The Parallel-Semantics Program Dependence Graph for Parallel Optimization

Yian Su*
Northwestern University
Evanston, USA

Brian Homerding*
Northwestern University
Evanston, USA

Haocheng Gao
Northwestern University
Evanston, USA

Federico Sossai
Northwestern University
Evanston, USA

Yebin Chon
Princeton University
Princeton, USA

David I. August
Princeton University
Princeton, USA

Simone Campanoni†
Northwestern University
Evanston, USA

***Abstract*—Modern shared-memory parallel programming models, such as OpenMP and Cilk, enable developers to encode a parallel execution plan within their code. Existing compilers, including Clang and GCC, directly lower or add additional compatible parallelism on top of the developers' plan. However, when better parallel execution plans exist that are incompatible with the original plan, compilers lack the capability of disregarding it and replacing it with a better one. To address this problem, this paper introduces the parallel-semantics program dependence graph (PS-PDG), an extension of the program dependence graph (PDG) abstraction that can simultaneously represent parallel semantics derived from both the developer's original plan and the compiler's own analysis. To demonstrate the power of PS-PDG, this paper also introduces GINO, an LLVM-based compiler capable of optimizing parallel execution plans using PS-PDG. Through exploring, reasoning, and implementing better parallel execution plans unlocked by PS-PDG, GINO outperforms the developer's original parallel execution plan by 46.6% at most, and by 15% on average over 56 cores across 8 benchmarks from the NAS benchmark suite.**

I. Introduction

Modern hardware is parallel. Despite decades of research, many factors still render automatic parallelization of sequential code impractical. Because of this, developers take advantage of multicore processors by explicitly writing parallel code. The community has introduced compiler extensions and libraries in an attempt to boost the productivity of developers interested in thread-level parallelism (TLP). OpenMP [34] and Cilk [10] shine as the most renowned parallel programming models, where developers can directly express sections of the code that *can* and *will* run in parallel. In these models, the transformation into multithreaded code (e.g., via pthreads) is done by the compiler, which follows developer specifications without objection. This constrained approach suffers from a major drawback. A compiler that by contract follows the developer-encoded *parallel execution plan*[1], i.e., which instructions will run in parallel and how, is severely limited in its ability to tailor optimizations for a target architecture. For example, loops marked for multithreaded execution will execute in that fashion regardless of the profitability of TLP in the target environment. In some cases, SIMD execution may be preferable for tight loops or single-core execution but due to the compilation constraints that OpenMP follows, each annotated loop must execute according to the model specified by the developer. Encouragingly, the same information that lets one reason about multithreading can be reused to reason about vectorization. DOALL loops annotated with `omp for` are candidates for vectorization since they are marked as having no inter-iteration dependences by the developer. Dependences between instructions have in fact been a key abstraction to reason about parallelization of *sequential* code, but the long-established *program dependence graph* (PDG) was conceived to represent sequential programs; its limitations become quickly evident when reasoning about *already parallel* code. Although patterns like reductions on scalars can be amenable for parallelization even when abstracted in the PDG (e.g., by inspecting its SCCs), in general, the semantics that parallel programs carry go beyond data or control dependences. For instance, clauses like `firstprivate` or `lastprivate` encode opportunities that cannot be represented by the sole use of dependences or lack thereof. Similar limitations led research to develop parallel IRs, e.g., TAPIR [40], that can capture what *will* run in parallel while enabling classic optimizations for sequential IRs. However, like TAPIR, other existing parallel IRs [38], [37], [42], [1], [20], [32], [20] are not suited for *parallel* optimization of parallel code as they encode a single inflexible execution plan that compilers must respect. In other words, **both the PDG and existing parallel IRs fail to abstract the semantics of parallel code while allowing optimization of the parallelism itself**.

This paper proposes the *Parallel-Semantics Program Dependence Graph* (PS-PDG) – an abstraction that can simultaneously represent parallel semantics derived from both the developer's original parallel execution plan and the compiler's own analysis. The PS-PDG extends the PDG to allow compilers to access a larger optimization space and restructure parallelism in a way that makes the best use of the target architecture. We introduce GINO, an optimizing compiler that

*Both authors contributed equally to this research.
†Now at Google.
[1]We use the terms *parallel execution plan* and *parallelization plan* interchangeably throughout the paper.

leverages the proposed PS-PDG and is able to craft and lower a parallel execution plan for both multithreaded and vectorized execution.

The main contributions of this work are:

- A definition of the PS-PDG: a hierarchical graph that represents parallel semantics expressed by both the developers' knowledge and compiler analyses (§III).
- A detailed analysis of the necessity of each element of the PS-PDG (§IV).
- An empirical proof of the inadequacy of the PDG as an abstraction for the optimization of parallel programs (§V).
- An evaluation of GINO – The first LLVM-based optimizing compiler that uses the PS-PDG for parallel optimization (§V).

II. Background & Motivation

This section presents an OpenMP program to illustrate how developers explicitly encode a parallel execution plan in the source code. We then demonstrate that an alternative, more efficient plan exists, which cannot be realized within the existing compilation pipeline. This case motivates the need for the PS-PDG abstraction, which captures the precise constraints of a parallel program and enables more effective parallel optimization.

A. Developers Explicitly Encode Parallelism

Modern shared-memory *parallel programming models* (PPMs) allow developers to encode parallelism directly in their applications to achieve high performance and energy efficiency. A *parallel execution plan* specifies *what* to parallelize (e.g., loops), *how* to parallelize (e.g., variable privatization and initialization), and the *execution model* to use (e.g., tasks, threads, vectorization). Among PPMs, OpenMP [34] is one of the most widely adopted: it enables developers to encode a parallel execution plan in their code using a set of compiler directives (pragmas in C/C++). For instance, `#pragma omp parallel for` distributes loop iterations across multiple threads. Other pragmas offer greater control over the parallel execution, such as `critical`, which indicates that the enclosed code region is protected and should only be executed by a single thread at any given time.

Fig. 1 shows the OpenMP source code from the hottest computation in the `IS` benchmark of the NAS benchmark suite [7], together with its parallel execution plan encoded using OpenMP pragmas. The entire kernel is enclosed in a `#pragma omp parallel` region, which spawns multiple threads, each executing the enclosed code region in parallel. Loops ① and ③ do not have worksharing pragmas, so each thread in the parallel region executes the entire loop on its private copy of array `prv_buff1`. Loop ② instead has its iterations distributed across threads. Finally, loop ④ is enclosed in a `critical` section to serialize updates to the shared array `key_buff1` and avoid data races.

B. Optimizing Parallel Programs

Let us reconsider the parallel region of `IS` shown on the right side in Fig. 1, now with a different parallel execution plan. In this new plan, iterations of loop ① are parallelized across threads, each operating on a disjoint slice of the shared array `pre_buff1`. Before loop ②, each thread privatizes `pre_buff1`, creating a local copy. Loop ② is then executed in parallel as before. Once finished, the private copies are reduced into the shared array. As a result, loop ③ only needs to run on a single thread, avoiding parallel overhead that the original plan incurred. Finally, with only one shared copy of `pre_buff1` remaining, loop ④ can be distributed across threads without requiring a `critical` section.

Transforming the original plan into this optimized one is non-trivial. The compiler must first recognize that `prv_buff1` is privatizable through complex dependence analysis or developer-provided annotations. It must then identify the updates in loop ② as reduction operations, and ensure that aliasing does not occur among the involved array pointers.

C. Existing Optimizing Compilers Do Not Leverage Parallel Semantics

Unfortunately, existing optimizing compilers for PPMs such as OpenMP [34] and Cilk [10] cannot realize this transformation. As illustrated in the left flow in Fig. 2, the compilation pipeline lowers the developer's parallel execution plan directly into runtime calls during the frontend stage. This design simplifies compiler implementation, given that much of the sequential optimization can be reused. However, this approach has a fundamental drawback: the **parallel semantics encoded by the developer are discarded** at the compiler frontend and **unrecoverable from the middle end**. As a result, valuable information – such as parallel loops, which have no loop-carried data dependencies, or variables that can be privatized along with their initialization information – is lost and cannot be recovered for optimization [40].

D. Optimizing Compilers Need Powerful Abstraction to Represent Parallel Semantics

The motivating example proves that compilers need the capability of optimizing the parallel execution plan expressed by developers. Today's compilers [11], [45], [31], [4], [5] successfully use the PDG abstraction, but PDG does not capture the parallel semantics expressed in the source code. To overcome this limitation, this paper proposes the PS-PDG, an abstraction that captures the precise parallel constraints implied by the developer's plan. With PS-PDG, compilers can reason about a space of semantically equivalent parallel execution plans. The PS-PDG enables the pipeline shown in the right flow in Fig. 2. The proposed pipeline does not rigidly follow the encoded parallel execution plan (like today's compilers do), and instead enables compilers to select the plan that best matches the underlying architecture.

