the amount to be paid is 999. The sequential cutoff is after depth=3.
- **countNodes**: This operates on ASTs gathered from the Racket compiler when processing large, real programs. The benchmark simply counts the number of nodes in a tree. For our implementation, we store the ASTs on disk in a serialized format which is read using a single `mmap` call. All others parse the text files before operating on them. To ensure an apples-to-apples comparison, we do not measure the time required to parse the text files. The size of the text file is 1.2G, and that same file when serialized for our implementation is 356M. The AST has around 100M nodes in it. The sequential cutoff is after depth=9.
- **constFold**: Run the `constFold` function shown in Figure 1 on an artificially generated syntax-tree, which is a balanced binary tree of `Plus` expressions, with a `Lit` as a leaf. The height of the syntax-tree is 26, the sequential cutoff is after depth=8.
- **mergeSort**: An in-place parallel merge sort, which bottoms out to a sequential quick sort when the array contains less than 8192 elements. For our implementation, we use the `qsort` function from the C standard library to sort small arrays. The Haskell implementation is taken from Kuper et al's artifact accompanying their paper [Kuper et al. 2014], and it makes an FFI call to a sequential quick sort written in C. MaPLe and OCaml bottom out to a sequential quick sort implemented in their source language. The input array contains 8M randomly generated floating point numbers.

5.2 Results: Parallel Versus Sequential Gibbon

Figures 9a and 9b show the results of comparing performance of benchmarks compiled using our parallel implementation, labeled "Ours", relative to Sequential Gibbon. The quantities in the table can be interpreted as follows. Column T_s shows the run time of a sequential program. which serves the purpose of a sequential baseline. T_1 is the run time of a parallel program on a single thread, and O the percentage overhead relative to T_s, calculated as $((T_1 - T_s)/T_s) * 100$. T_{48} is the run time of a parallel program on 48 threads and S is the speedup relative to T_s, calculated as T_s/T_{48}. R is the number of additional regions created to enable parallel allocations, calculated as $R_{48} - R_s$. For a majority of benchmarks, the overhead is under 3%, and the speedups range between 31.7× and 43.5×. These speedups match, or in cases such as `barnesHut` and `allNearest`, exceed those of optimized implementations that have been analyzed on similar machines [Acar et al. 2018; Shun et al. 2012; Westrick et al. 2019].

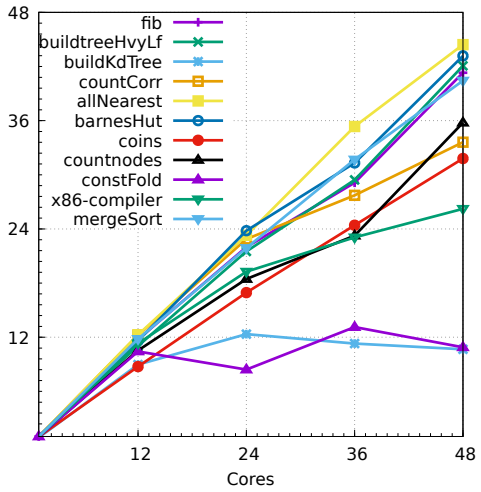
With respect to the difference in speedups between different benchmarks, we see the expected relationship among them which reflects their memory access patterns. Compute-bound benchmarks such as `fib` scale very well, whereas benchmarks such as `constFold` and `buildKdTree` can become memory bound, and do not scale over a certain number of cores[9]. With respect to `buildKdTree`, a significant portion of its total running time is spent in sorting the points at each node. We observed that our `mergeSort` doesn't scale well on small inputs, and since `buildKdTree` performs a series of smaller and smaller sorts, it eventually runs into this, leading to lower scalability. But

[8] https://gitlab.haskell.org/ghc/nofib

[9] For example, a simple parallel dot-product computation (in Cilk) has a similar linear access pattern and low arithmetic intensity to `constFold`, and it achieves only a 6X speedup on this same machine.

