The Dependence Termination Graph for Parallelization

Federico Sossai
Northwestern University
Evanston, Illinois, USA
fede@u.northwestern.edu

David I. August
Princeton University
Princeton, New Jersey, USA
august@princeton.edu

Simone Campanoni*
Northwestern University
Evanston, Illinois, USA
simone.campanoni@northwestern.edu

Abstract

In the multicore era, parallelism is as important as it is difficult. Strong scaling often requires reshaping the layout of data structures to support parallel access, or exploiting algorithm-specific properties. Most of the insights behind these transformations cannot be inferred by automatic parallelizing compilers. As a consequence, parallelization falls to the programmer, becoming laborious and repetitive. What compilers lack is a coarser-grained representation of which dependences can be safely eliminated, and how. Existing compilers cannot capture coarse-grained dependences because they reason only about instruction-level dependences, missing the higher-level properties that programmers routinely exploit during parallelization. We propose the Dependence Termination Graph (DTG), a compiler abstraction that augments dependences with programmer-defined layout transformations to break them. This allows compilers to parallelize operations such as custom reductions and insertions into sets, bags or hash-tables. Programmers define high-level operations and how they can be made independent, while the compiler detects where this information unlocks parallelism. We evaluate our approach on 11 benchmarks from GAPBS, PARSEC, Rodinia, and SPEC CPU 2017. Our compiler outperforms prior work by 3.0×, retaining on average 96.7% of the speedup of manual parallelization.

CCS Concepts: • **Software and its engineering → Compilers**.

Keywords: parallelism, dependences, multithreading, compilers, intermediate representations

1 Introduction

Multicore processors facilitate weak scaling, while strong scaling remains difficult. To relieve programmers of manual parallelization, compiler writers have long sought to build automatic parallelizers, yet these have largely failed outside kernels with strong regularities. The reasons go beyond alias analysis imprecision and persist even given perfect knowledge of all data dependences.

Recent work [45] showed that in order to extract meaningful parallelism from sequential programs, compilers would have to violate the original semantics enforced by the sequential model. In other words, the insights that developers use to expose thread-level parallelism are not inferable by the compiler. Designing new programming languages or parallel Intermediate Representations (IRs) can mitigate this problem but represent disruptive changes in compilers' architecture or complicate sequential optimization [19, 33]. In contrast, our study reveals that a surprisingly small extension of the LLVM IR enables it to host sufficient information for parallelizing compilers to identify and exploit parallelism, overcoming most roadblocks faced by prior work. These capabilities stem from three key observations. **First**, parallelism is typically discovered through dependence analysis. The Program Dependence Graph (PDG), used by all automatic parallelizers to encode dependences, is an *inferred* abstraction limited in that it cannot represent dependences between arbitrary groups of IR instructions. Hence, the compiler cannot reason at the same granularity at which programmers conceive opportunities for parallelism. For example, Figure 1 shows consecutive insertions into a standard vector-based bag. While a developer would know how to parallelize these two insertions, the data dependences between instructions appear as an intractable tangle to the compiler. This overconstrains parallelism by imposing synchronization at a granularity that is both unnecessarily fine and costly. **Second**, making data structures amenable to parallel accesses often requires memory layout transformations. Because of this coupling between parallelism decisions and data layout decisions, a compiler that leaves the layout untouched cannot

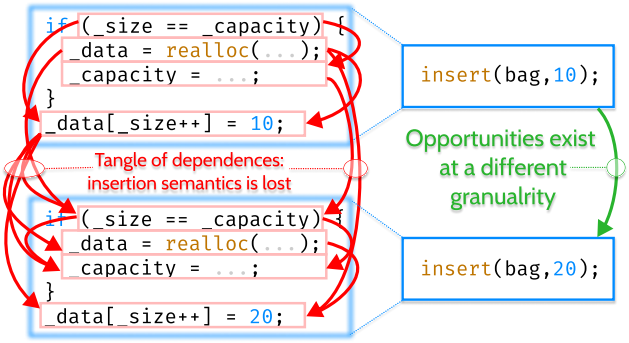

Figure 1. Data dependences (arrows) across two consecutive insertions into a vector-based bag. The granularity at which programmers reason about parallelism does not match the analysis granularity.

*Now at Google.

expose enough parallelism to scale. **Third**, some parallelization strategies rely on semantic properties such as tolerance to stale data accesses or benign races. These properties arise from the algorithm's theoretical aspects, making them impossible to recover via static analysis. While routine for developers, reasoning about these properties is not possible in today's IRs. We propose a unified solution that tackles the three above limitations without disrupting how parallelism decisions are made within existing compiler workflows.

This paper introduces the *Dependence Termination Graph* (DTG), an extended data dependence graph that equips each memory dependence with layout transformations guaranteed to break it when applied to the memory objects underlying that dependence. Unlike the PDG, which relates individual instructions, the DTG relates operations at a coarser granularity: a bag insertion can be represented as a single indivisible unit rather than a tangle of loads and stores. The DTG is defined over a small extension of the LLVM IR, called *T-IR*, that models memory operations and declares when two of them can be made independent via a layout transformation. Our compiler automatically generates T-IR and constructs DTGs by leveraging a C/C++ programming interface based on annotations that this work makes available to developers. Thanks to this interface, developers can now express how a bag's internal representation should be changed to let insertions run in parallel, or when reads and writes to an array can race harmlessly.

This paper makes the following contributions:

1. Introduces T-IR, a small extension of LLVM IR for expressing high-level operations and the layout transformations that make them independent (§ 3.1).
2. Introduces the Dependence Termination Graph (DTG), a compiler abstraction defined on T-IR that augments dependences with recipes to break them (§ 3.2).
3. Describes a programmer interface through which T-IR is generated (§ 4) and an analysis to construct DTGs (§ 5).
4. Evaluates T-800, our DTG-based compiler, on an x86-64 and an AArch64 multicore platform across 11 benchmarks (§ 6).

2 Background and Motivation

This section describes how parallel execution is typically achieved on shared-memory multicore CPUs, and the respective roles of compilers and developers in enabling it. We organize our discussion around what limits automatic parallelization.

2.1 Lessons From Parallelizing Compilers

Motivated by the big performance gains of parallelization, researchers have long sought to build compilers capable of generating multithreaded programs, i.e., parallelizing compilers. A parallelizing compiler is said to be *automatic* (autopar for short) when, given a sequential program, it can identify which instructions *can* and *will* run in parallel and produce a multithreaded program that implements such logic without programmer intervention. An autopar is responsible for not only performing the heavy analyses needed for finding these opportunities but also deciding when they are profitable. The first step of an autopar is finding independent instructions. An analysis finds *data dependences*, that is, pairs of instructions where at least one modifies a register that both access, or where at least one modifies a memory region that both access through their pointers. While the former case can be detected easily, the latter is often computationally intractable due to the difficulty of determining whether two pointers will *alias*, that is, access common memory [21, 29]. There exists another type called control dependences that originate from the control-flow graph, though we do not address them here due to their limited relevance in this work. The results of this step are organized in an abstraction called Program Dependence Graph (PDG) [13], where each node represents an IR instruction and edges encode dependences between them. Once the PDG is constructed, the autopar has all it needs to decide what will run in parallel.

While two instructions that do not depend on each other can always run in parallel, a dependence can often be removed by a transformation that preserves program semantics. Scalar reduction is the canonical example: a loop accumulating into a single variable carries a dependence through that accumulator, yet each thread can accumulate into a private variable and combine the partial results at the end. Moreover, compilers can occasionally detect and exploit reductions for arrays [5, 6, 16, 30, 37, 41]. However, this success is largely confined to highly regular array codes. In many domains such as graph analytics, data compression, and computer vision, more general data structures dominate and these techniques become ineffective. Yet these domains are far from inherently sequential.

2.2 Motivating Example

Figure 2 (left) shows one step of Breadth-First Search (BFS). For each node `n` in `currFrontier`, it scans `n`'s neighbors; any neighbor `m` that has not yet been assigned a parent is given `n` as its parent and added to `nextFrontier`, the frontier processed in the following step. Moreover, even if the same node were inserted into `nextFrontier` more than once, BFS would still produce a correct result. With this in mind, programmers know that iterations of the outermost loop can be parallelized once the insertions into `nextFrontier` are made thread-safe, yet a compiler cannot infer the same opportunity. This limitation is not a consequence of dependence analysis imprecision, as we discuss next.