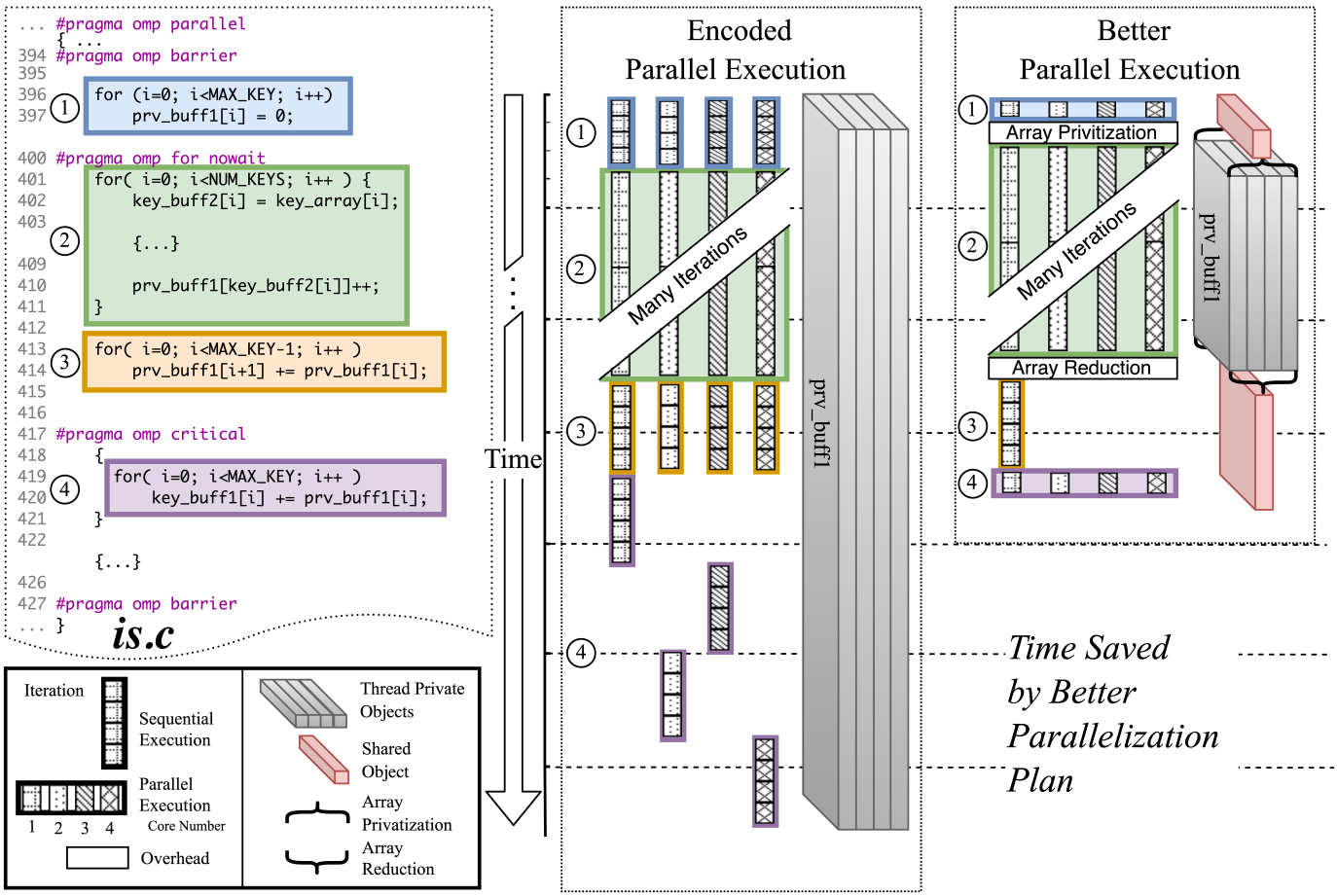

Fig. 1. The key computational kernel from the IS benchmark with the original and a more performant compiler-selected parallel execution plan.

III. PS-PDG Definition

PPMs like OpenMP enable developers to make parallelization decisions explicit. A developer can decide where to spawn threads or tasks to distribute the computation of a loop, and how to synchronize their execution (i.e., the parallel execution plan of that program).

Beyond controlling what code can run in parallel and when it can do so, a parallel execution plan also *implies properties of the code* of the original program. For example, a parallel execution plan described using OpenMP can include the declaration that iterations of a loop will run in parallel during their executions. This plan implies the property that the target loop has no loop-carried dependences between its iterations. Another example is an OpenMP `critical` section in a loop, which implies both the need to enforce the atomic property of the target code segment and that any order of invocations of the target segment between loop iterations is valid. We refer to this implied information as *the precise constraints of a parallel program*, which is captured by the PS-PDG.

The PS-PDG extends the PDG abstraction to capture the precise parallel constraints of an OpenMP or Cilk program. Like the PDG, the PS-PDG has nodes to represent computation and edges to represent dependences within the computation, it also includes variables to represent data and use/def edges to represent the relation between data and its computation. As shown in Table I, a PS-PDG consists of one or more nodes with zero or more edges, variables and variable accesses. The rest of this Section describes each extension in detail.

TABLE I
Complete PS-PDG Definition

PS-PDG	::=	(Node$^+$, Edge*, Variable*, VariableAccess*)
Node	::=	(Instruction | HierarchicalNode, Trait*)
HierarchicalNode	::=	(Node$^+$, Context?)
Trait	::=	(Singuler | Unordered | Atomic, Context)
Edge	::=	DirectedEdge | UndirectedEdge
DirectedEdge	::=	(Node$_{\text{producer}}$, Node$_{\text{consumer}}$, Data-selector?)
UndirectedEdge	::=	(Node, Node, Context)
Data-selector	::=	(Any-Producer | Last-Producer | All-Consumers, Context)
Variable	::=	(Privatizable | Reducible, Context)
VariableAccess	::=	(Variable, Node$^*_{\text{use}}$, Node$^*_{\text{def}}$)
Context	::=	*Unique Identifier*

A. Hierarchical Nodes

Explicit parallel programming enables developers to specify properties of a code region. Often such properties do not hold