	Gibbon	Ours			
Benchmark	T_s	T_1	O	T_{48}	S
fib	12.8	12.8	0	0.31	41.3
buildtreeHvyLf	4.69	4.69	0	0.11	42.6
buildKdTree	2.33	2.67	14.6	0.22	10.6
countCorr	1.46	1.47	0.68	0.044	33.2
allNearest	1.0	1.01	1	0.023	43.5
barnesHut	3.21	3.21	0	0.074	43.4
coins	3.04	3.13	3	0.096	31.7
countnodes	0.21	0.21	0	0.006	35.0
constFold	1.78	1.78	0	0.16	11.1
x86-compiler	1.08	1.08	0	0.041	26.3
mergeSort	1.58	1.60	1.27	0.039	40.5

(a) T_s is the run time of Sequential Gibbon. T_1 and T_{48} are the run times of Parallel Gibbon on 1 thread and on 48 threads respectively. O is the single-thread percentage overhead: $O = (T_1 - T_s)/T_s * 100$. S is the 48-thread speedup: $S = T_s/T_{48}$.



(b) Speedups relative to Sequential Gibbon.

Fig. 9. Parallel Versus Sequential Gibbon.

its high overhead (14.6%) and low speedup (10.6×) are in the same ballpark as an optimized C implementation which Choi et al. [2010] analysed on a 32-core machine. Table 2 and Figure 15 (given in the Appendix of the extended version [Koparkar et al. 2021]) show that MaPLe, GHC and OCaml also scale similarly. Overall, these results show that our technique is able to handle parallelism in a mostly-serialized data representation effectively.

Fragmentation. Traversals on a serialized heap are efficient because serialization minimizes pointer-chasing and maximizes data locality. But the heap produced by running intra-region allocations in parallel is fragmented, which can affect the performance of subsequent traversals that consume this heap, due to additional pointer dereferences and worse locality. To measure this downstream effect, we compare the run time of a single-threaded traversal operating on a sequentially allocated value to that traversal operating on a value allocated in parallel, which will be fragmented. Thus, the thing whose run time is being compared—the traversal—stays the same but it is given inputs that have different levels of fragmentation. We also measure the amount of fragmentation introduced for parallelism by counting the number of regions created solely to enable parallel allocations. Table 1 shows the results.

We use a subset of the benchmarks from Section 5.1 whose output is a serialized value (the other benchmarks do not measure the construction of new values), and measure the time required to *traverse* the output. For example, the benchmark `trav(constFold)` constructs an input expression, runs constant folding over it, and then sequentially traverses the resulting expression (counts the number of leaves in it), but the number reported is only the run time of the traversal and it does not include the time taken to run `constFold` itself. Each benchmark sample is the median of 9 runs, such that each run allocates a value and then traverses it N times, where N is set high enough to

Table 1. T_s is the time required to sequentially traverse a sequentially allocated value. R_{opt} is the number of *additional* regions created to allocate a value in parallel using 48 threads, T_{opt} is the time required to sequentially traverse it, and O_{opt} is the percentage overhead of the traversal: $O_{opt} = (T_{opt} - T_s)/T_s * 100$. R_{max}, T_{max} and O_{max} are the corresponding numbers for a value allocated in parallel using 48 threads with maximum fragmentation. O_{max} is the percentage overhead relative to T_s calculated as $O_{max} = (T_{max} - T_s)/T_s * 100$.

	Seq Alloc.	Optimum Fragmentation			Maximum Fragmentation		
Benchmark	T_s	R_{opt}	T_{opt}	O_{opt}	R_{max}	T_{max}	O_{max}
trav(buildtreeHvyLf)	0.99ms	341	1.02ms	3.03	262K	11.45ms	1056.6
trav(buildKdTree)	6.22ms	31	6.64ms	6.75	262K	55ms	784.2
trav(coins)	0.35s	9K	0.37s	5.71	75M	5.21s	1388.6
trav(constFold)	0.30s	132	0.32s	6.67	67M	2.70s	800
trav(x86-compiler)	366.2μs	1K	377.5μs	3.09	14K	464.3μs	26.8
geomean	-	-	-	4.74	-	-	476.9

get the total run time over one second, and the run time of a *single* traversal is reported. T_s is the time required to sequentially traverse a sequentially allocated value, which is the baseline.