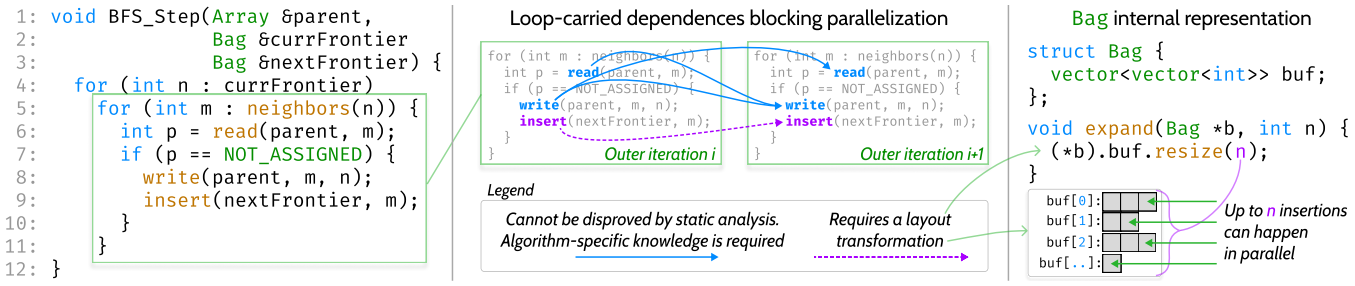

Figure 2. In a BFS step (left), loop-carried dependences arise from `parent` accesses and insertions into the next frontier (center). However, the former kind can be safely relaxed, while the latter kind can be broken by expanding the internal containers of the bag so that parallel insertions do not race (right). These opportunities cannot be inferred by a compiler.

Analysis Granularity is Inadequate. A fundamental disconnect exists between the granularity at which developers conceive memory operations and the granularity of the IR instructions analyzed by the compiler. Developers reason about coarse-grained memory operations such as "append to a sequence" or "keep the minimum between two numbers", which might naturally possess properties like associativity, commutativity, and/or thread safety. However, conventional IRs represent memory as a collection of contiguous regions, each defined by a start and end address. Widely-used containers like sequences or bags are typically implemented by many low-level memory regions and updates are realized by accesses distributed across those regions. Traditional dependence analysis is done at these low-level accesses and the operation semantics is shattered into a myriad of loads, stores, as in Figure 1. This *mismatch* between the natural and analytical granularity renders the recovery of the bag insertion semantics nearly impossible. To be meaningfully analyzable, an IR must let operations be defined at a coarser grain, as whole units rather than the individual instructions that compose them.

Parallelism and Data Layout Are Coupled. In the BFS step of Figure 2, building `nextFrontier` (line 9) is an associative operation. As such, it can be parallelized as a reduction: each thread accumulates its own partial frontier, and these are concatenated once the loop completes. To realize this, the bag of Figure 2 (right) allows insertions to proceed in parallel as long as each occurs on a different element of `buf`. For N parallel insertions, however, `buf` must first be resized to hold at least N `vectors`. In other words, the parallelization is enabled by a *layout transformation* tightly coupled to the number of independent tasks generated. Other data structures impose different transformations. A set, for instance, would require duplicates to be removed after the loop finishes. In general, such transformations depend on both the semantics and the internal representation of the object, and static analysis alone cannot recover them.

Algorithmic Options Are Unknown. At line 8 of Figure 2 (left), different iterations of the outermost loop may update a same entry m of the `parent` array. However, any node from the current frontier is a valid parent for m, so the algorithm remains correct even if two threads race on the same entry. Moreover, BFS is correct even under stale reads to the `parent` array (line 6). These properties stem from an algorithmic understanding that programmers possess but the compiler does not. In fact, a compiler would correctly identify the memory dependences relating reads and writes on `parent` and block the parallelization.

2.3 Putting Everything Together

Complex operations may be associative. Data races may be benign due to algorithm-specific invariants. Updates to a shared container may be thread-safe by construction. The space of the properties helpful for parallelism is vast and we cannot expect compiler analyses to capture all of them or, crucially, to infer the ones for algorithms that have yet to be devised. However, they share a **common denominator**: each property captures whether a given dependence *can* be broken, and *how* to do it.

Our approach builds on this intuition and addresses the three limitations above. We propose T-IR, an extension of LLVM IR equipped with two new constructs. The first models memory operations as indivisible units, with guarantees on the memory regions they affect. The second relates two such operations, declaring when they can run in parallel and the layout transformations that make it possible. We define the DTG, an abstraction that exposes this information to a parallelizing compiler as a new kind of dependence called a *terminable dependence*.

3 The Dependence Termination Graph

In this section, we present the formal definition of the Dependence Termination Graph (DTG) and describe the parallelization opportunities it enables. Because the DTG is defined over our extended LLVM IR, *T-IR*, we start by introducing the concepts on which it relies, then formalize the DTG, and finally describe how the compiler uses it to generate parallelism.

3.1 Two Fundamental Ingredients of T-IR

To let the compiler reason about parallelism at the granularity of high-level operations rather than individual loads and stores, we extend LLVM IR with two concepts: E-functions and Clauses, which together provide indivisible operations affecting memory and rules to make them occur in parallel, respectively. To simplify the presentation, code examples are given in C++.

E-functions: Operations on High-Level Objects. We call a *low-level region* a contiguous range of addresses, the unit in which LLVM IR models memory. We call a *high-level object* an entity realized by one or more low-level regions, all reachable recursively from a single pointer that we call its *base pointer*. An E-function is a function that models an operation on a single high-level object.

Definition 1 (E-function)**.** An E-function is a T-IR program function that takes (i) the base pointer p of a high-level object, and (ii) an integer parameter called *context*, with the following signature:

$$(\texttt{ptr}\ p, \texttt{int}\ ctx, \ldots) \to \tau$$

where any additional parameters may follow and τ is an arbitrary return type. Every valid T-IR program must satisfy the following **constraints** that guard the interaction between E-functions and memory:

1. Every high-level object is either accessed only through p of E-functions, or never accessed by any E-function.
2. No two high-level objects accessed via E-functions share the same low-level memory region.
3. Two E-functions invoked on the same high-level object with distinct contexts are mutually thread-safe.

E-functions are declared and implemented by the programmer via an interface discussed in § 4. For example, a programmer can implement a bag as a high-level object composed of resizable arrays as in Figure 2 (right). An insertion can then be modeled as an E-function that writes to a single internal array as follows:

```
void e_insert(Bag *bag, int ctx, int x) {
  (*bag).buf[ctx].push_back(x);
}
```

Here, the context parameter is the index of the array on which the insertion should occur. What remains unspecified is how to transform an object so that its operations can run on distinct contexts, and when such a transformation is valid. The next construct fills this gap.

Clauses: Transformations for Independence. In the bag example, supporting n parallel insertions requires resizing the bag to contain at least n different arrays before any parallel insertion can begin. The goal of Clauses is to let the compiler know which pairs of E-functions may execute in parallel, and what transformations need to be done to a high-level object to make that parallelism possible.

Definition 2 (Clause)**.** A clause is a binary relation over E-functions that states whether two E-functions may execute in parallel and, if so, which functions transform the layout of the high-level object to make this possible. Formally, a clause is a tuple $(f, g, h_\alpha, h_\omega, k)$ where f, g are two E-functions; h_α and h_ω are functions implementing the layout transformation that would allow f and g to run in parallel; k is a predicate stating whether the clause is commutative, that is, whether exchanging f and g execution order preserves the intended semantics of the high-level object.

The functions h_α and h_ω are called *layout transformations* and have the following signature:

$$(\texttt{ptr}\ p, \texttt{int}\ n) \to \texttt{void}.$$

where p is the base pointer of the high-level object to modify and n is the number of contexts that the resulting layout must support.