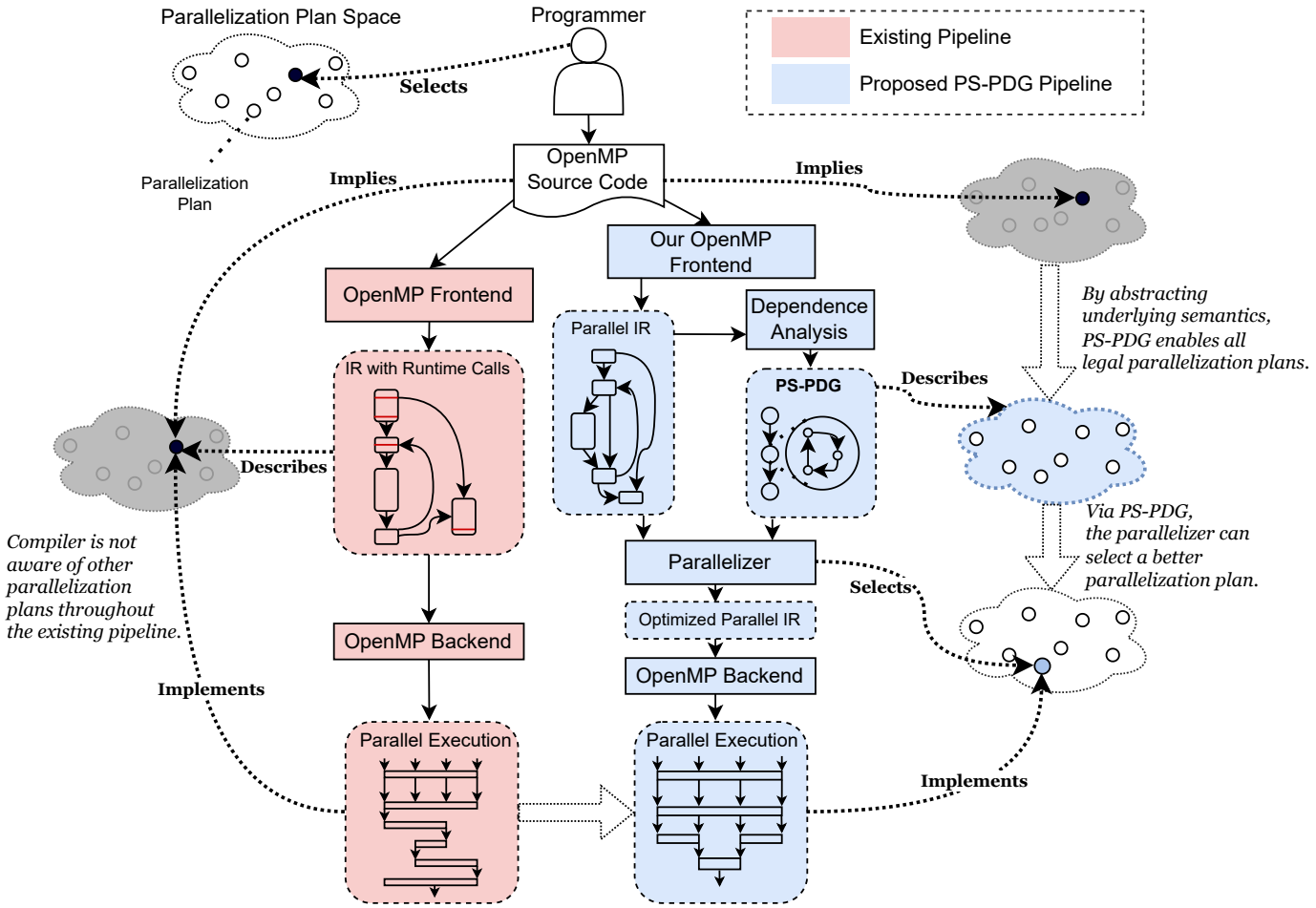

Fig. 2. Comparison of the existing pipeline with our proposed PS-PDG pipeline. Our proposed pipeline enables an optimizing compiler to reason and implement a more performant parallel execution plan. The Parallel IR refers to MLIR's `omp` dialect in our implementation.

at finer granularities (e.g., single instruction). For example, an OpenMP `critical` section declares that the code region as a whole has the atomicity property. This atomic property does not hold at a finer granularity like at the single instruction level that composes this critical code region. For this reason, the PS-PDG both adds the ability to have a single node that represents an entire code region and the ability to express properties at their node granularity (§III-B).

A node in the PS-PDG represents a non-empty set of instructions organizing the code hierarchically, shown in Fig. 3. For example, all instructions of a `critical` section in the PS-PDG is represented by a single node. More generally, a node `N` of the PS-PDG is a non-empty set of one or more instructions or other nodes such that both direct and indirect self-inclusions are not allowed. Having a single node representing a set of instructions is needed to capture the parallel semantics of parallel constructs that target more than a single instruction.

```
1 #pragma omp parallel for
2 for(int i=0; i<N; i++) {
3   {... parallel work ...}
4   #pragma omp critical
5   { x = {...};
6     y = x + {...};
7     z += globalUpdate(y, x); }
8   {... parallel work ...}
9 }
```

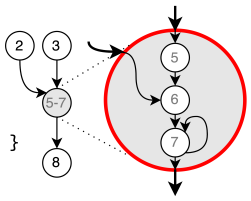

Fig. 3. Capturing properties of a region into hierarchical nodes with traits

B. Node Traits

Some properties expressed in a PPM are traits of a code region. These traits can be important for the correctness and/or performance of a parallel application, for instance, atomicity.

A node in the PS-PDG can have various traits. This paper implemented the three types of traits that are enough for the target languages OpenMP and Cilk: the atomic, orderless, and singular traits. An atomic node represents a set of computations that must be executed atomically during its parallel

execution. An orderless node expresses that different instances of that node can be executed in any order for a given context. A singular node represents a set of computations that must be executed by only a single instance for a given context. An example of a node traits is shown in Fig. 4.

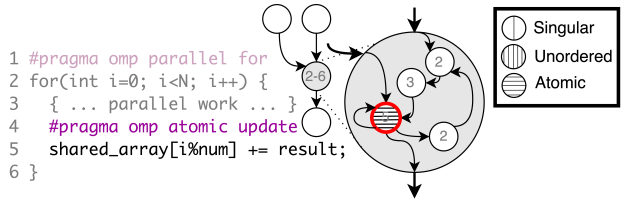

Fig. 4. How traits can be used to capture atomic updates.

C. Context

PPMs allow developers to express semantics attached to a code region only when executed within the context of another code region. For example, the code in a `single` OpenMP pragma needs to only be executed for one of the iterations of the innermost parallel loop that contains it. It does not however specify that code should only be executed by a single iteration of an outer loop. In other words, the parallel semantics of a `single` section is valid only in the context of the innermost loop that contains it. Because the contexts in which parallel semantics is valid cannot always be computed, the PS-PDG can specify contexts and their relation with parallel semantics.

A context in the PS-PDG represents a code region to which a parallel semantic applies. A context in the PS-PDG is a labeled hierarchical node, where the label is a unique identifier. Hence, hierarchical nodes of the PS-PDG that do not have a label are not contexts. A parallel semantic explicitly lists the contexts in which it is valid, as shown in Tab. I. For example, the hierarchical node `S` of Fig. 5 that captures the `single` code section declares its semantics applies only to context `A`, which is its target loop.

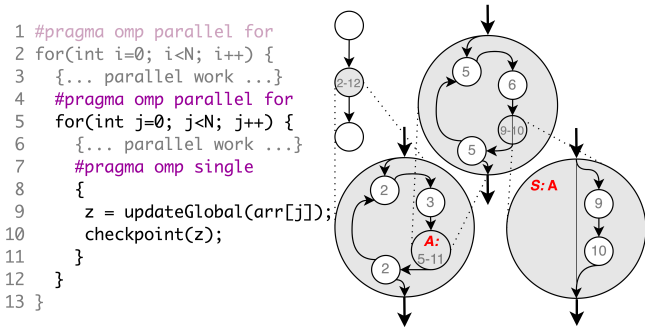

Fig. 5. Traits applied to context `A` captures the region's semantics.

D. Directed and Undirected Edges

Parallel programming allows the declaration that two code regions (or two instructions) depend on each other but their relative execution order is not important. This enables efficient parallel executions by avoiding unnecessary synchronizations. PS-PDG includes both directed and undirected dependences (edges) to capture this semantics (Fig. 6).

A directed edge in a PS-PDG follows the semantics of the PDG abstraction where the execution of the destination of that edge must wait for the edge's source execution. Instead, an undirected edge expresses a dependence between two computations (e.g., instructions) that cannot run in parallel, but any ordering of their execution is allowed.

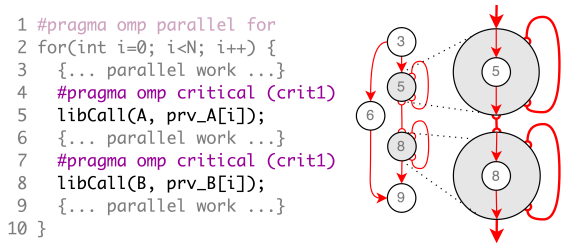

Fig. 6. Ordering constraints are captured with directed and undirected edges.

E. Data-Selector Directed Edge

The execution of an application typically includes many instances of a single static instruction (e.g., multiple executions of a single static instruction within a loop). There is always a clear producer-consumer relation between dependent instructions for sequential programs. For example, consider the instructions i and j shown in Fig. 7 which has a dependence from i to j. In this sequential program, the last instance of i executed before j will generate the data consumed by j (this is captured by the PDG). However, developers can express richer semantics when developing a parallel program. For example, a developer can express that the data generated by any instance of i can be used by j. This is not expressible in prior abstractions like the PDG. Hence, the PS-PDG introduces data-selectors that can be attached to a direct dependence.

A data-selector defines the set of dynamic instances of a static instruction. A directed edge in the PS-PDG can have up to two data-selectors: one per static instruction attached to the edge. A data-selector of the producer of a dependence defines which dynamic instance(s) of that producer are allowed to generate the data that will unlock the consumer.

This paper implements only the data-selectors required to capture the semantics of OpenMP and Cilk, which are the following:

- `Any Producer Selector`: The consumer may use data generated by any instance of the producer.
- `Last Producer Selector`: The consumer must use data generated by the last instance of the producer.
- `All Consumers Selector`: All consumers must use the data generated by the producer.