We compare against this baseline the performance of traversing a value allocated in parallel with two levels of fragmentation: *optimum* and *maximum*. In the *optimum* setting, the allocators are identical to those used for measurements reported in Figure 9a. That is, they control fragmentation by controlling the granularity of parallelism (using the thresholds given in Section 5.1) and are compiled using the *region-upon-steal* allocation policy (Section 4.5). R_{opt} is the number of *additional* regions created to allocate a value in parallel using 48 threads. For example, the sequential `constFold` uses a single region and its parallel version requires 132 additional regions (133 regions total). T_{opt} is the time required to sequentially traverse the allocated value, and O_{opt} is the percentage overhead relative to T_s, calculated as $O_{opt} = (T_{opt} - T_s)/T_s * 100$. In the *maximum* setting, the allocators do not control fragmentation at all (no granularity control), and they are compiled using the *region-upon-spawn* allocation policy, which creates a fresh region for every intra-region allocation task that is spawned. This setting thus represents the *upper bound* on the amount of fragmentation that can be introduced due to parallelism, where the heap essentially degenerates to a full pointer-based representation. R_{max}, T_{max} and O_{max} are the corresponding numbers for this fragmentation setting when using 48 threads. O_{max} is the percentage overhead relative to T_s, calculated as $O_{max} = (T_{max} - T_s)/T_s * 100$.

Comparing R_{opt} and R_{max}, we see that in the optimum setting most of the allocated heap is still serialized and uses only a small number of additional regions—less than 0.13% of the maximum— in most cases. For `x86-compiler`, this percentage is higher (7.14%) compared to the other benchmarks because even in the optimum setting this benchmark does not control the granularity of parallelism, which is controlled only by the structure of the input as we discuss in Section 5.4. In this case, the *region-upon-steal* allocation policy is responsible for trimming the number of regions from 14K to 1K. With respect to the run time of the traversal, the overhead with optimum fragmentation is between 3.03% to 6.75%, with a geomean of 4.74%. In addition to fragmentation, the NUMA memory policy [10] has a significant impact on the overall overhead because in the parallel version potentially 50% of memory accesses are at a non-local NUMA memory node. If we run the experiment using a single NUMA node (`numactl --membind=0 --physcpubind=1-24,49-71`), the geomean overhead drops to

[10] As described in the experimental setup, the shared memory on this machine is divided into two NUMA nodes and we use 48 threads evenly distributed across both nodes with the default *local* memory allocation policy.

1.44%, with a significant reduction in the overheads for `buildKdTree` (0.96%) and `constFold` (2.95%). In the presence of maximum fragmentation, we see the expected result: since heap degenerates to a full pointer-based representation, the traversals are several times slower and the benefits of using a serialized representation are lost. This slowdown in traversing a pointer-based representation compared to a serialized one is consistent with the results given in previous work [Vollmer et al. 2019, 2017]. Overall, these results show that using the granularity of parallelism to guide the data representation works well in practice, and gives us an efficient, *mostly-serialized* representation.

5.3 Results: Gibbon Versus Other Compilers

Table 2 shows the results of comparing performance of our implementation to MaPLe, OCaml, and GHC. For each compiler, Column T_s is the run time of a sequential program, Column T_{48} is the run time of a parallel program on 48 threads, and an adjacent column to each shows the corresponding speedup (or slowdown) of our implementation relative to this compiler. For example, on 48 threads, `allNearest` is 3.95× faster with our implementation compared to OCaml. Figure 10 shows how a subset of benchmarks scale on 48 threads. The scaling results for the remaining benchmarks are available in the Appendix of the extended version [Koparkar et al. 2021]. With respect to self-relative comparisons, on average, we scale similarly to MaPLe, and *better* than OCaml or GHC.

