The Parallel Semantics Program Dependence Graph

BRIAN HOMERDING, Northwestern University, USA
ATMN PATEL, Northwestern University, USA
ENRICO ARMENIO DEIANA, Northwestern University, USA
YIAN SU, Northwestern University, USA
ZUJUN TAN, Princeton University, USA
ZIYANG XU, Princeton University, USA
BHARGAV REDDY GODALA, Princeton University, USA
DAVID I. AUGUST, Princeton University, USA
SIMONE CAMPANONI, Northwestern University, USA

A compiler's *intermediate representation (IR)* defines a program's execution plan by encoding its instructions and their relative order. Compiler optimizations aim to replace a given execution plan (which instructions to execute and when) with a semantically-equivalent one that increases the program's performance for the target architecture. Alternative representations of an IR, like the Program Dependence Graph (PDG), aid this process by capturing the minimum set of constraints that semantically-equivalent execution plans must satisfy. Parallel programming like OpenMP extends a sequential execution plan by adding the possibility of running instructions in parallel, creating a parallel execution plan. Recently introduced parallel IRs, like TAPIR, explicitly encode a parallel execution plan. These new IRs finally make it possible for compilers to change the parallel execution plan expressed by programmers to better fit the target parallel architecture. Unfortunately, parallel IRs do not help compilers in identifying the set of parallel execution plans that preserve the original semantics. In other words, we are still lacking an alternative representation of parallel IRs to capture the minimum set of constraints that parallel execution plans must satisfy to be semantically-equivalent. Unfortunately, the PDG is not an ideal candidate for this task as it was designed for sequential code. In more detail, this paper shows that the PDG over-constrains the optimization space when used for parallel code. We propose the Parallel Semantics Program Dependence Graph (PS-PDG) to precisely capture the salient program constraints that all semantically-equivalent parallel execution plans (and therefore parallel IRs) must satisfy. This paper defines the PS-PDG, justifies the necessity of each extension to the PDG, and demonstrates the increased optimization power of the PS-PDG over an existing PDG-based automatic-parallelizing compiler. Compilers can now rely on the PS-PDG to select different parallel execution plans while maintaining the same original semantics.

1 INTRODUCTION

A compiler's *intermediate representation (IR)* defines a program's execution plan by encoding its instructions and their relative order. Most compiler optimizations are performed by changing the IR of a program. Many of these optimizations (e.g., code scheduling) need to change the relative order of IR instructions (i.e., the execution plan) to reach their optimization goal and/or to better target the underlying architecture. This is possible because not all orders specified by a given IR instance are necessary (thanks to having independent instructions). It is therefore important to understand what is the minimum *subset* of instruction order constraints that must be enforced to preserve the original semantics. Unfortunately, IRs (e.g., LLVM IR) do not highlight such subset; instead, a specific instance of an IR specifies the *total* order of its instructions, but some of these orders are the result of a choice (e.g., execution order of independent instructions within a basic block) rather than a constraint that must be satisfied. To overcome this limitation, code optimizations rely on a different representation of the IR code called the *Program Dependence Graph* (PDG) [20], which encodes the (ideally small) set of order constraints that all possible total order of instructions (hence, IR instances) must satisfy to preserve the original semantics of the input code. The PDG can alternatively be seen as the (ideally large) set of degrees of freedom that a code optimizer can leverage to generate an IR instance that better fits the target architecture.

The typical compilation pipeline followed by a code optimization is shown in Figure 1. A given IR instance is analyzed by a dependence analysis to generate its PDG [1]. Then, a code optimization decides the order of instructions that are left unspecified by the PDG (those that can execute in multiple orders). These decisions are then enforced by generating a new IR instance with (potentially) a different total order compared to the one given as input. In other words, a code optimization changes the execution plan of an IR by relying on the degrees of freedom highlighted by its PDG representation.

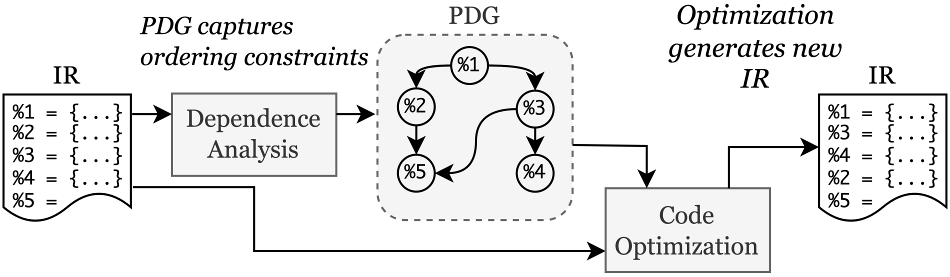


Fig. 1. Compiler pipeline utilizing IR and PDG for code optimization.