F. Parallel Semantic Variables

Efficient parallel execution often requires developers to express knowledge about the program's variables that go beyond their reads and writes and their data types. For example, developers can express that a variable can be privatized in

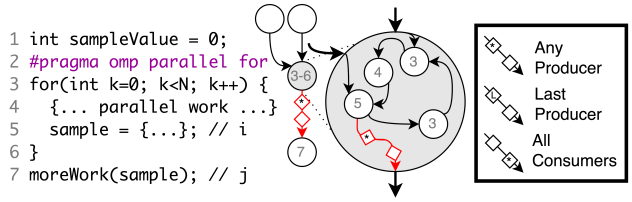

Fig. 7. Data-selector directed edges can capture non-trivial data relations.

threads/tasks and all private instances can be merged (reduced) using application-specific knowledge. This semantics goes beyond what can be expressed in sequential programming and therefore beyond what the PDG can capture (as the PDG was designed for sequential code). To preserve this semantic, the PS-PDG introduces the concept of variables and their parallel semantics (how to clone them, their identity value, and how to reduce them) in its abstraction.

A parallel semantic variable in PS-PDG represents a variable or memory object that can be cloned to create private copies that a thread or task can independently use and modify. This extension includes the code to execute to merge pairs of private copies together. To do so, the variable description includes the reference to a computational node of the PS-PDG that represents a function. This function takes two copies of a variable and it updates the first one with the result of the merge. This merging operation is what compilers can use to reduce all private copies of a variable into a single one. An example of parallel semantic variable is shown in Fig. 8.

Parallel semantic variables are accessed by computation (e.g., an instruction). Because such variables can be stored in memory, their accesses are not captured by the conventional use-def chains [3]. To preserve this relation, the PS-PDG adds the Use/Def edges from a variable to PS-PDG nodes to encode the semantics that a target node uses and/or defines the variable at the source of that edge.

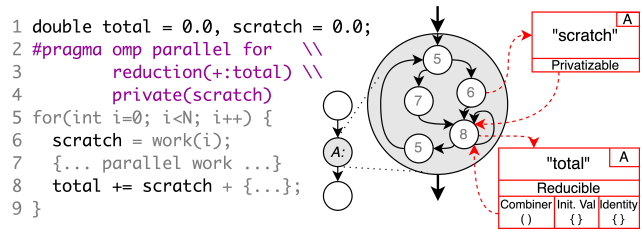

Fig. 8. Capturing developer knowledge about data through privatizable and reducible parallel semantic variables.

IV. The Necessity of PS-PDG Components

This section demonstrates that each feature of the PS-PDG is necessary to capture the semantics expressible using the OpenMP programming model. The same result can be obtained similarly for Cilk (not included for space constraints). Each extension is proved to be necessary by removing it from the proposed abstraction and showing that two parallel programs with two different parallel execution plans and semantics are now translated to the same PS-PDG when the extension under evaluation is not available. Additionally, this section provides an example that shows how each feature enables an important optimization. Overall, this section shows the value of each PS-PDG extension.

A. Hierarchical Nodes and Undirected Edges

To understand the value of Hierarchical Nodes (HN) and Undirected Edges (UE), consider the two semantically different programs shown in Fig. 9-A. The program on the left requires avoiding overlapping dynamic instances of the critical section but puts no restriction on their order. In contrast, the program on the right requires each dynamic instance of its critical section to be executed in loop-iteration order. The program on the left executes significantly faster than the one on the right because it does not require synchronizations to enforce this additional constraint. A compiler seeking the best parallel execution plan for each program in this way must know whether or not this extra degree of freedom (orderless) exists. In the PS-PDG, the undirected edge and hierarchical node features combined remove the ordering constraint while ensuring that dynamic instances of the connected nodes do not overlap. When this feature is removed, this semantic information is lost. Fig. 9-A demonstrates this by showing how these two programs map to the same PS-PDG lacking these features ("PS-PDG w/o HN and UE"). Furthermore, this orderless semantics cannot be represented by the "PS-PDG w/o HN and UE" because the orderless semantics does not hold at the single instruction granularity.

B. Node Traits

A node in the PS-PDG can hold various traits expressed in a parallel programming language. These traits can be important for the correctness and performance of the parallel application. To understand the value of Node Traits (NT), consider the two semantically different programs shown in Fig. 9-B. The program on the left requires the singular execution of the print statement, allowing for quick and simple output from the parallel application. In contrast, the program on the right does not include a `single` annotation for its print statement, meaning multiple calls to `printf`. To maintain correctness, then the compiler must understand how the `printf` fits into the parallel execution plan. Unfortunately, this is not possible when using the PS-PDG without Node Traits ("PS-PDG w/o NT"). Fig. 9-B demonstrates this by showing how these two programs map to the same "PS-PDG w/o NT". In the "PS-PDG w/o NT", the single execution semantic is lost. Further, the single execution trait cannot be determined by compiler analysis from any other aspect of the "PS-PDG w/o NT".

C. Contexts

To understand the value of Contexts (C), consider the two programs shown in Fig. 9-C. The program on the left executes the first call to `worker` in parallel while the program on the right executes it sequentially. By leveraging the parallelism in the hardware, the left program executes significantly faster

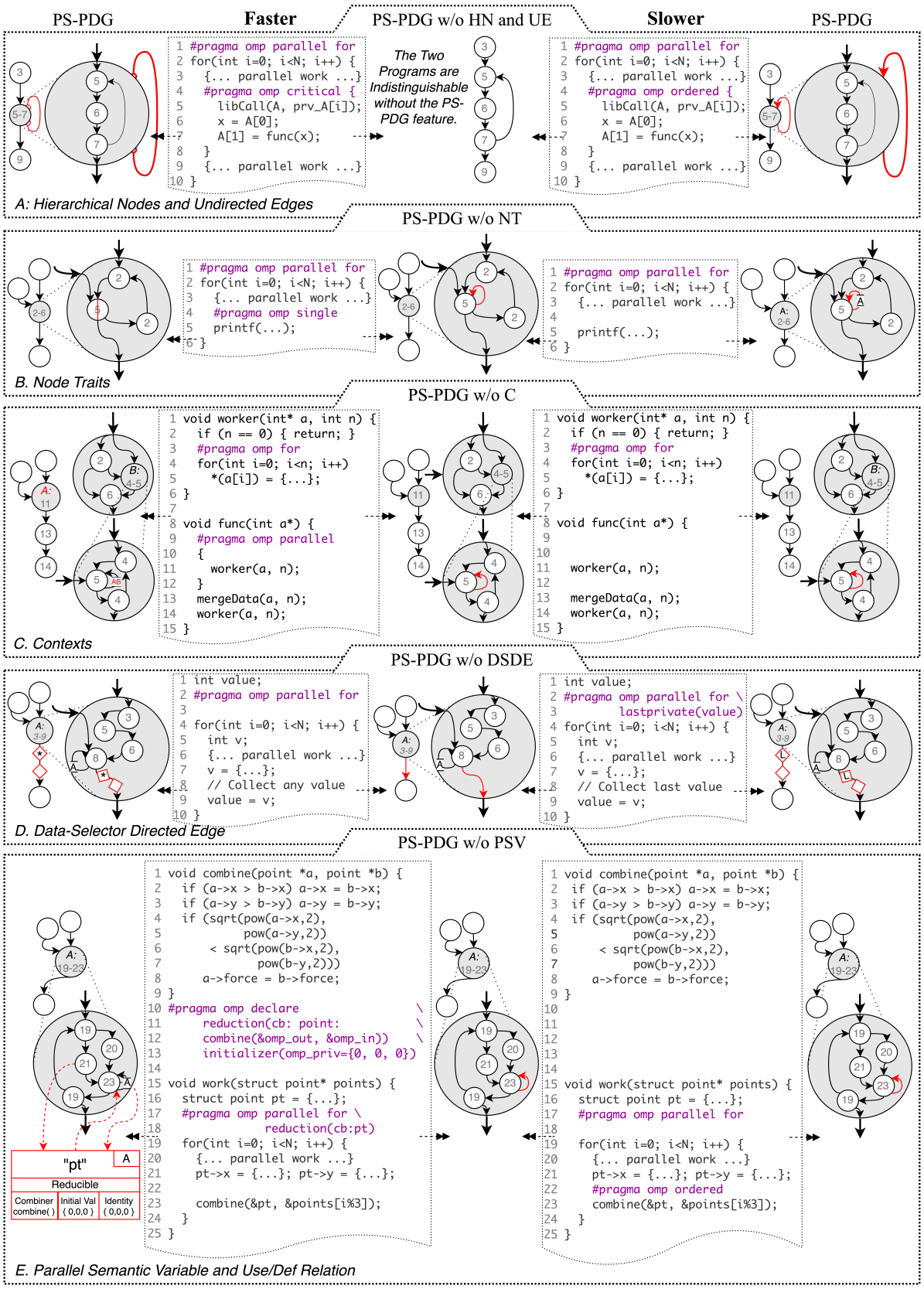

Fig. 9. Each feature of the PS-PDG is necessary since the removal of any PS-PDG feature would result in a loss of information. Without the given feature, the resulting abstraction is indistinguishable for the faster code (left) and the slower code (right). For readability, we present the program in C. In our implementation, the PS-PDG is built from the MLIR `omp` dialect.

than the program on the right. To generate the best parallel execution plan for the first `worker` call, the compiler needs to know in which context(s) the independent loop iteration semantic holds. PS-PDG Contexts represent the contexts in which code region parallel semantics hold. Without PS-PDG Contexts, the two programs map to the same "PS-PDG w/o C". Using only the "PS-PDG w/o C" in this example, the compiler cannot know when the loop iterations are independent of each other and must assume they are not.