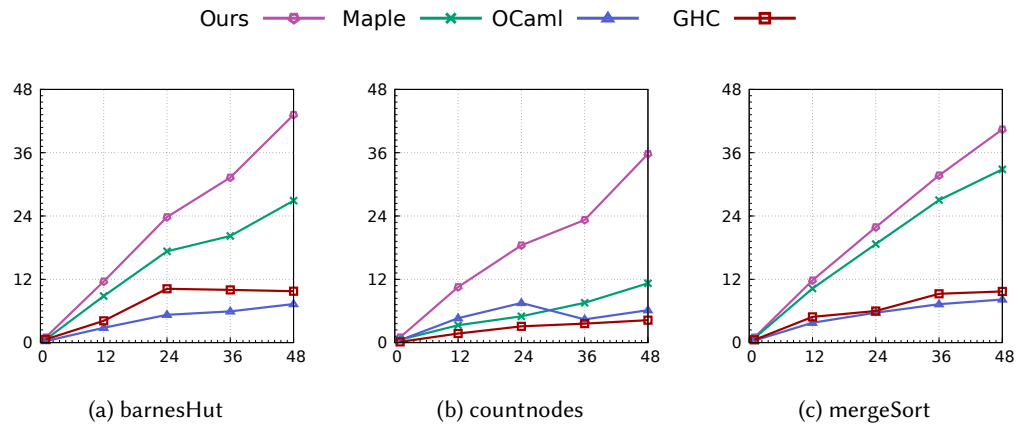
Across all benchmarks, on a single thread our Parallel Gibbon offers a 1.93×, 2.53×, and 2.14× geomean speedup compared to MaPLe, OCaml, and GHC, respectively. When utilizing 48 cores, our geomean speedup is 1.92×, 3.73× and 4.01×. Overall, these results show that we start with a faster baseline in the sequential world, and we're able to preserve the speedups in the parallel as well, meaning that the use of dense representations to improve sequential processing performance *coexists with scalable parallelism*. The only benchmark for which our implementation is slower compared to others is `coins`. This benchmark makes heavy use of linked-list operations such as `cons`, `head`, and `tail`, and our implementation uses `malloc` to allocate memory for every `cons`, which is inefficient. Also, our dense representation currently offers no benefit when building linked-lists by cons'ing onto existing lists. All others, MaPLe, OCaml and GHC, use a copying garbage collector [Marlow et al. 2008; Sivaramakrishnan et al. 2020b; Westrick et al. 2019] allowing them to use a bump-allocator, making them more efficient than our implementation. Figure 15 in the Appendix of the extended version [Koparkar et al. 2021] gives the *self-relative* performance results for MaPLe, OCaml and GHC.

5.4 Results: x86-Compiler Case Study

As an example of a complex benchmark which performs multiple traversals over different datatypes, we implement a subset of a compiler drawn from a university course [Siek et al. 2020]. Our version compiles to x86, from a source language that supports integers and arithmetic and comparison operations on them, booleans and operations such as `and` and `or`, and a conditional expression, `if`. To compile this high level language, we first translate it to an intermediate language similar to C, in which the order of evaluation is explicit in its syntax. The compiler is written in a nanopass style [Sarkar et al. 2004], and is made up of five passes: (1) `typecheck` type checks the source program, (2) `uniqify` freshens all bound variables to handle shadowing, (3) `explicateControl` translates to an intermediate language similar to C, (4) `selectInstructions` generates x86 code which has variables in it, (5) and `assignHomes` maps each variable to a location on the stack. The input to this benchmark is a synthetically generated, balanced syntax-tree with conditional expressions at the top, followed by a sequence of let bindings. The structure of the input program is used to control the granularity of parallelism. The first three passes process the branches of conditionals in parallel, and subsequent ones process every *block* of instructions in parallel.

Table 2. Comparison of Ours with MaPLe, OCaml, and GHC — execution time in seconds, and ratios to Ours. T_s is the run time of a sequential program, and T_{48} is the run time of a parallel program on 48 threads.

	MaPLe				OCaml				GHC			
Benchmark	T_s	$\frac{\text{MaPLe}}{\text{Ours}}s$	T_{48}	$\frac{\text{MaPLe}}{\text{Ours}}48$	T_s	$\frac{\text{OCaml}}{\text{Ours}}s$	T_{48}	$\frac{\text{OCaml}}{\text{Ours}}48$	T_s	$\frac{\text{GHC}}{\text{Ours}}s$	T_{48}	$\frac{\text{GHC}}{\text{Ours}}48$
fib	37.4	2.92	1.06	2.93	21.1	1.65	0.50	1.61	31.9	2.5	0.76	2.45
buildtreeHvyLf	14.5	3.09	0.35	3.18	8.60	1.83	0.25	2.27	12.4	2.64	0.34	3.09
buildKdTree	7.26	3.11	0.41	1.86	10.9	4.68	1.84	8.36	13.4	5.75	2.21	10.0
countCorr	10.5	7.19	0.27	6.14	13.9	9.52	0.37	8.41	3.54	2.42	0.15	3.41
allNearest	2.38	2.38	0.06	2.60	3.01	3.01	0.091	3.95	2.07	2.07	0.068	2.96
barnesHut	5.05	1.57	0.12	1.62	10.9	3.40	0.44	5.94	4.97	1.55	0.33	4.46
coins	1.71	0.56	0.05	0.52	1.05	0.34	0.036	0.37	0.82	0.27	0.085	0.88
countnodes	0.37	1.76	0.019	3.16	0.46	2.19	0.034	5.67	1.45	6.90	0.049	8.16
constFold	2.36	1.32	0.23	1.44	17.7	9.94	2.23	13.9	3.71	2.08	0.64	4.00
x86-compiler	1.34	1.24	0.042	1.02	1.20	1.11	0.09	2.20	2.34	2.16	0.44	10.7
mergeSort	1.74	1.10	0.047	1.20	3.83	2.42	0.19	4.87	2.74	1.73	0.16	4.10
geomean	-	1.93×	-	1.92×	-	2.53×	-	3.73×	-	2.14×	-	4.01×