Definition 3 (Clause Semantics)**.** Let $(f, g, h_\alpha, h_\omega, k)$ be a clause, let R be a single-entry single-exit region [18], and let $n \geq 1$ be the desired degree of parallelism. Let c_f and c_g be two calls, to f and g respectively, that occur within R and operate on the same base pointer p. The clause states that c_f and c_g may execute in parallel while preserving the semantics that the high-level object at p has under sequential execution, provided that both of the following hold:

1. the calls $h_\alpha(p, n)$ and $h_\omega(p, n)$ are executed at the entry and exit of R, respectively, so that the object at p is reshaped to support n concurrent contexts before any of c_f, c_g runs and restored afterwards; and
2. c_f and c_g are invoked with distinct context values $ctx_f, ctx_g \in \{0, \ldots, n-1\}$. Moreover, if the clause is non-commutative (k = false), the assignment must be monotone in program order: whichever of c_f, c_g comes first in the original sequential execution must receive the smaller context value.

As an example, consider the clause $f = g =$ `e_insert`, $h_\alpha =$ `expand`, $h_\omega =$ `reduce`, $k =$ true and a region containing two consecutive insertions into the same bag:

```
e_insert(&bag, 0, 10);
e_insert(&bag, 0, 20);
```

Thanks to the semantics of the above clause, the compiler is allowed to perform the following transformation:

```
expand(&bag, 2);        // h_alpha
e_insert(&bag, 0, 10);
e_insert(&bag, 1, 20);
reduce(&bag, 2);        // h_omega
```

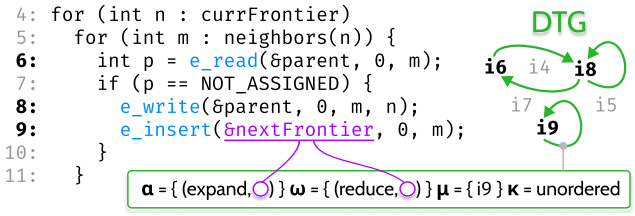

Figure 3. DTG for the BFS step of Figure 2, using C++ for illustration. For clarity, only terminable dependences are shown, and edge attributes are depicted for the dependence (i_9, i_9).

Each insertion now can run in parallel. Because insertions commute (k = true), it is irrelevant which insertion takes context 0 and which takes 1: the bag's contents after `reduce` are the same either way. It is the programmer's responsibility to declare clauses and ensure that their semantics is correct via an interface that we present in § 4.

3.2 DTG Definition

For a given program in T-IR, let $\mathcal{F}$ be the set of its functions and let $\mathcal{F}_E \subseteq \mathcal{F}$ be the set of E-functions. The *Dependence Termination Graph* (DTG) for a loop L is a directed graph

$$\mathcal{G}_L = (V, E, E_T, \alpha, \omega, \mu, \kappa)$$

with edges $E \subseteq V \times V$, where

- V is a set of vertices, each one of them representing an instruction in L,
- $E \subseteq V \times V$ is the set of edges, each edge representing a data dependence between two instructions; as usual, data dependences comprise register and memory dependences.
- $E_T \subseteq E$ is the subset of edges that we call *terminable dependences*, i.e., memory dependences for which there exist a set developer-encoded transformations to break such a dependence,
- α, ω, μ and κ are edge attributes that represent, for each terminable dependence e, the information needed by the compiler to transform the code such that the dependence e no longer exists.

Edge Attributes α and ω. For an edge $e \in E_T$, $\alpha(e)$ and $\omega(e)$ are sets of pairs (f, p), where f is a layout transformation function and p is a pointer to the high-level object on which it must be applied. To break the dependence e, the compiler inserts a call to f for each pair in $\alpha(e)$ just before entering the loop and for each pair in $\omega(e)$ just after leaving the loop. Either set may be empty when no transformation is needed. We describe how α and ω are recovered from clauses in § 5.

Edge Attribute μ. For an edge $e \in E_T$, $\mu(e)$ indicates a set of E-function calls that access common memory and induce the dependence e. This information is needed because eliminating the dependence requires the compiler to reassign the context arguments of those E-function calls so that their writes are redirected to distinct low-level memory regions.

Edge Attribute κ. For a terminable dependence $e = (i, j) \in E_T$, $\kappa(e)$ indicates whether e is *ordered* ($\kappa(e)$ = true) or *unordered* ($\kappa(e)$ = false). This classification has implications for parallel scheduling, as discussed in § 3.4.

Figure 3 shows a (simplified) instance of the DTG for the BFS step of Figure 2, once its memory accesses are modeled with E-functions.

3.3 Generating Parallelism

We now show how our compiler turns the information encoded in the DTG into actual parallelism, through a procedure we call *Termination* (Algorithm 1). Given a loop L and a target number of tasks $nTasks$, Termination attempts to extract iteration-level parallelism and either produces a parallelized IR or reports failure. Termination assumes that the loop has a well-defined preheader and a unique exit basic block. To ensure that the compiler can partition the iteration space and generate independent tasks, we restrict our target loops to *counted loops*, defined as follows.

Definition 4 (Counted Loop)**.** A loop is *counted* if:

1. the trip count T is computable before execution begins and remains fixed for the duration of the loop, and
2. it is indexed by an induction variable i ranging over $[0, T)$.

The procedure unfolds in three steps. It first analyzes the loop to determine whether its dependences can be broken, then prepares the data layout required to break them, and finally rewrites the loop into parallel tasks relying on runtime support.

Step 1: Loop Analysis. The algorithm computes the DTG of L and inspects its loop-carried dependences D, ignoring those induced by the induction variable as they are handled by partitioning the iteration space. Parallelization is admissible only when *every* dependence in D is terminable; a single non-terminable $e \notin E_T$ causes failure. For the terminable ones it gathers the attributes needed to break them (α, ω, μ) and tracks ordering: a single ordered dependence ($\kappa(e)$ true) makes the whole loop ordered, which later constrains scheduling.

Step 2: Layout Transformation. The algorithm materializes the gathered layout changes, inserting each $(f, p) \in A$ as a call $f(p, nTasks)$ in the preheader and each $(f, p) \in \Omega$ symmetrically at the loop exit. Either set may be empty when a dependence can be terminated without any layout change, for example when the operations involved are naturally thread-safe.

Step 3: Loop Transformation. With the layout in place, the algorithm outlines the loop body into a task function f_L

Algorithm 1 Loop Parallelization via Termination

Input: Counted loop L, number of tasks $nTasks$.
Output: Parallelized IR or failure.

$\mathcal{G}_L = (V, E, E_T, \alpha, \omega, \mu, \kappa) \leftarrow$ compute DTG for L.
$A \leftarrow \emptyset$, $\Omega \leftarrow \emptyset$, $M \leftarrow \emptyset$.
$unordered \leftarrow$ true.
▷ Step 1: Loop Analysis
$D \leftarrow$ computeLoopCarriedDataDeps(L).
for each $e \in D$ **do**
  if $e \in E_T$ **then** // is e terminable?
    // *gather layout transformation*
    $A \leftarrow A \cup \{\alpha(e)\}$.
    $\Omega \leftarrow \Omega \cup \{\omega(e)\}$.
    // *gather instructions to be recontextualized*
    $M \leftarrow M \cup \mu(e)$
    if $\kappa(e) =$ true **then** // is e ordered?
      $unordered \leftarrow$ false.
    end if
  else
    return failure // L *cannot be parallelized*
  end if
end for
▷ Step 2: Layout Transformation
for $(f, p) \in A$ **do**
  $P \leftarrow$ lastValidInsertionPoint(Preheader(L)).
  Insert the call @f($p, nTasks$) at P.
end for
for $(f, p) \in \Omega$ **do**
  $P \leftarrow$ firstValidInsertionPoint(Exit(L)).
  Insert the call @f($p, nTasks$) at P.
end for
▷ Step 3: Loop Transformation
$f_L \leftarrow$ outlineLoopToParameterizedTask(L, $nTasks$).
recontextualize(f_L, M).
$c \leftarrow$ create call @dispatcher($f_L, nTasks, unordered$).
Replace L with c.
return success

parameterized by an iteration range and a context parameter c, *recontextualizes* the E-function calls in M (next paragraph), and replaces L with a call to the runtime dispatcher. The *unordered* flag from Step 1 is passed to the dispatcher, which uses it to restrict the admissible schedules.