D. Data-Selector Directed Edge

Data selectors can be added to the directed edges of the PS-PDG abstraction. Data selectors define which dynamic instance (or instances) of the source node can generate the data that the destination node needs. To understand the value of Data-Selector Directed Edge (DSDE), consider the two parallel programs shown in Fig. 9-D. Their semantics are different. The program on the right enforces that the value of the live-out variable `value` that can propagate outside the loop has to be the one generated during the last iteration of that loop. The program on the left of Fig. 9-D allows the propagation of the value generated by any loop iteration, which adds an extra degree of freedom. With this freedom, a consumer of the live-out variable can start before the end of the loop execution, allowing for more overlapping computation. Unfortunately, a PS-PDG without Data-Selector Directed Edges ("PS-PDG w/o DSDE") cannot distinguish these cases. This can be seen as both programs in Fig. 9-D map to the same "PS-PDG w/o DSDE". Thus, for correctness, the parallel execution plan generated from the "PS-PDG w/o DSDE" must enforce a stricter semantics of the program, the slower program on the right. Note that the DSDE semantics cannot be inferred by a code analysis on the "PS-PDG w/o DSDE".

E. Parallel Semantic Variable and Use/Def Relation

The PS-PDG includes Parallel Semantic Variables and Use/Def Relations (PSV) to represent a variable or object upon which the developer has encoded parallel semantics (e.g., how to reduce an object between tasks). These variables are connected to their computation (reads and writes) through Use/Def edges from parallel variables to nodes in the PS-PDG. Consider the two parallel programs shown in Fig. 9-E. The program on the left runs all iterations of the loop in parallel without any synchronization between them. Each thread operates on a private copy of the struct `pt`, then all private copies are reduced into a single one to be propagated to the code after the loop. This reduction is performed using application-specific knowledge. In contrast, the program on the right of Fig. 9-E has a single copy of the struct `pt` shared among all iterations of a loop. Accesses (reads and writes) of this array are synchronized using an ordered section. The program on the left executes significantly faster than the one on the right because it does not require any synchronization between the loop iterations running in parallel. A compiler that needs to decide the parallel execution plan to apply to the program on the left of Fig. 9-E needs to be aware of the ability to privatize and reduce the struct `pt` to generate the best parallel execution plan. Unfortunately, this is not possible when using a PS-PDG without Parallel Semantic Variables ("PS-PDG w/o PSV"). This becomes clear by observing that both programs in Fig. 9-E map to the same "PS-PDG w/o PSV". This means that the parallel execution plan generated from the "PS-PDG w/o PSV" must enforce the stricter semantics of the program on the right of Fig. 9-E where all array accesses are ordered. Furthermore, notice that the lost application-specific knowledge about the reduction of the struct `pt` cannot be inferred from the "PS-PDG w/o PSV".

V. Evaluation

This section evaluates GINO, the first optimizing compiler that leverages PS-PDG for parallel optimization. After describing GINO's compilation pipeline, parallel execution plan selection mechanism, as well as the experimental settings, we measure GINO's ability to improve performance when it automatically selects and implements a parallel execution plan better than the one specified by the developer, achieving an average **+15%** speedup. Next, we compare GINO against OpenMP when both use the same developer-encoded plan, showing performance parity with the state-of-the-art compiler infrastructure. Finally, we contrast GINO with a state-of-the-art PDG-based optimizing compiler, and demonstrate that the PDG alone is limited in expressiveness: even after supplying parallel semantics, it fails to match the PS-PDG's performance.

A. GINO Pipeline and Implementation

GINO takes OpenMP C/C++ source code as input and produces parallel binaries that implement an optimized parallel execution plan selected using the PS-PDG abstraction.

a) Implementation: GINO is built by extending NOELLE [31], a PDG-based compilation framework that augments LLVM with dependence-oriented abstractions and advanced alias analysis. In this work, we modified NOELLE in the following two ways. 1) To construct PS-PDG, we extended NOELLE's PDG abstraction to capture and represent parallel semantics described in §III. 2) We extended NOELLE to accept MLIR's `omp` [2] dialect as input (generated using Clang), which preserves explicit parallel semantics. This extended framework is referred to as NOELLE-PSPDG. The original NOELLE authors built an automatic parallelizer (referred to as NOELLE-par) on top of NOELLE, which parallelizes the code by selecting the best parallelization transformation among DOALL [17], HELIX [11], and DSWP [45] on a per-hot-loop basis, solely relying on the PDG. We extended NOELLE-par to operate on PS-PDG and implemented a heuristic-based plan selection mechanism. The above system overall is called GINO. After a plan is selected, GINO lowers it to LLVM IR and relies on the off-the-shelf Clang backend for code generation.

b) PS-PDG construction: The construction algorithm initially allocates a PS-PDG node per loop and uses

NOELLE's Forest abstraction [31] to organize them hierarchically per function. Then, for every `omp` operation (e.g., `omp.parallel`, `omp.wsloop`, `omp.critical`), we create a corresponding PS-PDG node. These nodes are nested within each other, respecting their region nesting relations. For each top-level node, we perform a data dependence analysis using both MLIR's AliasAnalysis [26] and MemoryEffects [29] interface. Nodes created from `omp.wsloop` operations don't need loop-carried dependences to be computed since the OpenMP worksharing semantics imply their absence. For all other constructs, memory dependences are computed. Critical sections (`omp.critical`) are conservatively modeled by inserting undirected dependence edges among all operations participating in the region, ensuring serialized execution. Directed edges are node-node pairs and are added between node A and B iff there exists at least one dependence (i_A, i_B) following the definition in PDG, where i_A (resp. i_B) is an operation contained in the region described by A (resp. B). Finally, dependence edges among `omp.critical` nodes within the same loop tree are marked as undirected. Register (def–use) dependences are not explicitly reconstructed, as they are already represented by MLIR's SSA form [28] and thus implicitly available to the PS-PDG. Parallel-semantics attributes such as `private`, `firstprivate`, and `reduction` are extracted directly from the argument list of the `omp` operations. For each such variable, we create a PSV instance and annotate it with its corresponding attribute kind, allowing the PS-PDG to precisely model the effects of privatization and reductions.

c) Parallel execution plan selection: We implemented a plan selection heuristic guided profile information. Specifically, loops contributing less than 1.2% of the total runtime are vectorized to avoid thread-management overhead, while loops above this threshold are both multithreaded and vectorized. If GINO detects a single core at compile time, it applies vectorization only.

B. Experimental Settings

We evaluated GINO against NOELLE-par, OpenCilk-2.0 [39], an optimizing compiler for parallel programs backed by TAPIR [40], and OpenMP. All systems use LLVM 14 as the underlying compilation infrastructure. This ensures a fair comparison since all compilers share most of the infrastructure and optimizations. Both OpenCilk and OpenMP binaries are generated using Clang. All benchmarks are compiled using `-O3 -march=native`.

a) Benchmarks: We evaluated GINO over all 8 benchmarks from the NPB 3.0-OMP C [33] version of the NAS [7] benchmark suite. We used input size `B` for benchmarks `BT` and `FT` due to gigabyte-size static variables and `C` for all others.