For decades, compilers have used sequential IRs (IRs that assume sequential execution; e.g., LLVM IR, GCC RTL) and PDG to compile and optimize both sequential and parallel programs. Parallel code extends the execution plan of sequential code by adding the choice of which instructions to run in parallel with respect to what. Parallel language compilers (e.g., OpenMP compilers) generally represent the parallel aspects of the execution plan (e.g., run iterations of a given loop in parallel) with annotations, built-ins, or calls added to their IRs. These constructs enable sequential optimizations to proceed correctly without changing the parallel aspects of the execution plan. This is obtained by outlining parallel regions into functions so that the sequential interpretation of the resulting code is valid (even if overly constrained). This design decision handcuffed compilers to maintain the parallel aspects of the execution plan that were encoded by programmers. This is the reason why OpenMP and Cilk compilers (for example) do not alter the decision of what to run in parallel that was encoded by programmers within their source code.

While sequential IRs and their PDG made sense when compilers were not expected to include code optimizations specifically designed for parallel code, they are insufficient for the next generation of compilers that include parallel optimizations. This is because parallel optimizations require modifying parallel aspects of the input code. In more detail, as parallel machines proliferate, the community frequently observed that the parallel execution plan expressed by programmers is sub-optimal for the target architecture [9, 16, 17, 22, 42, 45, 48]. This is because different architectures require different parallel execution plans to reach high performance. In other words, just as the sequential execution plans specified by programmers are sub-optimal [13, 18, 19, 34, 36, 47, 54], so too are the parallel execution plans found in manually-written parallel programs. To address this, the community has introduced parallel IRs like TAPIR [2, 30, 41, 49–51, 53] to represent parallel execution plans and changes thereto. Explicit encoding of a specific parallel execution plan into a parallel IR made changes to it finally possible. Unfortunately, changing a given parallel execution

[1] Some code optimizations target small code regions like loops. Some compilers therefore generate only the subset of the PDG that is needed by the selected code optimizations.

plan is still not practical because a given instance of a parallel IR does not highlight the minimum set of constraints that all parallel execution plans must satisfy to preserve the original semantics. Similarly to the need to have a separate representation for sequential IRs to enable code optimizers to change the sequential execution plan (i.e., the total order of instructions), we need a separate representation of parallel IRs to enable parallel code optimizers *to change the parallel execution plan* (i.e., what to run in parallel and how). Unfortunately, the PDG is not a good representation for this goal because it was designed to target sequential IRs, which blocks it from encoding all degrees of freedom that parallel code optimizers could take advantage of. In other words, the PDG is a sub-optimal representation for parallel IRs and our empirical results (§6) clearly show that its limitations for parallel IRs are too significant to be ignored.

To overcome this limitation, this paper proposes the Parallel Semantics Program Dependence Graph (PS-PDG) representation to capture the salient program constraints of parallel programs. The PS-PDG generalizes the PDG to capture the constraints that define the set of semantically equivalent execution plans for modern parallel programming models such as OpenMP and Cilk. After defining the PS-PDG, this paper shows how each of its extensions to the PDG is necessary.

To reduce disruptions to existing compilers, we propose a PS-PDG-based compilation pipeline (shown in Figure 2 and described next) that is similar to what has been in use for decades. Our pipeline substitutes the PDG representation with the PS-PDG one. We believe this is the missing piece to enable parallel compilers to reach their full potential. In more detail, in a compiler developed with the PS-PDG, Cilk[12] and OpenMP[43] source code is first translated into their parallel IR while preserving the execution plan expressed by the programmers. Then, the IR is analyzed to generate a PS-PDG, which captures the minimum constraints necessary to preserve the original semantics. With the PS-PDG, the compiler is finally capable to identify all possible parallel execution plans that are guaranteed to preserve the original semantics of the input code. Hence, parallel compilers can now find the most appropriate parallel execution plan for the target architecture. The parallel execution plan chosen is then realized into the generated parallel IR. The parallel IR is then translated into the target assembly code.