Recontextualization. Recontextualization (Algorithm 2) rewires the context argument of each such call so that, instead of reading its original value, it reads the f_L's context parameter c. When call instructions in M are not located directly in f_L but are nested within functions that it invokes, the algorithm specializes each call chain from f_L to every reachable E-function call in M. It does so by cloning the functions along the chain and adding the context parameter c to each clone. Each clone then forwards c to its callees, while the original functions remain unchanged for the rest of the program. Because every recontextualized call in f_L now reads the same c, the runtime can control all high-level memory operations in f_L through a single parameter.

Algorithm 2 Recontextualization of E-function calls

Input: Loop task $f_L(start, end, c)$, where c is the context parameter; set M of E-function calls.
Output: Every call in M rewired so its context is set to c.

for $i \in M$ **do**
  if $i \in f_L$ **then**
    i.setArgOperand($1, c$).
  else
    $j \leftarrow$ specializeCallChain(f_L, i, c).
    j.setArgOperand($1, c$).
  end if
end for

Algorithm 3 Runtime Dispatcher

Input: Loop task $f_L(start, end, ctx)$, trip count T, target parallelism $nTasks$, predicate $unordered$.

// *no more contexts than iterations*
$m \leftarrow \min(nTasks, T)$.
if $unordered$ **then**
  $policy \leftarrow$ getPreferredSchedPolicy().
else
  // *fall back on blocked scheduling*
  $policy \leftarrow$ BLOCKED.
end if
Spawn workers $w_0, \ldots, w_{m-1}$. Each worker w_i runs f_L with i as the context argument, while $policy$ determines which iterations w_i executes.
Wait for all workers to finish.
return

3.4 Runtime and Admissible Scheduling

A terminable dependence is *ordered* precisely when it arises from a non-commutative clause ($k =$ false). During Termination, a loop is classified as *ordered* if any of its terminable dependences is ordered, and *unordered* otherwise. The two classes differ in how iterations may be mapped to the contexts $\{0, \ldots, m-1\}$ used at runtime.

For **unordered** loops no clause imposes any ordering, so the mapping is unconstrained: any assignment of iterations to contexts is admissible. For **ordered** loops, the mapping must satisfy the *interval constraint*: each context covers a contiguous interval of iterations. Formally, if iterations i_1 and i_2 are assigned to contexts ctx_1 and ctx_2, respectively, then

$$i_1 < i_2 \implies ctx_1 \leq ctx_2. \tag{1}$$

This is exactly the loop-level counterpart of the monotonicity condition of the clause semantics: an earlier iteration never receives a larger context, so that context indices follow program order.

Our runtime uses a task dispatcher, shown in Algorithm 3, to select a scheduling policy based on the *unordered* predicate. The dispatcher receives the outlined loop body f_L, parameterized by an iteration range $[start, end)$ and a context ctx, together with the target degree of parallelism $nTasks$ and the *unordered* predicate. Unordered loops admit any policy, whereas ordered loops require *blocked* scheduling: the iteration range is split into m contiguous blocks of roughly equal size and context i runs the i-th block, so earlier iterations always map to lower contexts as the interval constraint demands.

4 Programmer Interface

Programmers express E-functions and clauses through a lightweight set of C/C++ annotations that map directly to the T-IR constructs of § 3. The interface consists of two directives.

```
#pragma name(str) [efunc]
{ function-definition }
#pragma clause(str,str) [pre(str)] \
        [post(str)] [comm]
```

`#pragma name(str)` precedes a function definition and assigns it the custom name `str`, which clauses use to refer to it instead of its mangled C++ name. The optional `efunc` qualifier marks the function as an E-function.

`#pragma clause(str`$_1$`, str`$_2$`)` declares a clause (f, g) between the E-functions named `str`$_1$ and `str`$_2$. The optional `pre(str`$_3$`)` and `post(str`$_4$`)` set the layout transformations h_α and h_ω (null if omitted), and `comm` marks the clause as commutative. As an example, the following code expresses the bag insertion through these annotations.

```
1  #pragma name(ins) efunc
2  void e_insert(Bag *bag, int ctx, int x)
3  { ... }
4
5  #pragma name(exp)
6  void expand(Bag *bag, int n) { ... }
7
8  #pragma name(red)
9  void reduce(Bag *bag, int n) { ... }
10
11 #pragma clause(ins, ins) \
12         pre(exp) post(red) comm
```

5 Constructing DTGs

In this section we describe how the DTG can be constructed via static analysis on T-IR. Let $\mathcal{P}$ be a program in T-IR composed of values $\mathcal{V}$, pointer values $\mathcal{V}_{ptr} \subseteq \mathcal{V}$, functions $\mathcal{F}$, and instructions $\mathcal{I}$. We denote the set of E-functions as $\mathcal{F}_E \subseteq \mathcal{F}$ and the set of clauses as $C \subseteq \mathcal{F}_E \times \mathcal{F}_E$, with helpers $h_\alpha(f, g)$ and $h_\omega(f, g)$ that retrieve the layout transformations of a clause $(f, g) \in C$. Like Mod/Ref-based dependence analysis, we summarize the memory effects of each instruction, but in terms of E-functions rather than reads and writes. Crucially, we tag each effect with the *pointer* through which it acts, that is, the object to which a layout transformation must later be applied. An effect is thus an element of $\mathcal{E} := \mathcal{F}_E \cup \{\bot\}$, where $\bot$ conservatively models any may-write not attributable to an E-function. The analysis computes two summaries, $\mathcal{S} : \mathcal{I} \to 2^{\mathcal{E} \times \mathcal{V}_{ptr}}$ and $\mathcal{T} : \mathcal{I} \times \mathcal{F} \to 2^{\mathcal{E} \times \mathcal{V}_{ptr}}$, both initialized to $\emptyset$:

$$\mathcal{S}(i) := \begin{cases} \bigcup_{f \in \text{Callees}(i)} \mathcal{T}(i, f) & \text{if } i \text{ is a call,} \\ \{\bot\} \times \text{Wr}(i) & \text{otherwise,} \end{cases} \tag{2}$$

$$\mathcal{T}(i, f) := \begin{cases} \{(f,\ \text{arg}_0(i))\} & \text{if } f \in \mathcal{F}_E, \\ \bigcup_{j \in f} \sigma_i(\mathcal{S}(j)) & \text{otherwise.} \end{cases} \tag{3}$$

where $\text{Callees}(i)$ over-approximates the functions that i may call, $\text{arg}_0(i)$ is the object-pointer argument of the E-function call i, and $\text{Wr}(i) \subseteq \mathcal{V}_{ptr}$ the pointers i may write through. The operator σ_i reconciles scopes across the call at i: the pointers in a callee summary $\mathcal{S}(j)$ are expressed in the callee's frame, so σ_i rewrites each one by substituting the callee's formal parameters with the actual arguments passed at i. Pointers with no caller-side counterpart, such as callee-local allocations, are discarded, since they cannot be named in the caller's scope. Two effects collide when their pointers *may alias*, written $\text{alias}(p, q)$, that is, when p and q may point to common memory.