To evaluate OpenCilk, we ported all 8 NAS benchmarks by refactoring their parallel code regions by replacing each `#pragma omp parallel for` loop with `cilk_for` construct. OpenMP critical sections were rewritten using `cilk::mutex`, and scalar reductions were expressed using OpenCilk reducers. We restructured OpenMP

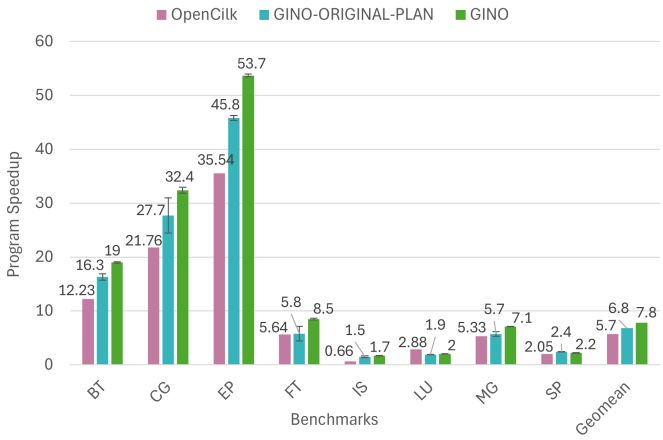

Fig. 10. The speedup on 56 cores with GINO following the developers' original parallel execution plans versus GINO's PS-PDG-improved parallel execution plan over 8 NAS benchmarks.

data attributes such as `private` and `firstprivate` to explicitly allocate all per-worker private data (and `memcpy` initialization where necessary) before entering the parallel region. Each thread-private object inside a parallel code region is accessed via a per-worker array indexed by the worker ID returned by `__cilkrts_get_worker_number()` runtime call. These changes preserve the original program semantics while enabling execution under Cilk's work-stealing [13] model.

b) Platform: All of our results are run and reported using an instance that runs Linux kernel 4.18.0 equipped with two Intel Xeon Gold 6258R processors featuring a total of 56 cores (28 per socket) running at 2.7 GHz, with 32K L1i, 32K L1d, 1024K L2, and 39424K L3 cache. Both hyperthreading and turbo-boost are disabled throughout the evaluation. We report the median result over 30 runs.

C. GINO Outperforms Developers' Parallel Execution Plans

GINO selects and implements better parallel execution plans that differ from those manually encoded by developers, and consistently delivers higher performance. Fig. 10 reports the speedups over sequential for parallel binaries with the original and improved parallel execution plans running on 56 cores. The speedup (on average) increases from **6.8**× to **7.8**× **(+15%)** over sequential baselines after delegating the choice of parallel execution plans to GINO. This demonstrates GINO's ability to generate optimized parallel binaries that leverage both the compiler's analysis and the parallel semantics expressed by developers.

a) GINO implements more performant parallel execution plans: GINO improves `EP` (**+17.2%**) and `IS` (**+13.2%**), by implementing more efficient parallel execution plans. As illustrated in the motivating example (§II), both benchmarks originally implement array reduction by privatizing and initializing a local copy of the array. The reduction is then protected by an OpenMP `critical` section, ensuring that each thread serializes updates to the global array.

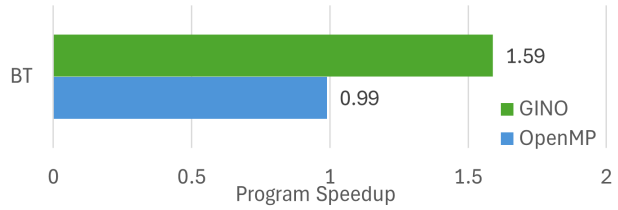

Fig. 11. Single core speedup for BT resulted from vectorization.

GINO recognizes this reduction pattern and chooses a more scalable implementation. Specifically, it allocates a contiguous block of stack memory, partitioned such that each thread owns a cacheline-aligned privatized array. Initialization of the privatized array is vectorized using `memset` instead of a scalar loop. The final reduction is performed by the main thread by aggregating results directly from each thread's local copy.

This plan improves performance in two key ways. 1) It eliminates the runtime overhead of lock/unlock operations protecting the `critical` region, avoiding contention and thread spinning. 2) It reduces cache coherence traffic: in the original plan, each update to the global array triggers cache invalidations between cores, whereas GINO's plan isolates updates to thread-local arrays and performs a single, cache-friendly aggregation step. The benefits become pronounced when the reduction array becomes large (e.g., 2048 integers for `IS`) and the program runs at scale (56 cores).

b) GINO vectorizes the code automatically when appropriate: GINO gains additional speedup on top of developers' parallel execution plans for `FT` (**+44.1%**), `BT` (**+16.6%**), and `CG` (**+16.5%**) by automatically vectorizing hot loops that are already parallelized. GINO treats parallelization and vectorization as distinct execution plans enabled by the same parallel semantics. Loops marked as DOALL parallel are identified as safe for vectorization since they contain no loop-carried dependences. For such loops, GINO inserts vectorization metadata in the LLVM IR and leverages LLVM's automatic vectorization pass [27]. To improve vectorization efficiency, GINO restructures candidate loops by introducing an inner loop with a fixed step size, determined at compile time from the available SIMD flags (e.g., AVX2, AVX-512) and operand data types. This approach frees developers from the burden of selecting an architecture-specific vector width, which is often error-prone and non-portable.

GINO also applies vectorization to loops where thread-level parallelization is not profitable. `MG` (**+24.6%**) benefits significantly from this strategy: the original plan parallelizes many small loops that are invoked repeatedly, incurring substantial overhead. GINO replaces them with vector execution, reducing overhead and improving performance.

c) GINO outperforms OpenCilk: GINO achieves higher performance than OpenCilk (**+37%**) overall, and performs better (**+20%**) than OpenCilk even when GINO respects the developer-encoded parallel execution plan. The performance difference stems from the fundamentally different execution

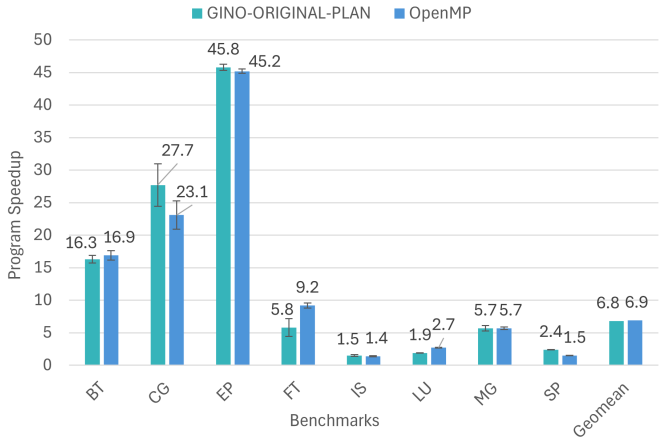

Fig. 12. The speedup on 56 cores with the developers' original parallel execution plans with GINO versus clang OpenMP over 8 NAS benchmarks.

models used by the two systems. GINO uses the OpenMP runtime for scheduling parallel loops, which uses large, statically assigned chunks and preserves per-thread data locality, which is well-suited to regular loop nests for NAS benchmarks. In contrast, OpenCilk relies on a divide-and-conquer work-stealing scheduler, which recursively subdivides iteration spaces and creates finer-grained tasks that introduce stealing overhead and reduce per-thread data locality. These effects become detrimental for regular loops such as those in `BT` and `SP`, where static scheduling delivers better performance.

Despite providing strong support for enabling sequential optimizations in parallel programs from TAPIR [40], OpenCilk shares the same limitation with OpenMP compilers that it does not explore or substitute semantically equivalent parallel execution plans, which can be realized using GINO.

d) GINO generates parallel binaries that adapt to the runtime environment: The best parallel execution plan depends on the runtime environment, for instance, the number of available cores. With fewer cores, thread-level parallelism introduces no performance but purely parallel overhead [44], making vectorization a more effective strategy. Fig. 11 illustrates this point by running the `BT` benchmark on a single core, compiled with OpenMP and GINO, separately. The OpenMP implementation simply lowers the developer's plan and yields no benefit over the sequential baseline. In contrast, GINO detects the runtime configuration and implements vectorization and achieves a **59%** performance improvement. This example demonstrates that the best parallel execution plan depends on the target environment, and PS-PDG enables compilers to generate binaries that adapt accordingly.

D. GINO Delivers On-Par Performance to OpenMP.

Fig. 12 compares GINO and OpenMP when both execute the developers' original parallel execution plans. **GINO achieves comparable or superior performance compared to the state-of-the-art OpenMP implementation**.

GINO achieves higher speedup in `CG` (**+17%**), where GINO-generated loop tasks get auto-vectorized by LLVM and

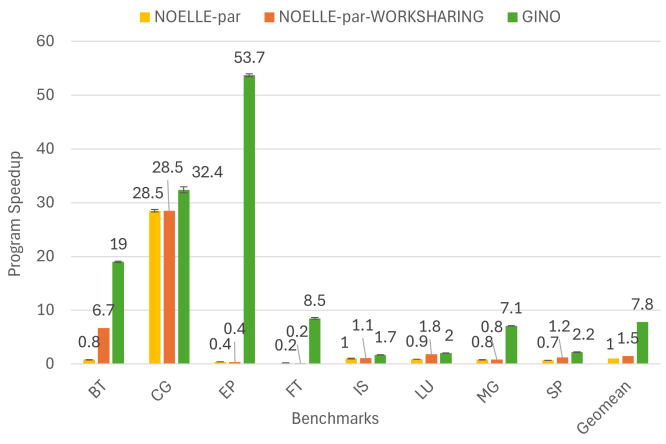

Fig. 13. Comparing GINO with two versions of NOELLE, one relies on the PDG generated from traditional dependence analysis, the other relies on the PDG improved with work-sharing pragmas.

outperform OpenMP. This optimization would require developers to explicitly specify `simd` pragmas and vectorization width hints in the OpenMP code.