(a) barnesHut (b) countnodes (c) mergeSort

Fig. 10. Speedups on 1-48 threads relative to the fastest sequential baseline, which is Sequential Gibbon for these three benchmarks.

On this compiler benchmark, Parallel Gibbon offers a 1.24×, 1.11×, and 2.16× speedup on a single thread, and 1.02×, 2.20× and 10.7× speedup when utilizing 48 threads, compared to MaPLe, OCaml, and GHC, respectively. Note that most of the run time and also the self-relative parallel speedup of this benchmark is due to `assignHomes`. The first four passes of the compiler have a linear memory access pattern and low arithmetic intensity, much like `constFold`. The passes inspect the input expression, perform inserts or lookups on shallow environments (which only contain entries for variables that are in scope at a point), and then allocate the output. But `assignHomes` works in a

different way. Since it needs to assign a unique stack location to every variable occurring in an expression, it constructs an environment containing *all* variables in the expression (as opposed to just those in scope at a point), and then performs repeated lookups on it to rewrite variable occurrences with stack location references. This environment is significantly larger than those used by other passes, making `assignHomes` much more work-intensive and suitable for parallel execution.

Here we have only taken the first step towards developing an efficient parallel compiler, and there is ample opportunity for further investigation in this area. In the future, we plan to expand the compiler's source language to include constructs such as top-level function definitions and modules which are a source of parallelism in many real compilers.

6 RELATED WORK

The most closely related work to this paper is, of course, Vollmer et al.'s LoCal [Vollmer et al. 2019], which was summarized in Section 2.1. As discussed there, Vollmer et al.'s treatment only provided sequential semantics, while this paper extends those semantics to incorporate parallelism.

This work, and LoCal, are related to several HPC approaches to serializing recursive trees into flat buffers for efficient traversal [Goldfarb et al. 2013; Makino 1990; Meyerovich et al. 2011]. Notably, these approaches *must* maintain the ability to access the serialized trees in parallel, despite the elimination of pointers internal to the data structure, or risk sacrificing their performance goals. The key distinction that makes enabling parallelism in the HPC setting "easier" than in our setting is that these approaches are application-specific. The serialized layouts are tuned for trees whose structure and size are known prior to serialization, and the applications that consume these trees are specially-written to deal with the application-specific serialization strategies. Hence, offsets are either manually included in the necessary locations, or are not necessary as tree sizes can be inferred from application-specific information.

Work on more general approaches for packing recursive structures into buffers includes Cap'N Proto [Varda 2015], which attempts to unify on-disk and in-memory representations of data structures, and Compact Normal Form (CNF) [Yang et al. 2015]. Neither of these approaches have the same design goals as LoCal and LoCal$^{\text{par}}$: both Cap'N Proto and CNF preserve internal pointers in their representations, eliding the problem of parallel access by invariably paying the cost (in memory consumption and lost spatial locality) of maintaining those pointers. We note that Vollmer et al. showed that LoCal's representations enable faster sequential traversal than either of those two approaches [Vollmer et al. 2019], and Section 5 shows that our approach is comparable in *sequential* performance to LoCal despite also supporting parallelism.

There is a long line of work on flattening and nested data parallelism, where parallel computations over irregular structures are *flattened* to operate over dense structures [Bergstrom et al. 2013; Blelloch 1992; Keller and Chakravarty 1998]. However, these works do not have the same goals as ours. They focus on array data, generating parallel code, and data layouts that promote data parallel access to the elements of the structure, rather than selectively trading off between parallel access to structures and efficient sequential access.