Equations 2 and 3 define a monotone may-analysis over the semilattice $(2^{\mathcal{E} \times \mathcal{V}_{ptr}}, \cup)$ and therefore have a least fixed-point solution. Given a loop L, the edges E of its DTG are computed via any traditional dependence analysis. To define the remaining attributes, let $\Pi(i, j)$ be the effect pairs of i and j that may collide on a common object, each tagged with the pointer p where the effect occurs:

$$\Pi(i, j) := \{(f, g, p) \mid (f, p) \in \mathcal{S}(i),\ (g, q) \in \mathcal{S}(j),\ \text{alias}(p, q)\}.$$

A dependence $(i, j) \in E$ is promoted to terminable as follows:

$$(i, j) \in E_T \iff \forall (f, g, p) \in \Pi(i, j) : (f, g) \in C \wedge \text{inv}_L(p), \tag{4}$$

where $\text{inv}_L(p)$ holds when the SSA value producing p dominates L and has no definition inside it. For every $(i, j) \in E_T$, the attributes α and ω are:

$$\alpha(i, j) := \{(h_\alpha(f, g),\ p) \mid (f, g, p) \in \Pi(i, j)\},$$

$$\omega(i, j) := \{(h_\omega(f, g),\ p) \mid (f, g, p) \in \Pi(i, j)\}.$$

By Equation 4, each such p is loop-invariant and therefore available at the preheader as a valid target for its transformation. Each E-function effect is generated at a single call site,

namely the instruction i in the E-function case of Equation 3, which the analysis records alongside the effect; $\mu(i, j)$ is the set of these sites for the effects in $\Pi(i, j)$. Finally, $\kappa(i, j)$ is false if all clauses contributing to Equation 4 are commutative, and true otherwise.

6 Evaluation

Our evaluation answers the following research questions:

- **RQ1: Performance.** Can T-800 generate speedup on par with manual parallelization?
- **RQ2: Memory.** What is the impact of T-800 on memory usage compared to the manual approach?
- **RQ3: Layout.** Are layout transformations beyond array privatization necessary for high performance?
- **RQ4: Generality.** How much do properties expressible via clauses generalize across benchmarks?
- **RQ5: Related Work.** How does T-800 compare against prior automatic and semi-automatic parallelization approaches?

Implementation. We implemented T-IR in a compiler called T-800 on top of LLVM 14. T-800 lowers the pragma annotations describing E-functions and Clauses to LLVM function-level and module-level metadata, respectively. The IR is then canonicalized to guarantee the existence of a unique preheader per loop. Dependence analysis is performed using NOELLE [23] which relies on SVF [36] and other memory analyses to improve dependence precision [2]. To reduce engineering effort, the analysis presented in § 5 has been implemented intraprocedurally by inlining function calls that participate in loop-carried dependences. Our runtime relies on OpenMP to distribute loop iteration chunks across worker threads.

Experimental Setup. We evaluate T-800 on two shared-memory multicore systems, summarized in Table 1. The Intel machine is our primary evaluation platform. We additionally use the Arm machine to study whether the observed performance trends hold across a different ISA, and core count. Frequency boost has been disabled to reduce run-to-run variability. In all experiments, threads are pinned to distinct physical cores to avoid thread migration. We use jemalloc [11] as it yields lower run-to-run variance. For all metrics we report the median of 5 runs. We run PARSEC and SPEC benchmarks with their *native* and *ref* inputs, respectively, and use the *twitter* for BC, *urand* for BFS, and *kronecker* inputs for the remaining GAPBS benchmarks. Kmeans is run on a dataset of 10^5 128-dimensional points and 16 clusters.

All speedups are relative to the fastest single-threaded execution. Measurements are taken over each benchmark's region of interest (ROI). Input loading and output verification are excluded where applicable. Details of each loop parallelized by T-800 are reported in Table 4. Since the selection of optimal loop scheduling parameters is orthogonal to the goals of this paper, T-800 binaries that contain unordered loops are evaluated under a small set of scheduling parameters and the best is reported.

Table 1. Evaluation platforms. L1/L2 sizes are per core.

Property	Intel	Arm
Processor	Xeon Gold 6258R	Ampere Altra Max M128-30
Microarch.	Cascade Lake-SP	Neoverse N1
ISA	x86-64	AArch64
Cores	28	128
SMT	disabled	not available
Frequency	2.70 GHz	3.00 GHz
Freq. boost	disabled	not available
NUMA	1	1
Line size	64 B	64 B
L1 I/D	32 / 32 KiB	64 / 64 KiB
L2	1 MiB	1 MiB
LLC (shared)	38.5 MiB (L3)	16 MiB (SLC)
OS	RHEL 8.10	Ubuntu 22.04.5 LTS

Benchmark Selection. We restrict the evaluation to benchmarks with a publicly available hand-written OpenMP or pthreads implementation, so that compiler-generated parallelization can be compared directly against an expert implementation. Our benchmark set is designed to include parallelization techniques that are difficult or impossible for compilers to recover from sequential code. We include GAPBS graph algorithms [4] due to their irregular memory accesses and work imbalances. From Rodinia, we include kmeans, which is particularly relevant to our setting because it makes use of a 2D array. We include *streamcluster* (SC), *canneal* (CN) and *bodytrack* from PARSEC3 [44] and XZ from SPEC CPU 2017 [7]. For PARSEC and SPEC, benchmarks beyond those reported are omitted due to limited engineering effort. Additional characterization is provided in Appendix A, which reports the algorithmic insight and mechanism used for each benchmark, and Appendix B, which details every loop parallelized by T-800.

6.1 RQ1: T-800 Against Manual Parallelization

On the Intel system, binaries generated by T-800 closely track the scalability of the manual parallelization, as shown by Figure 4. On BC and BFS, T-800 achieves 94% and 91%, respectively, of the manual version. This gap is due to our bag implementation. Looking up an element by index requires scanning per-thread bucket sizes to find which bucket it falls in, making each access $O(P)$. Since BC and BFS stress this operation, the cost compounds. The 8% gain of KM on Arm stems from the spatial locality of the 2D-array layout we implemented.

Arm experiments (Figure 5) highlight a key advantage of compiler parallelization. Exposing parallelism and realizing

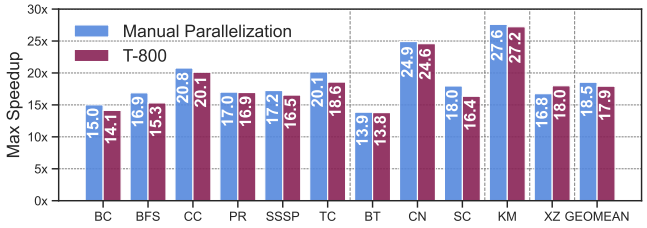

Figure 4. Intel platform - Maximum speedup over 28 cores.

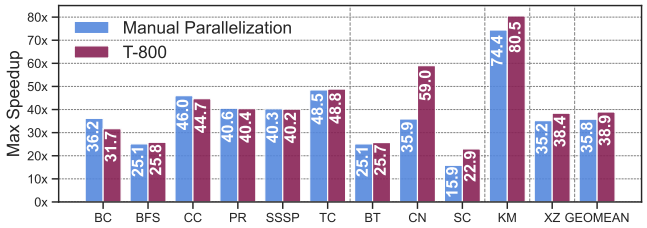

Figure 5. Arm platform - Maximum speedup over 128 cores.

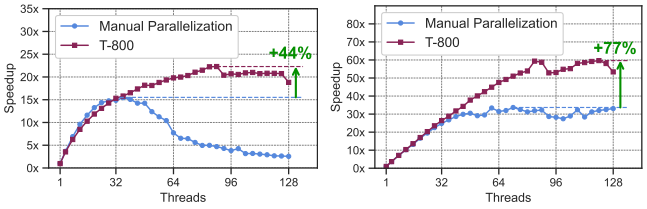

Figure 6. SC (left) and CN (right) on the Arm machine. T-800 uses synchronization mechanisms optimized for the target architecture, while the manual implementations are limited by the primitives they adopt.

it efficiently across architectures are distinct problems. Figure 6 illustrates this point. In both cases, the hand-written parallel implementation commits early to concrete synchronization mechanisms that prove suboptimal on a different machine. In CN, barriers that are adequate on the 28-core Intel system become a bottleneck on the 128-core Arm system. These results suggest that programmers should specify the semantic conditions that make parallel execution legal, i.e., clauses, while the compiler remains responsible for selecting the architecture-specific synchronization mechanisms that realize it.