GINO performs worse on `FT` (-35.8%), because Clang's OpenMP frontend benefits from specialized knowledge of OpenMP runtime APIs, enabling high-level optimizations such as parallel-region merging [25]. Those optimizations are currently unavailable in GINO.

E. The PS-PDG Serves as A Practical Abstraction for Parallel Optimization.

For decades, compilers have relied on the PDG for automatic parallelization and optimizations. Our results show that the PDG is insufficient for modern parallel optimization. The first and third bar of Fig. 13 compares the speedup of binaries generated by GINO (PS-PDG) and NOELLE-par (PDG) running on 56 cores. On average, GINO achieves a **7.8× speedup**, and consistently outperforms NOELLE-par across all benchmarks. One might think that this gap exists because of the alias analysis imprecision. Fig. 13 shows this hypothesis is false; We extended NOELLE to use the worksharing pragmas such as `omp for` to remove all spurious loop-carried dependences of the corresponding loops (as explored by prior work [46]). This is possible thanks to the OpenMP semantics, guaranteeing that each loop of this kind is DOALL. While this reduces the conservatism of the PDG, its performance still lags far behind GINO. The reason is that the PDG cannot capture the parallel semantics of `private`, `firstprivate`, and `lastprivate`. Without these, PDG-based compilers can fail to detect array reductions, allocate thread-private memory and initialize private copies. These results demonstrate that **the PS-PDG is strictly more expressive than the PDG**. By unifying compiler analyses and parallel semantics expressed by developers, the PS-PDG enables optimizations that the PDG alone cannot express or safely apply.

VI. Related Work

Previous work [37], [38], [42] proposed graph representations of the explicit parallelism encoded in a program to help developers understand their parallelization. These representations directly capture the parallel control flow encoded in the parallel program, in contrast, the PS-PDG captures the precise constraints of the parallel program decoupled from the encoded parallel execution plan. Other prior work [40], [1], [20], [32], [19] lowered the explicit parallelism into the IR of the compiler, introducing a new IR where the parallel execution plan can be encoded explicitly. Some prior work [18] analyzed parallel IRs that capture simple fork-join models to remove dependences from the PDG generated by compiler analyses (to unblock vectorization), but they do not handle semantics beyond simple fork-join. Finally, HPVM [22] is designed specifically for heterogeneous hardware to enable optimizations while still maintaining performance portability. The PS-PDG is orthogonal to HPVM as it does not target heterogeneous hardware via a hierarchical dataflow graph or enable optimizations on the graph.

The Galois System [23] and the Kinetic Dependence Graph [16] targeted implicit parallelism in imperative languages. These approaches focus on irregular programs exploiting amorphous data-parallelism to improve performance by dynamically modifying computation task graphs at runtime.

Many functional programming languages represent parallelism either implicitly or explicitly through annotations or parallel constructs in the language itself (e.g., map) [8], [12], [47], [15], [6], [9], [24], [30], [35], [14], [36], [41], [21]. These works directly translated the parallelism into a single or a few predetermined parallel execution plans (usually based on task or fork-join parallelism), where the runtime system is left with few decisions to make (e.g., number of threads).

VII. Conclusion

This paper introduces PS-PDG, a novel abstraction that unifies compiler analyses with developer-expressed parallel semantics. Our PS-PDG-empowered optimizing compiler GINO enables powerful parallel execution plan optimization going beyond those manually encoded by developers. Compared to a state-of-the-art PDG-based optimizing compiler, PS-PDG consistently delivers improved performance, demonstrating that PDG alone is limited for parallel optimization. Together, these results establish PS-PDG as a practical and effective abstraction for future optimizing compilers.

Acknowledgements & Data Statement

We thank members of the ARCANA Lab for their support and feedback on this work. We also thank the anonymous reviewers for their insightful comments and feedback, which made this work stronger. This effort is based upon work supported by the National Science Foundation under Grants NSF-2119069, NSF-2148177, NSF-2107042, and NSF-2107257. Additional data related to this publication may be found in the repository at 10.5281/zenodo.17633089 [43].

APPENDIX

A. Abstract

Our artifact includes the source files for the GINO compiler, as described in the paper. This artifact also includes the source code for all benchmarks evaluated in the paper. Furthermore, it includes an automated workflow to set up and run all experiments after building/running the Docker image/container. The output of this artifact generates all evaluation figures included in the paper, plus the raw results needed to generate the figures. The artifact also provides a `README` file detailing the steps to follow to reproduce the evaluation results.

B. Artifact check-list (meta-information)

- **Algorithm: GINO Compiler.**
- **Program: NAS benchmark suite.**
- **Compilation: LLVM 14.0.6. The Dockerfile handles the installation of this dependency.**
- **Transformations: None.**
- **Data set: Class B for BT and FT, the rest of the NAS benchmarks use class C.**
- **Run-time environment: Linux**
- **Hardware: Intel x86_64 CPU**
- **Metrics: Benchmark execution time**
- **Output: All experimental results and evaluation figures from Figure 10 to Figure 13. The artifact also includes results and figures from various systems.**
- **Experiments: Setting up the artifact and running experiments both use an automated workflow.**
- **How much disk space required (approximately)?: 10 GB**
- **How much time is needed to prepare workflow (approximately)?: 20 minutes**
- **How much time is needed to complete experiments (approximately)?: 30 hours (using the default configuration**
- **Publicly available?: Yes**
- **Code licenses (if publicly available)?: MIT License**
- **Workflow framework used?: Docker, Unix Makefiles, and Bash**
- **Archived?: https://doi.org/10.5281/zenodo.17633089**

C. Description

1) How to access: The artifact can be accessed and downloaded from Zenodo, at https://doi.org/10.5281/zenodo.17633089.

2) Hardware dependencies: 32+ physical cores x86_64 CPU and 16+ GB RAM. This artifact has been tested on Intel processors, including Intel Xeon Gold 6258R and Silver 4116.

3) Software dependencies: Running the artifact requires LLVM 14.0.6, GLLVM and OpenCilk-2.0. The dependency is handled when building with the Docker image. This artifact has been tested on Docker 27.5.1.

4) Data sets: The NAS benchmark source code is included in this artifact. Benchmark BT and FT use input size B ,and the rest of the benchmarks use class C.

D. Installation

Please run the artifact inside a Docker container. A Dockerfile is provided to handle the installation of all dependencies. First, please download and extract the artifact. Then, from within the artifact, run the following commands:

```
docker build -t cgo26ae .
docker run --privileged -it cgo26ae
cd pspdg-cgo26-artifact && make setup
```

To verify the artifact is set up correctly, run the following command:

```
make test
```

If you see the text "The artifact sets up correctly, and all tests passed!", then the installation phase is complete.

E. Experiment workflow

The artifact includes a fully automatic workflow to generate all experimental results included in the paper. The workflow runs GINO to compile all benchmarks evaluated in the paper. Then, the generated binaries are invoked to generate the raw results (e.g., the execution times of multiple runs of a benchmark). Then, these raw results are used to generate the figures included in the paper.

F. Evaluation and expected results

After installing the artifact and passing all tests, please run

```
make
```

The `plots/current_machine` directory will be populated with the figures as the output of the experiment workflow. The raw results used to feed all figures can be found under `results/current_machine` directory.

Figure 10 (`fig10.pdf`) plots the speedup of 8 NAS benchmarks between OpenCilk, developer's original parallel execution plan and the GINO-improved execution plan.

Figure 11 (`fig11.pdf`) plots the speedup of the parallel binary of benchmark BT running on a single core between OpenMP and GINO.

Figure 12 (`fig12.pdf`) plots the performance between GINO's original parallel execution plan and OpenMP.

Figure 13 (`fig13.pdf`) plots the speedup of 8 NAS benchmarks between compilation NOELLE and GINO.

G. Experiment customization

The artifact can be configured during the setup stage. The configurable options are as follows:

- **number of runs**: specifies how many times to run the experiment per benchmark. The default value is 5.
- **number of workers**: specifies the number of threads to use when the generated binaries are evaluated on multiple cores. The default value is the number of physical cores detected on the underlying machine.

H. Notes

We assume you have an internet connection during the setup stage of this artifact to download all necessary repositories and external libraries. For more details on the running Docker commands and , please reference the `README` file.

I. Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html

REFERENCES

[1] LLVM/OpenMP design and overview, 2023.

[2] OMP dialect - mlir, 2024.

[3] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers: principles, techniques, & tools*. Pearson Education India, 2007.