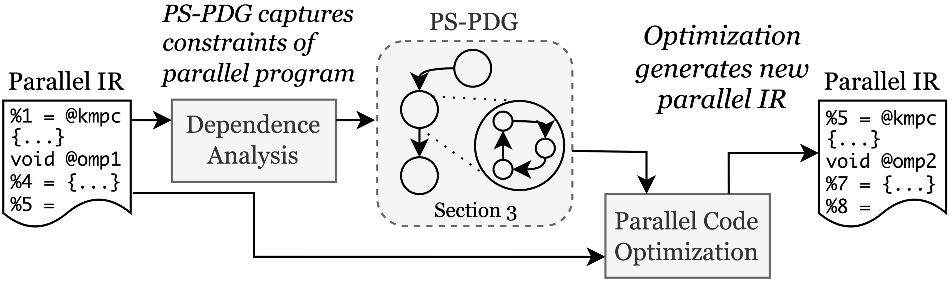


Fig. 2. Compiler pipeline utilizing parallel IR and PS-PDG for parallel code optimization.

The main contributions of this work are:

- A definition of the PS-PDG: a representation that captures the salient program constraints of parallel programs (§3).
- A detailed analysis of the necessity of each element of the PS-PDG representation (§4).
- A demonstration that the PS-PDG is sufficient to fully capture the parallel semantics of OpenMP programs (§5).
- The first compiler to generate the PS-PDG for existing OpenMP benchmarks (§6.1).

- A demonstration of PS-PDG's ability to expand the options available to an automatic-parallelizing compiler when selecting a parallelization plan. We compare this to the PDG and source code parallelization plan (§6.2).
- A measurement of the reduction in the critical path on an ideal machine using existing automatic-parallelizing compiler techniques with the PS-PDG. We compare this to the PDG and source code parallelization plan (§6.3).

2 BACKGROUND & MOTIVATION

Programmers need to express well-tuned parallelism in their applications to achieve high performance and energy efficiency. This requires rich parallel programming models (PPMs) so that programmers can express complex parallel execution plans for their applications. This has led to an ever-increasing number of PPMs and features therein. Two examples of widely-adopted PPMs are OpenMP and Cilk. This section uses an OpenMP code example to demonstrate how a programmer can explicitly encode a parallel execution plan. Then, the same example serves as motivation for using the PS-PDG to represent the precise parallel constraints of a parallel program for use in parallel execution plan transformations.

2.1 Programmers Explicitly Encode Parallelism

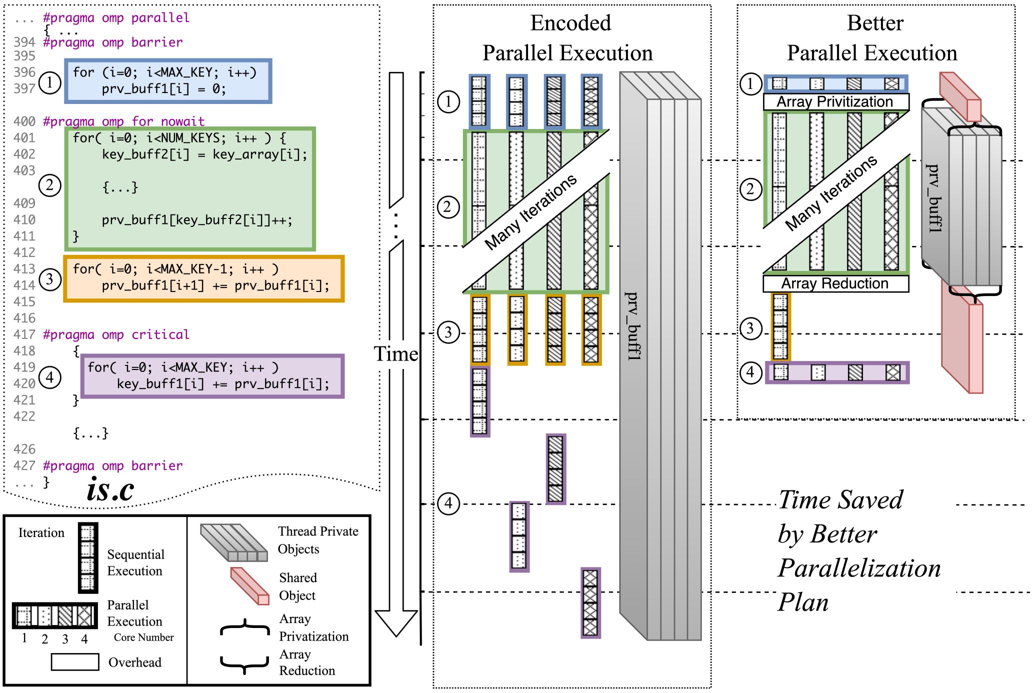
The OpenMP PPM allows programmers to parallelize their code with pragmas. An example is `#pragma omp parallel for` which specifies that the iterations of the annotated loop can execute in parallel. In this pragma, the worksharing pragma `omp for` specifies that the iterations of the loop should be distributed across multiple threads. Other pragmas offer greater control over the parallel execution, such as `critical` which specifies that the given code region should only be executed by a single thread at any given time.