6.2 RQ2: Peak Memory Usage

Because our approach exposes parallelism through data-layout transformations, we evaluate peak RSS to determine whether any memory overhead is a fundamental limitation of the approach or simply a consequence of how the transformations are implemented. To determine if our approach incurs additional memory overheads due to layout transformations, we measure the peak Resident Set Size (RSS) of

Peak RSS (Intel)

	Manual	T-800	Increase
BC	13.1G	13.2G	+0.5%
BFS	18.0G	18.0G	+0.0%
CC	35.6G	35.6G	+0.0%
PR	17.8G	17.8G	+0.0%
SSSP	33.9G	33.9G	-0.2%
TC	4.7G	4.7G	+0.0%
BT	23M	28M	+22.1%
CN	785M	783M	-0.2%
SC	126M	128M	+1.8%
KM	173M	172M	-0.6%
XZ	4.1G	4.1G	+0.0%

Peak RSS (Arm)

	Manual	T-800	Increase
BC	13.1G	13.2G	+0.5%
BFS	18.0G	18.0G	+0.0%
CC	35.6G	35.6G	+0.0%
PR	17.7G	17.7G	+0.0%
SSSP	33.8G	33.7G	-0.1%
TC	4.6G	4.6G	+0.0%
BT	28M	36M	+29.4%
CN	775M	772M	-0.3%
SC	112M	124M	+10.4%
KM	163M	162M	-0.5%
XZ	4.1G	4.6G	+12.0%

Figure 7. Peak resident set size (RSS) comparison.

each benchmark. To limit the effect of jemalloc's retention of dirty pages in each arena, we constrain the execution to a single arena. Otherwise, pages freed in one arena may not be reused by another, inflating aggregate resident memory. Figure 7 compares the peak RSS of each benchmark for the manual implementation and for the binaries produced by T-800.

The higher RSS in BT reflects a fundamental aspect of how T-800 uses the DTG to generate parallelism. When parallelizing a loop, the Termination algorithm inserts layout transformations in the loop preheader and exit. As a result, frequently invoked loops may repeatedly allocate and free memory, increasing peak RSS. This effect is amplified by allocator behavior, since freed pages are often retained rather than returned immediately to the OS. The loop parallelized by T-800 in BT is invoked 1305 times and collects particles through a bag reduction. Repeated bag relayout, i.e., allocation and reduction of private copies, inflate the observed peak RSS.

XZ groups input chunks into batches and repeatedly serializes the compressed output after each batch. Our implementation instead accumulates the compressed output of all chunks in a `Sequence` container and serializes only once at the end. This increases peak RSS by 12.0%, but also improves speedup by 9.0% on the Arm system.

6.3 RQ3: The Necessity of Layout Transformations

Several benchmarks in our suite rely on collections that are parallelized through layout transformations. As summarized in Table 2, these include bags in BT, BC, and BFS, and a multimap in SSSP. A natural question is whether comparable parallelism could be obtained without such transformations, using locks or other synchronization mechanisms instead. One option is to rely on libraries that provide thread-safe operations, such as TBB concurrent containers [31]. To evaluate this alternative, we implemented variants of our bag and multimap as thin wrappers over TBB containers. The

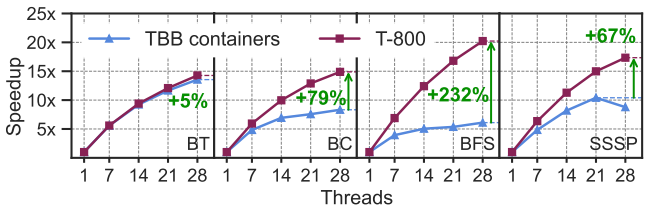

Figure 8. Intel - T-800 vs concurrent collections.

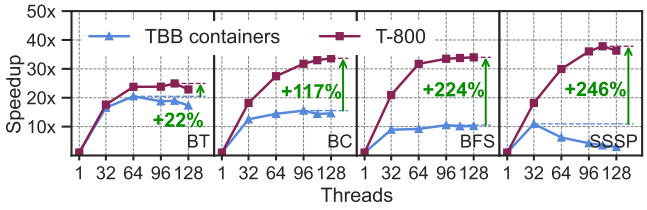

Figure 9. Arm - T-800 vs concurrent collections.

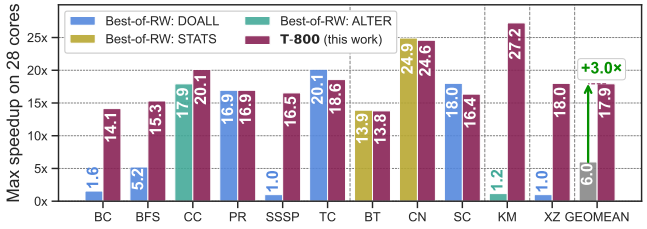

Figure 10. Comparison to three main approaches from related work. For each benchmark, the left bar is the highest speedup achieved among the three related, colored to indicate which baseline produced it.

bag is implemented with `concurrent_vector`, using the thread-safe `push_back` for insertion. The multimap is implemented with `concurrent_hash_map`, with STL `vector` values. In our benchmarks, BT, BC, and BFS rely on bags, while SSSP relies on the multimap. Figure 8 (Intel) and Figure 9 (Arm) compare the strong scaling of the T-800 and the TBB approach. As thread count increases, concurrent containers experience substantial contention, which severely limits scalability. In SSSP, performance deteriorates as thread count increases. Excessive lock contention causes the TBB version to spend 83.6% of CPU cycles stalled on the backend, compared to 21.8% for T-800. This is because the map makes insertions thread-safe by acquiring a per-bucket lock when linking a new node.

Overall, concurrent containers provide meaningful speedup at low core counts, but layout transformations are necessary to sustain scaling at higher core counts, unlocking up to 232% additional performance on Intel and 246% on Arm.

Table 2. Clauses allow programmers to implement classes that can be reused across different benchmarks to enable parallelism.

Class	BC	BFS	CC	SSSP	TC	PR	BT	SC	CN	KM	XZ
`Scalar`	✓	✓			✓	✓	✓	✓	✓	✓	
`Array{1,2}D`	✓							✓		✓	
`RelaxedArray`	✓	✓	✓	✓		✓					
`Bag/Sequence`	✓	✓					✓				✓
`Multimap`				✓							
`Set`	✓	✓									
`PrivateObject`							✓		✓		
`RandNumGen`							✓		✓		

6.4 RQ4: Generality of Clauses

To port each benchmark so that the properties enabling its parallelism are encoded via clauses we implemented a small set of reusable classes. As Table 2 shows, the same classes recur across benchmarks and even across suites, so that porting a new benchmark largely reduces to assembling clauses we had already defined. `Bag` and `Sequence` appear as a single entry in the table because they share the same implementation, while exposing a different interface. We now describe, benchmark by benchmark, how these containers were reused.

In **BC**, path counts are accumulated through an `Array` reduction, while each level's frontier is built with a `Bag`, whose insertions commute and thus need no ordering. **BFS** reuses the same `Bag` to construct the next frontier and a `RelaxedArray` for the `parent` array, exploiting that any discoverer may set a node's parent. **CC** relies entirely on a `RelaxedArray`: union-find link and compress converge even when their accesses race, so stale reads and atomic min-updates suffice. **PR** combines a `Scalar` reduction for the additively accumulated error with a `RelaxedArray` for node updates, whose staleness convergence tolerates. **SSSP** builds its bucketed worklist with a `Multimap`, since the nodes within a bucket are unordered. **TC** needs only a `Scalar` reduction to accumulate triangle counts. **SC** and **KM** both accumulate per-point contributions through `Array` (1D and 2D) and `Scalar` reductions, as each point's contribution is computed independently. **BT** generates particles in any order with a `Bag` reduction and gives each worker a `PrivateObject` body model as scratchpad, drawing random numbers from a `RandNumGen`. **CN** similarly uses `PrivateObject` annealer states and a `RandNumGen`, tolerating stale netlist states through atomic updates. Finally, **XZ** compresses input chunks independently and accumulates the results in a `Sequence`, serialized once at the end. The clause allowing two sequence appends to run in parallel was the only non-commutative clause we declared across the entire evaluation.

6.5 RQ5: Related Work

We compare T-800 against three representative approaches, shown in Figure 10. DOALL represents an automatic parallelizing compiler able to detect and exploit scalar and array reductions. ALTER [38] relies on annotations that declare which dependences in a loop may be speculated on or ignored. STATS [9] is designed to exploit the non-determinism inherent in programs such as BT and CN. For each benchmark, the figure reports the best speedup achieved among these baselines.