[4] Sotiris Apostolakis, Ziyang Xu, Greg Chan, Simone Campanoni, and David I. August. Perspective: A sensible approach to speculative automatic parallelization. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 351–367, New York, NY, USA, 2020. Association for Computing Machinery.

[5] Sotiris Apostolakis, Ziyang Xu, Zujun Tan, Greg Chan, Simone Campanoni, and David I. August. SCAF: a speculation-aware collaborative dependence analysis framework. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 638–654, New York, NY, USA, 2020. Association for Computing Machinery.

[6] Arvind Arvind, Rishiyur Nikhil, and Keshav Pingali. I-Structures: Data structures for parallel computing. *ACM Trans. Program. Lang. Syst.*, 11:598–632, October 1989.

[7] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks—summary and preliminary results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, page 158–165, New York, NY, USA, 1991. Association for Computing Machinery.

[8] Guy Blelloch and John Greiner. Parallelism in sequential functional languages. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture - FPCA '95*, pages 226–237, La Jolla, California, United States, 1995. ACM Press.

[9] Guy E. Blelloch, Jonathan C. Hardwick, Siddhartha Chatterjee, Jay Sipelstein, and Marco Zagha. Implementation of a portable nested data-parallel language. *ACM SIGPLAN Notices*, 28(7):102–111, July 1993.

[10] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of parallel and distributed computing*, 37(1):55–69, 1996.

[11] Simone Campanoni, Timothy Jones, Glenn Holloway, Vijay Janapa Reddi, Gu-Yeon Wei, and David Brooks. HELIX: Automatic parallelization of irregular programs for chip multiprocessing. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 84–93, New York, NY, USA, 2012. ACM.

[12] Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. Implicitly-threaded parallelism in Manticore. page 12.

[13] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. *SIGPLAN Not.*, 33(5):212–223, May 1998.

[14] Adrien Guatto, Sam Westrick, Ram Raghunathan, Umut Acar, and Matthew Fluet. Hierarchical memory management for mutable state. *ACM SIGPLAN Notices*, 53(1):81–93, February 2018.

[15] Robert H. Halstead. Implementation of multilisp: Lisp on a multiprocessor. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, LFP '84, pages 9–17, New York, NY, USA, August 1984. Association for Computing Machinery.

[16] Muhammad Amber Hassaan, Donald D. Nguyen, and Keshav K. Pingali. Kinetic dependence graphs. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, page 457–471, New York, NY, USA, 2015. Association for Computing Machinery.

[17] Ali R Hurson, Joford T Lim, Krishna M Kavi, and Ben Lee. Parallelization of DOALL and DOACROSS loops—a survey. In *Advances in computers*, volume 45, pages 53–103. Elsevier, 1997.

[18] Nicklas Bo Jensen and Sven Karlsson. Improving loop dependence analysis. *ACM Trans. Archit. Code Optim.*, 14(3), aug 2017.

[19] Richard Johnson, David Pearson, and Keshav Pingali. The program structure tree: Computing control regions in linear time. *SIGPLAN Not.*, 29(6):171–185, jun 1994.

[20] Herbert Jordan, Simone Pellegrini, Peter Thoman, Klaus Kofler, and Thomas Fahringer. INSPIRE: The insieme parallel intermediate representation. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, pages 7–17, 2013.

[21] Ulrike Klusik, Rita Loogen, Steffen Priebe, and Fernando Rubio. Implementation skeletons in eden: Low-effort parallel programming. Lecture Notes in Computer Science, page 71–88, Berlin, Heidelberg, 2001. Springer.

[22] Maria Kotsifakou, Prakalp Srivastava, Matthew D. Sinclair, Rakesh Komuravelli, Vikram Adve, and Sarita Adve. HPVM: heterogeneous parallel virtual machine. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '18, page 68–80, New York, NY, USA, 2018. Association for Computing Machinery.

[23] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, page 211–222, New York, NY, USA, 2007. Association for Computing Machinery.

[24] Peng Li, Simon Marlow, Simon Peyton Jones, and Andrew Tolmach. Lightweight concurrency primitives for GHC. In *Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, Haskell '07, pages 107–118, New York, NY, USA, September 2007. Association for Computing Machinery.

[25] LLVM Project. A Compiler's View of OpenMP. https://www.openmp.org/wp-content/uploads/A-compilers-view-of-OpenMP.pdf.

[26] LLVM Project. AliasAnalysis Class Reference. https://mlir.llvm.org/doxygen/classmlir_1_1AliasAnalysis.html.

[27] LLVM Project. Auto-Vectorization in LLVM. https://llvm.org/docs/Vectorizers.html.

[28] LLVM Project. MLIR Language Reference. https://mlir.llvm.org/docs/LangRef/.

[29] LLVM Project. Side effects & speculation. https://mlir.llvm.org/docs/Rationale/SideEffectsAndSpeculation/.

[30] Simon Marlow. Parallel and concurrent programming in haskell. In Viktória Zsók, Zoltán Horváth, and Rinus Plasmeijer, editors, *Central European Functional Programming School: 4th Summer School, CEFP 2011, Budapest, Hungary, June 14-24, 2011, Revised Selected Papers*, Lecture Notes in Computer Science, pages 339–401. Springer, Berlin, Heidelberg, 2012.

[31] 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. NOELLE offers empowering LLVM extensions. In *Proceedings of the 20th IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '22, page 179–192. IEEE Press, 2022.

[32] V. Krishna Nandivada, Jun Shirako, Jisheng Zhao, and Vivek Sarkar. A transformation framework for optimizing task-parallel programs. *ACM Trans. Program. Lang. Syst.*, 35(1), apr 2013.

[33] Omni Compiler Project. NAS-C-OpenMP3.0, 2014. https://benchmark-subsetting.github.io/cNPB/.

[34] OpenMP Architecture Review Board. OpenMP application program interface version 5.0, November 2018.

[35] Simon Peyton Jones, Roman Leshchinskiy, Gabriele Keller, and Manuel Chakravarty. Harnessing the multicores: Nested data parallelism in Haskell. *Leibniz International Proceedings in Informatics, LIPIcs*, 2, December 2008.

[36] Ram Raghunathan, Stefan K. Muller, Umut A. Acar, and Guy Blelloch. Hierarchical memory management for parallel programs. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 392–406, New York, NY, USA, September 2016. Association for Computing Machinery.

[37] Vivek Sarkar. Analysis and optimization of explicitly parallel programs using the parallel program graph representation. In *LCPC*, 1997.

[38] Vivek Sarkar and Barbara Simons. Parallel program graphs and their classification. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, page 633–655, Berlin, Heidelberg, 1993. Springer-Verlag.

[39] Tao B. Schardl and I-Ting Angelina Lee. Opencilk: A modular and extensible software infrastructure for fast task-parallel code. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, PPoPP '23, page 189–203, New York, NY, USA, 2023. Association for Computing Machinery.

[40] Tao B. Schardl, William S. Moses, and Charles E. Leiserson. 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*, PPoPP '17, page 249–265, New York, NY, USA, 2017. Association for Computing Machinery.

[41] K. C. Sivaramakrishnan, Lukasz Ziarek, and Suresh Jagannathan. MultiMLton: A multicore-aware runtime for standard ML. *Journal of Functional Programming*, 24(6):613–674, November 2014. Publisher: Cambridge University Press.

[42] Harini Srinivasan and Michael Wolfe. Analyzing programs with explicit parallelism. In *Proceedings of the Fourth International Workshop on Languages and Compilers for Parallel Computing*, page 405–419, Berlin, Heidelberg, 1991. Springer-Verlag.

[43] Yian Su. The parallel-semantics program dependence graph for parallel optimization, November 2025. https://doi.org/10.5281/zenodo.17633089.

[44] Yian Su, Mike Rainey, Nick Wanninger, Nadharm Dhiantravan, Jasper Liang, Umut A. Acar, Peter Dinda, and Simone Campanoni. Compiling loop-based nested parallelism for irregular workloads. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '24, page 232–250, New York, NY, USA, 2024. Association for Computing Machinery.

[45] Neil Vachharajani, Ram Rangan, Easwaran Raman, Matthew J Bridges, Guilherme Ottoni, and David I August. Speculative decoupled software pipelining. In *16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*, pages 49–59. IEEE, 2007.

[46] Hans Vandierendonck, Sean Rul, and Koen De Bosschere. The paralax infrastructure: automatic parallelization with a helping hand. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, page 389–400, New York, NY, USA, 2010. Association for Computing Machinery.

[47] Sam Westrick, Rohan Yadav, Matthew Fluet, and Umut A. Acar. Disentanglement in nested-parallel programs. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–32, January 2020.