Efficient automatic memory management is a longstanding challenge for parallel functional languages. Recent work has addressed scalable garbage collection by structuring the heap in a hierarchy of heaps, enabling task-private collections [Guatto et al. 2018]. There is work proposing a split-heap collector that can handle a parallel lazy language [Marlow et al. 2009] and a strict one [Sivaramakrishnan et al. 2020a], and there is work on a scalable, concurrent collector [Ueno and Ohori 2016]. A promising new line of work explores hierarchical heaps for parallel functional programs [Raghunathan et al. 2016]. All of these designs focus on a conventional object model for algebraic datatypes that, unlike LoCal$^{\text{par}}$, assume a uniform, boxed representation. In the future,

we plan to investigate how results in these collectors relate to our model, where objects may be laid out in a variety of ways.

7 CONCLUSIONS AND FUTURE WORK

We have shown how a practical form of task parallelism can be reconciled with dense data representations. We demonstrated this result inside a compiler designed to implicitly transform programs to operate on such dense representations. For a set of tree-manipulating programs we considered in Section 5, this experimental system yielded better performance than existing best-in-class compilers.

To build on what we have presented in this paper, we plan to explore automatic granularity control [Acar et al. 2019, 2018]; this would remove the last major source of manual tuning in Gibbon programs, which already automate data layout optimizations. Parallel Gibbon with automatic granularity control would represent the dream of implicitly parallel functional programming with good absolute wall-clock performance. While our current approach supports limited examples of data parallelism–friendly data structures beyond trees, such as dense arrays (Section 4.6), we plan to further generalize our system by adding additional data structures that capture mutable sparse and dense multi-dimensional data. We plan to support limited in-place mutation of densely-encoded algebraic data, by adding primitives based on linear types, which we expect to mesh well with the implicitly parallel functional paradigm. While Parallel Gibbon already out-performs competing parallel, functional approaches, we expect these additional features will both improve programmability (by relieving the programmer of the burden of granularity control) and performance (by supporting more efficient parallel structures and strategies).

ACKNOWLEDGMENTS

This work was supported in part by National Science Foundation awards CCF-1725672, CCF-1725679 and CCF-1919197, as well as by the Engineering and Physical Sciences Research Council award EP/T013516/1. We would like to thank our shepherd, Cyrus Omar, as well as the anonymous reviewers for their suggestions and comments.

REFERENCES

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. http://tensorflow.org/ Software available from tensorflow.org.

Umut A. Acar, Vitaly Aksenov, Arthur Charguéraud, and Mike Rainey. 2019. Provably and Practically Efficient Granularity Control. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming* (Washington, District of Columbia) *(PPoPP '19)*. ACM, New York, NY, USA, 214–228. http://mike-rainey.site/papers/oracle-ppop19-long.pdf

Umut A Acar, Arthur Charguéraud, , Adrien Guatto, Mike Rainey, and Filip Sieczkowski. 2018. Heartbeat Scheduling: Provable Efficiency for Nested Parallelism. (2018). http://mike-rainey.site/papers/heartbeat.pdf

Todd A. Anderson, Hai Liu, Lindsey Kuper, Ehsan Totoni, Jan Vitek, and Tatiana Shpeisman. 2017. Parallelizing Julia with a Non-Invasive DSL. In *31st European Conference on Object-Oriented Programming (ECOOP 2017) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 74)*, Peter Müller (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 4:1–4:29. https://doi.org/10.4230/LIPIcs.ECOOP.2017.4

Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. 1989. I-structures: Data structures for parallel computing. *ACM Trans. Program. Lang. Syst.* 11 (October 1989), 598–632. Issue 4. https://doi.org/10.1145/69558.69562

Lars Bergstrom, Matthew Fluet, Mike Rainey, John Reppy, Stephen Rosen, and Adam Shaw. 2013. Data-Only Flattening for Nested Data Parallelism. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel*

Programming (Shenzhen, China) *(PPoPP '13)*. Association for Computing Machinery, New York, NY, USA, 81–92. https://doi.org/10.1145/2442516.2442525

Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. 2017. Linear Haskell: Practical Linearity in a Higher-Order Polymorphic Language. *Proc. ACM Program. Lang.* 2, POPL, Article 5 (Dec. 2017), 29 pages. https://doi.org/10.1145/3158093

Guy E. Blelloch. 1992. *NESL: A Nested Data-Parallel Language.* Technical Report. USA.