DOALL offers speedup only on benchmarks whose parallelism it can statically recover. On BFS, which alternates a top-down and a bottom-up step, it parallelizes the bottom-up step whose iterations are independent, while PR, TC, and SC are handled because they rely solely on scalar and array reductions. ALTER, instead, unlocks parallelism in CC thanks to its permissive execution model, and obtains some speedup on KM through speculation, where prior static approaches fail to detect all of the array reductions [16]. STATS targets the non-determinism inherent in BT and CN; since we could not evaluate the system directly, we report for these benchmarks the speedup of their manual implementations. Across the suite, T-800 matches or exceeds the best of these approaches on nearly every benchmark, achieving a 3.0× higher speedup on average.

7 Related Work

Prior compilers either fail at detecting when instructions can run in parallel or provide no mechanism for expressing enough algorithmic properties.

Layout Transformations. Layout transformations applicable to scalars and arrays have a rich history. Prominent examples include array reduction [10, 14, 16, 27, 35, 43] and array privatization [3, 12, 22, 24, 37]. These techniques are predominantly pattern-driven and based on a fixed repertoire of idioms specified by compiler developers. More recent work underlines the importance of decoupling the specification from detection of reductions [16]. The user is required to express a reduction in a constraint language through which the compiler detects. This allows compiler developers to extend the kinds of array reductions that a compiler can detect. However, the proposed solution is not suitable for capturing more than one reduction within a single loop. For more general data structures, [43] introduces a dependence-based approach to discover pointer-based data structures that can be privatized. Such data structures are then expanded via a fixed set of rules.

Collections. An early example of "looking at dependences at a coarse granularity" has been explored on collection semantics for Java [42]. Authors emphasize the importance of higher-level semantics for code analysis and propose a collection-aware analysis that eliminates some dependences arising from operations like set insertions or traversal. However, it is not possible to extend the structural relations between collection operations without compiler modifications. Earlier work on pointer-based data structures uses programmer-provided aliasing descriptions [17] or pointer and shape analyses [15] to prove that accesses to distinct parts of a structure do not conflict. This improves dependence-analysis precision but is ineffective on dependences that would manifest at runtime. MEMOIR [25] models sequences, and associative arrays as first-class citizens in the compiler. While this decouples collections from their implementation, it remains unclear how a compiler could automatically infer the appropriate layout transformation for composed collections. These models lock programmers to a fixed set of options and force compilers to handle them via hard-coded transformations that may not align with user needs. Concurrent collections (e.g., TBB [31], Folly [26]) offer out-of-the-box thread-safe operations that require no layout transformations. We consider these complementary to our work, as their thread-safety guarantees can be expressed as clauses and thus exploited by our compiler.

Handling Dependences. Some systems leverage annotations written by developers. In ALTER [38] developers can annotate loops with assertions that permit "breakable dependences", instructing the compiler that not all loop-carried dependences need to be respected. It also allows compilers to speculate that during a chunk of loop iterations some dependence will not manifest. Similarly, GALOIS [20] runs loop iterations as if any real dependence could be ignored, rolling back any computation that turns out to have violated it. Such speculation, however, is effective only when conflicts are rare. Other systems like Paralax [39] allow users to assist dependence analysis in identifying variables that are always written before being used in a loop, enabling privatization of scalar variables. Breaking dependences has been proven helpful for parallelism in algorithms that can tolerate output distortion [8, 34, 38] or are non-deterministic [9]. In STATS [9], programmers declare "state" dependences in non-deterministic programs through a dedicated interface. The compiler satisfies these dependences with compiler-generated auxiliary code that speculatively produces alternative data, which a runtime validates against the original computation to preserve output quality.

Commutativity Analysis. Commutativity analysis also reasons about programs at the level of high-level operations rather than individual instructions [1, 32]. Dynamic commutativity analysis [40] finds cases where permutations of the iteration order lead to the same output. These approaches focus primarily on detecting parallelism, while exploiting the detected opportunities may require specialized parallel code generation. In a DTG, terminable dependences address this need by carrying recipes for their removal, rather than being satisfied through synchronization as in CommSet [28].

Moreover, iteration commutativity is not necessary for parallelization. In XZ, for example, the hottest loop compresses and concatenates chunks. Associativity enables parallelization even though concatenation is non-commutative.

8 Conclusion

Automatic parallelization stalls not because dependence analysis is imprecise, but because the knowledge developers use to expose parallelism cannot be recovered by static analysis. Rather than teaching static analyses to match the programmer's intuition, this paper takes a step toward bridging that gap by giving developers a direct channel to convey it to the compiler. We let developers model high-level operations and declare the layout transformations that make them independent. Thanks to this information, we are able to define the DTG, which encodes not only dependences but also recipes to break them. We empirically show that, through terminable dependences, our compiler is able to nearly match the performance of hand-written parallel code on two different architectures. These results suggest that a modest, non-disruptive IR extension can encode those key properties that programmers routinely leverage during manual parallelization.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant NSF CCF-2107042. Any opinions, findings, conclusions, and recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] Farhana Aleen and Nathan Clark. 2009. Commutativity analysis for software parallelization: letting program transformations see the big picture. *ACM Sigplan Notices* 44, 3 (2009), 241–252.

[2] Sotiris Apostolakis, Ziyang Xu, Zujun Tan, Greg Chan, Simone Campanoni, and David I August. 2020. SCAF: a speculation-aware collaborative dependence analysis framework. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 638–654.

[3] Utpal Banerjee, Rudolf Eigenmann, Alexandru Nicolau, and David A Padua. 1993. Automatic program parallelization. *Proc. IEEE* 81, 2 (1993), 211–243.

[4] Scott Beamer, Krste Asanović, and David Patterson. 2015. The GAP Benchmark Suite. arXiv:1508.03619 [cs.DC]

[5] William Blume and Rudolf Eigenmann. 1992. Performance analysis of parallelizing compilers on the Perfect BenchmarksTM Programs. *IEEE Transactions on Parallel and Distributed Systems* 3, 6 (1992), 643–656.

[6] William Blume, Rudolf Eigenmann, Keith Faigin, John Grout, Jay Hoeflinger, David A. Padua, Paul Petersen, William M. Pottenger, Lawrence Rauchwerger, Peng Tu, and Stephen Weatherford. 1994. Polaris: Improving the Effectiveness of Parallelizing Compilers. In *Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing (LCPC '94)*. Springer-Verlag, Berlin, Heidelberg, 141–154.

[7] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. 2018. SPEC CPU2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. 41–42.

[8] Simone Campanoni, Glenn Holloway, Gu-Yeon Wei, and David Brooks. 2015. HELIX-UP: Relaxing program semantics to unleash parallelization. In *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 235–245.

[9] Enrico A Deiana, Vincent St-Amour, Peter A Dinda, Nikos Hardavellas, and Simone Campanoni. 2018. Unconventional parallelization of nondeterministic applications. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. 432–447.

[10] Johannes Doerfert, Kevin Streit, Sebastian Hack, and Zino Benaissa. 2015. Polly's polyhedral scheduling in the presence of reductions. In *5th International Workshop on Polyhedral Compilation Techniques (IMPACT)*. Amsterdam, The Netherlands.

[11] Jason Evans. 2006. A Scalable Concurrent malloc(3) Implementation for FreeBSD. In *Proceedings of the BSDCan Conference*. Ottawa, Canada. https://people.freebsd.org/~jasone/jemalloc/bsdcan2006/jemalloc.pdf

[12] Paul Feautrier. 1988. Array expansion. In *ACM International Conference on Supercomputing 25th Anniversary Volume*. 99–111.

[13] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. 1987. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9, 3 (1987), 319–349.

[14] Allan L Fisher and Anwar M Ghuloum. 1994. Parallelizing complex scans and reductions. *ACM SIGPLAN Notices* 29, 6 (1994), 135–146.