Fig. 3 shows the OpenMP source code from the hottest computation kernel of the `IS` benchmark from the `NAS` benchmark suite [8], along with the execution of its encoded parallelization plan. The entire kernel is within a `#pragma omp parallel` which spawns many threads. Since the loops ① (blue) and ③ (orange), do not have a worksharing pragma, each thread executes the entire loop on its private copy of `prv_buff1`. Loop ② (green) instead has its iterations running in parallel between threads. Loop ④ (purple) is wrapped in a `critical` section to avoid a data race while updating `key_buff1` concurrently.

The code of Fig. 3 is an example of the OpenMP PPM where the programmer has encoded a specific parallelization plan into the application. A *parallelization plan* is the selection of what to parallelize (e.g., which loops) along with enabling features (e.g., which variables are thread-private), combined with the chosen parallel execution model (e.g., tasks, threads). Beyond the explicit parallelization encoded, the parallelization plan implies properties of the original code.

2.2 Enabling Compilers to Optimize Parallel Code

Let us re-consider the hot code of `IS` shown in Fig. 3, but with a different parallelization plan than the one encoded by its OpenMP implementation, shown on the right side of Fig. 3. Now iterations of loop ① execute in parallel while accessing different slices on a shared copy of `prv_buff1` (the original plan had loop ① accessing only the thread-private copies of this array). We divide the iterations of loop ① across many threads that access different slices of the single shared array `prv_buff1`. Then, we perform an array privatization of `prv_buff1` before executing loop ②. Iterations of loop ② execute between threads as in the original plan. We then reduce the private copies of `prv_buff1` to their shared copy as soon as loop ② ends its parallel execution. Therefore, loop ③ will only need to execute on a single thread (avoiding its parallel overhead that the original plan had). As there is only one shared copy of `prv_buff1` at this point of execution, loop ④ can now execute in



```
... #pragma omp parallel
{ ...
#pragma omp barrier

for (i=0; i<MAX_KEY; i++)
    prv_buff1[i] = 0;

#pragma omp for nowait
for( i=0; i<NUM_KEYS; i++ ) {
    key_buff2[i] = key_array[i];

    {...}

    prv_buff1[key_buff2[i]]++;
}

for( i=0; i<MAX_KEY-1; i++ )
    prv_buff1[i+1] += prv_buff1[i];


#pragma omp critical
    {
     for( i=0; i<MAX_KEY; i++ )
        key_buff1[i] += prv_buff1[i];
    }

    {...}

#pragma omp barrier
... }
```

Fig. 3. The key computational kernel from the IS benchmark with the original and a more performant compiler-selected parallel execution.

parallel (the original plan executed the loop sequentially between threads) by dividing its iterations between threads (without any critical section). This new parallelization plan's execution is shown on the right side of Fig. 3.

Consider the transformations required to change the parallelization plan on the left to the parallelization plan on the right of Fig. 3. The private buffer prv_buff1 of loop ② must first be recognized as private from the IR explicit representation, then its uses beyond the loop must be considered. The updates to prv_buff1 in both loops ② and ③ must be recognized as reducible in both uses before either can be transformed. Furthermore, the compiler must leverage the developer knowledge that the various arrays do not alias with one another and that the indirect index into prv_buff1 does not go beyond the use of the prv_buff1 in the other loops.

This example suggests that compilers need to become capable of modifying the parallelization plan expressed by programmers. To do so, compilers must be equipped with the correct abstraction that captures the precise parallel constraints of a parallel program. Today's compilers of PPMs like OpenMP and Cilk cannot do it. The compilation pipeline of these compilers look like the one shown on the left of Fig. 4. As the source code is processed by the compiler, the parallelization plan is simply lowered to the runtime calls that implement it; these compilers do not have an abstraction that captures the precise parallel constraints of the parallel program being compiled. To empower compilers to transform the parallelization plan to better utilize the underlying hardware, we need to change the compiler's internal abstractions.