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, USA) *(PPOPP '95)*. Association for Computing Machinery, New York, NY, USA, 207–216. https://doi.org/10.1145/209936.209958

Manuel M.T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. 2011. Accelerating Haskell Array Codes with Multicore GPUs. In *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming* (Austin, Texas, USA) *(DAMP '11)*. Association for Computing Machinery, New York, NY, USA, 3–14. https://doi.org/10.1145/1926354.1926358

Adam Chlipala. 2015. An Optimizing Compiler for a Purely Functional Web-application Language. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming* (Vancouver, BC, Canada) *(ICFP 2015)*. ACM, New York, NY, USA, 10–21. https://doi.org/10.1145/2784731.2784741

Byn Choi, Rakesh Komuravelli, Victor Lu, Hyojin Sung, Robert L. Bocchino, Sarita V. Adve, and John C. Hart. 2010. Parallel SAH K-D Tree Construction. In *Proceedings of the Conference on High Performance Graphics* (Saarbrucken, Germany) *(HPG '10)*. Eurographics Association, Goslar, DEU, 77–86.

Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. 2009. *Semantics engineering with PLT Redex.* Mit Press.

Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel. 1977. An Algorithm for Finding Best Matches in Logarithmic Expected Time. *ACM Trans. Math. Softw.* 3, 3 (Sept. 1977), 209–226. https://doi.org/10.1145/355744.355745

Michael Goldfarb, Youngjoon Jo, and Milind Kulkarni. 2013. General transformations for GPU execution of tree traversals. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (Supercomputing) (SC '13)*.

Adrien Guatto, Sam Westrick, Ram Raghunathan, Umut 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* (Vienna, Austria) *(PPoPP '18)*. Association for Computing Machinery, New York, NY, USA, 81–93. https://doi.org/10.1145/3178487.3178494

Robert H. Halstead. 1985. MULTILISP: A Language for Concurrent Symbolic Computation. *ACM Trans. Program. Lang. Syst.* 7, 4 (Oct. 1985), 501–538. https://doi.org/10.1145/4472.4478

Tim Harris and Satnam Singh. 2007. Feedback Directed Implicit Parallelism. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming* (Freiburg, Germany) *(ICFP '07)*. Association for Computing Machinery, New York, NY, USA, 251–264. https://doi.org/10.1145/1291151.1291192

Gabriele Keller and Manuel M. T. Chakravarty. 1998. Flattening Trees. In *Proceedings of the 4th International Euro-Par Conference on Parallel Processing (Euro-Par '98)*. Springer-Verlag, Berlin, Heidelberg, 709–719.

Chaitanya Koparkar, Mike Rainey, Michael Vollmer, Milind Kulkarni, and Ryan R. Newton. 2021. Efficient Tree-Traversals: Reconciling Parallelism and Dense Data Representations. arXiv:2107.00522 [cs.PL]

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

Doug Lea. 2000. A java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*. 36–43.

Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. 2020. The OCaml system release. (Feb 2020). https://ocaml.org/releases/4.10/htmlman/index.html

Junichiro Makino. 1990. Vectorization of a treecode. *J. Comput. Phys.* 87 (March 1990), 148–160. https://doi.org/10.1016/0021-9991(90)90231-O

Simon Marlow, Tim Harris, Roshan P. James, and Simon Peyton Jones. 2008. Parallel Generational-Copying Garbage Collection with a Block-Structured Heap. In *Proceedings of the 7th International Symposium on Memory Management* (Tucson, AZ, USA) *(ISMM '08)*. Association for Computing Machinery, New York, NY, USA, 11–20. https://doi.org/10.1145/1375634.1375637

Simon Marlow, Ryan Newton, and Simon Peyton Jones. 2011. A Monad for Deterministic Parallelism. *SIGPLAN Not.* 46, 12 (Sept. 2011), 71–82. https://doi.org/10.1145/2096148.2034685

Simon Marlow, Simon Peyton Jones, and Satnam Singh. 2009. Runtime Support for Multicore Haskell. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming* (Edinburgh, Scotland) *(ICFP '09)*. Association for Computing Machinery, New York, NY, USA, 65–78. https://doi.org/10.1145/1596550.1596563

R. Menon and L. Dagum. 1998. OpenMP: An Industry-Standard API for Shared-Memory Programming. *Computing in Science and Engineering* v, 01 (jan 1998), 46–55. https://doi.org/10.1109/99.660313

Leo A. Meyerovich, Todd Mytkowicz, and Wolfram Schulte. 2011. Data Parallel Programming for Irregular Tree Computations, In HotPAR. https://www.microsoft.com/en-us/research/publication/data-parallel-programming-for-irregular-tree-computations/

Robin Milner, Mads Tofte, and David Macqueen. 1997. *The Definition of Standard ML.* MIT Press, Cambridge, MA, USA.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*. 8026–8037.