[15] Rakesh Ghiya, Laurie J. Hendren, and Yingchun Zhu. 1998. Detecting Parallelism in C Programs with Recursive Data Structures. In *Compiler Construction (Lecture Notes in Computer Science, Vol. 1383)*. Springer, 159–173. doi:10.1007/BFb0026429

[16] Philip Ginsbach and Michael FP O'Boyle. 2017. Discovery and exploitation of general reductions: A constraint based approach. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 269–280.

[17] Joseph Hummel, Alexandru Nicolau, and Laurie J. Hendren. 1994. A Language for Conveying the Aliasing Properties of Dynamic, Pointer-Based Data Structures. In *Proceedings of the 8th International Symposium on Parallel Processing*. IEEE Computer Society, 208–216.

[18] Richard Johnson, David Pearson, and Keshav Pingali. 1994. The program structure tree: Computing control regions in linear time. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*. 171–185.

[19] Maria Kotsifakou, Prakalp Srivastava, Matthew D Sinclair, Rakesh Komuravelli, Vikram Adve, and Sarita Adve. 2018. Hpvm: Heterogeneous parallel virtual machine. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 68–80.

[20] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L Paul Chew. 2007. Optimistic parallelism requires abstractions. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 211–222.

[21] William Landi. 1992. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)* 1, 4 (1992), 323–337.

[22] Zhiyuan Li. 1992. Array privatization for parallel execution of loops. In *Proceedings of the 6th International Conference on Supercomputing*. 313–322.

[23] Angelo Matni, Enrico Armenio Deiana, Yian Su, Lukas Gross, Souradip Ghosh, Sotiris Apostolakis, Ziyang Xu, Zujun Tan, Ishita Chaturvedi, Brian Homerding, Tommy McMichen, David I. August, and Simone Campanoni. 2022. NOELLE Offers Empowering LLVM Extensions. In *CGO*. doi:10.1109/CGO53902.2022.9741276

[24] D Maydan, S Amarsinghe, and M Lam. 1993. Data dependence and data-flow analysis of arrays. In *Languages and Compilers for Parallel Computing: 5th International Workshop New Haven, Connecticut, USA, August 3–5, 1992 Proceedings 5*. Springer, 434–448.

[25] Tommy McMichen, Nathan Greiner, Peter Zhong, Federico Sossai, Atmn Patel, and Simone Campanoni. 2024. Representing Data Collections in an SSA Form. In *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 308–321.

[26] Meta. 2012. Folly: An open-source C++ library. https://github.com/facebook/folly. Accessed: 2026-01-26.

[27] Bill Pottenger and Rudolf Eigenmann. 1995. Idiom recognition in the Polaris parallelizing compiler. In *Proceedings of the 9th International Conference on Supercomputing*. 444–448.

[28] Prakash Prabhu, Soumyadeep Ghosh, Yun Zhang, Nick P Johnson, and David I August. 2011. Commutative set: A language extension for implicit parallel programming. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. 1–11.

[29] Ganesan Ramalingam. 1994. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16, 5 (1994), 1467–1471.

[30] Lawrence Rauchwerger and David Padua. 1994. The privatizing doall test: A run-time technique for doall loop identification and array privatization. In *Proceedings of the 8th International Conference on Supercomputing*. 33–43.

[31] James Reinders. 2007. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media, Inc.

[32] Martin C Rinard and Pedro C Diniz. 1996. Commutativity analysis: A new analysis framework for parallelizing compilers. *ACM SIGPLAN Notices* 31, 5 (1996), 54–67.

[33] Tao B Schardl, William S Moses, and Charles E Leiserson. 2017. Tapir: Embedding fork-join parallelism into LLVM's intermediate representation. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 249–265.

[34] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. 2011. Managing performance vs. accuracy trade-offs with loop perforation. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 124–134.

[35] Toshio Suganuma, Hideaki Komatsu, and Toshio Nakatani. 1996. Detection and global optimization of reduction operations for distributed parallel machines. In *Proceedings of the 10th international conference on Supercomputing*. 18–25.

[36] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th international conference on compiler construction*. 265–266.

[37] Peng Tu and David Padua. 1993. Automatic array privatization. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 500–521.

[38] Abhishek Udupa, Kaushik Rajan, and William Thies. 2011. ALTER: exploiting breakable dependences for parallelization. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. 480–491.

[39] Hans Vandierendonck, Sean Rul, and Koen De Bosschere. 2010. The paralax infrastructure: Automatic parallelization with a helping hand. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. 389–400.

[40] Christos Vasiladiotis, Roberto Castaneda Lozano, Murray Cole, and Björn Franke. 2021. Loop parallelization using dynamic commutativity analysis. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 150–161.

[41] Michael Joseph Wolfe. 1978. *Techniques for improving the inherent parallelism in programs*. Technical Report UIUCDCS-R-78-929. University of Illinois.

[42] Peng Wu and David Padua. 1999. Containers on the parallelization of general-purpose Java programs. In *1999 International Conference on Parallel Architectures and Compilation Techniques (Cat. No. PR00425)*. IEEE, 84–90.

[43] Hongtao Yu, Hou-Jen Ko, and Zhiyuan Li. 2013. General data structure expansion for multi-threading. *ACM SIGPLAN Notices* 48, 6 (2013), 243–252.

[44] Xusheng Zhan, Yungang Bao, Christian Bienia, and Kai Li. 2017. PARSEC3. 0: A multicore benchmark suite with network stacks and SPLASH-2X. *ACM SIGARCH Computer Architecture News* 44, 5 (2017), 1–16.

[45] Xiaochun Zhang, Timothy M Jones, and Simone Campanoni. 2021. Quantifying the Semantic Gap Between Serial and Parallel Programming. In *2021 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 151–162.

A Benchmark Parallelization Insights

Table 3. Principal algorithmic insights and mechanisms used to enable parallelization in each benchmark.

Ben.	Insight behind parallelization	Mechanism to leverage it
BC	Path counts accumulate additively; per-level frontier order is irrelevant	Array reduction; frontier creation via bag reduction
BFS	Any discoverer may set the parent; per-level frontier order is irrelevant	Relaxed array accesses; frontier creation via bag reduction
CC	Union-find link and compress converge even under racing operations	Relaxed array accesses; atomic min-updates
PR	Error accumulates additively; convergence tolerates stale node updates	Scalar reduction; relaxed array accesses
SSSP	Nodes in each worklist bucket are unordered	Bucketed worklist created via multimap reduction
TC	Triangle counts accumulate additively	Scalar reduction
BT	Particles and weights can be generated in any order; body model serves as per-particle scratchpad	Particle generation via bag reduction; body model defined as a private object
CN	Netlist updates may use stale states; annealers can run independently	Atomic netlist updates; states are private objects
SC	Point contributions to the clustering cost can be computed independently	Array reduction; scalar reduction
KM	Points independently contribute to per-cluster centroids and counts	1D and 2D array reductions; scalar reduction
XZ	Input chunks can be compressed independently	Compressed chunks accumulated in a sequence and serialized via a sequence reduction

B Parallelized Loop Characterization

Table 4. Characterization of loops parallelized by T-800 *before* parallelization. Clauses and E-function calls report the static information enabling each parallelization; Unordered identifies loops that admit arbitrary iteration assignment.

Benchmark	Total time (s)	Number of invocations	Average time (s)	ROI coverage (%)	Clauses	E-function calls	Unordered
BC	23.2	13	1.784	65.0	2	2	✓
	12.4	14	0.884	34.7	5	6	✓
BFS	6.9	2	3.436	84.3	4	4	✓
	1.2	6	0.194	14.3	4	4	✓
CC	11.2	2	5.589	67.3	3	7	✓
	3.5	3	1.179	21.3	3	6	✓
	1.6	1	1.649	9.9	3	8	✓
	0.2	1	0.245	1.5	1	1	✓
PR	156.0	5	31.196	99.8	6	7	✓
SSSP	54.1	561	0.096	99.6	3	4	✓
TC	189.4	1	189.42	98.1	1	1	✓
CN	116.2	6000	0.019	100.0	5	7	✓
SC	478.5	8771	0.055	97.2	1	1	✓
BT	118.5	1305	0.091	94.9	4	8	✓
KM	126.1	461	0.273	100.0	2	2	✓
XZ	184.5	1	184.5	99.9	1	2	×