Ram Raghunathan, Stefan K Muller, Umut A Acar, and Guy Blelloch. 2016. Hierarchical memory management for parallel programs. *ACM SIGPLAN Notices* 51, 9 (2016), 392–406.

Mike Rainey, Kyle A Hale, Ryan Newton, Nikos Hardavellas, Simone Campanoni, Peter Dinda, and Umut Acar. 2021. Task Parallel Assembly Language for Uncompromising Parallelism. (2021).

John Reppy, Claudio V. Russo, and Yingqi Xiao. 2009. Parallel Concurrent ML. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming* (Edinburgh, Scotland) *(ICFP '09)*. Association for Computing Machinery, New York, NY, USA, 257–268. https://doi.org/10.1145/1596550.1596588

Dipanwita Sarkar, Oscar Waddell, and R. Kent Dybvig. 2004. A Nanopass Infrastructure for Compiler Education. In *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming* (Snow Bird, UT, USA) *(ICFP '04)*. Association for Computing Machinery, New York, NY, USA, 201–212. https://doi.org/10.1145/1016850.1016878

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)*. Association for Computing Machinery, New York, NY, USA, 68–70. https://doi.org/10.1145/2312005.2312018

Jeremy Siek, Carl Factora, Andre Kuhlenschmidt, Ryan Newton, Scott Ryan, Cameron Swords, Michael Vitousek, and Michael Vollmer. 2020. Essentials Of Compilation: An Incremental Approach. https://iucompilercourse.github.io/IU-P423-P523-E313-E513-Fall-2020/

KC Sivaramakrishnan, Stephen Dolan, Leo White, Sadiq Jaffer, Tom Kelly, Anmol Sahoo, Sudha Parimala, Atul Dhiman, and Anil Madhavapeddy. 2020a. Retrofitting Parallelism onto OCaml. *Proc. ACM Program. Lang.* 4, ICFP, Article 113, 30 pages. https://doi.org/10.1145/3408995

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

Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. 2004. A Retrospective on Region-Based Memory Management. *Higher Order Symbol. Comput.* 17, 3 (Sept. 2004), 245–265. https://doi.org/10.1023/B:LISP.0000029446.78563.a4

Mads Tofte and Jean-Pierre Talpin. 1997. Region-Based Memory Management. *Inf. Comput.* 132, 2 (Feb. 1997), 109–176. https://doi.org/10.1006/inco.1996.2613

Katsuhiro Ueno and Atsushi Ohori. 2016. A fully concurrent garbage collector for functional programs on multicore processors. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. 421–433.

Kenton Varda. 2015. Cap'n Proto. https://capnproto.org/

Michael Vollmer, Chaitanya Koparkar, Mike Rainey, Laith Sakka, Milind Kulkarni, and Ryan R. Newton. 2019. LoCal: A Language for Programs Operating on Serialized Data. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 48–62. https://doi.org/10.1145/3314221.3314631

Michael Vollmer, Sarah Spall, Buddhika Chamith, Laith Sakka, Chaitanya Koparkar, Milind Kulkarni, Sam Tobin-Hochstadt, and Ryan R. Newton. 2017. Compiling Tree Transforms to Operate on Packed Representations. In *31st European Conference on Object-Oriented Programming (ECOOP 2017) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 74)*, Peter Müller (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 26:1–26:29. https://doi.org/10.4230/LIPIcs.ECOOP.2017.26

Sam Westrick, Rohan Yadav, Matthew Fluet, and Umut A. Acar. 2019. Disentanglement in Nested-Parallel Programs. *Proc. ACM Program. Lang.* 4, POPL, Article 47 (Dec. 2019), 32 pages. https://doi.org/10.1145/3371115

Edward Z. Yang, Giovanni Campagna, Ömer S. Ağacan, Ahmed El-Hassany, Abhishek Kulkarni, and Ryan R. Newton. 2015. Efficient Communication and Collection with Compact Normal Forms. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)*. ACM, New York, NY, USA, 362–374. https://doi.org/10.1145/2784731.2784735