To perform transformations such as the one described above, the compiler needs to implement the transformed parallelization. Today's automatic parallelizing compilers [5, 6, 14, 40, 55] successfully use the PDG abstraction, but they cannot rely on the parallel semantics expressed by programmers

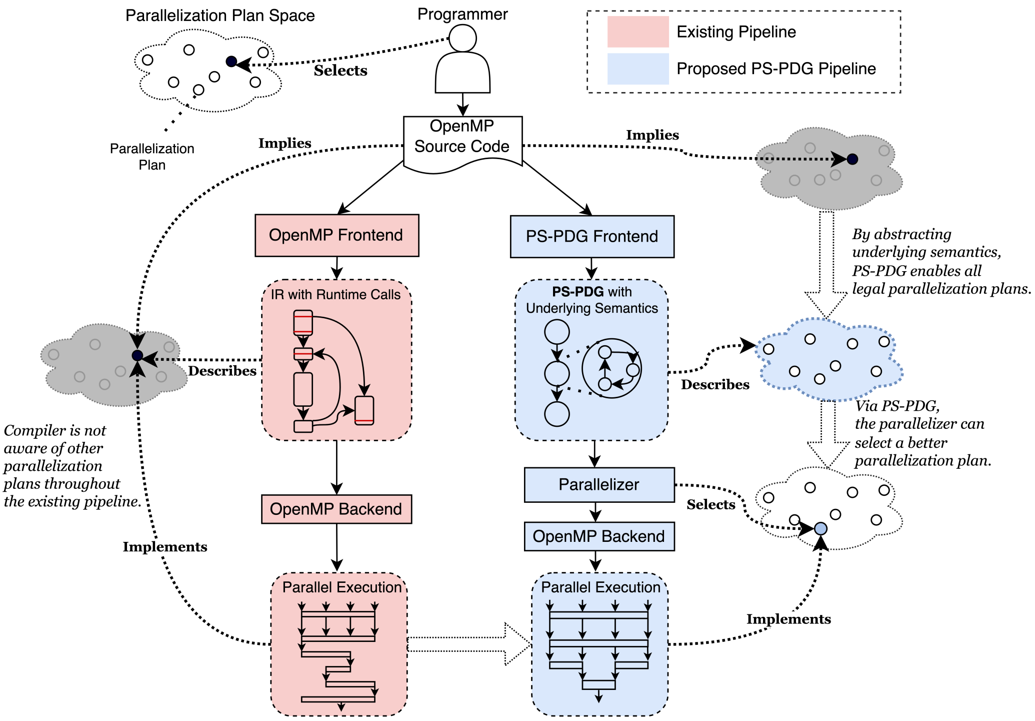

Parallelization Plan Space
Programmer
Existing Pipeline
Proposed PS-PDG Pipeline
Selects
Parallelization Plan
Implies
OpenMP Source Code
Implies
OpenMP Frontend
PS-PDG Frontend
By abstracting underlying semantics, PS-PDG enables all legal parallelization plans.
IR with Runtime Calls
PS-PDG with Underlying Semantics
Describes
Describes
Compiler is not aware of other parallelization plans throughout the existing pipeline.
Via PS-PDG, the parallelizer can select a better parallelization plan.
Parallelizer
Selects
OpenMP Backend
OpenMP Backend
Implements
Parallel Execution
Parallel Execution
Implements

Fig. 4. Comparison of the existing pipeline with our proposed PS-PDG pipeline. The parallelization plan in the source code is abstracted away.

because the PDG does not capture it. To overcome this limitation, this paper proposes the PS-PDG, an abstraction capable of capturing the precise parallel constraints of parallel programs while decoupling it from the encoded parallel execution plan. With the PS-PDG, compilers can now explore the space of semantically equivalent parallelization plans, while preserving semantics, to fully leverage the precise parallel constraints of the parallel program.

The PS-PDG enables the pipeline shown in Fig. 4. The proposed pipeline does not rigidly follow the encoded parallelization plan (like today's compilers do). In this new pipeline, the precise parallel constraints of a parallel program is captured by the PS-PDG abstraction so that compilers can now see the space of semantically-equivalent parallelization plans. The middle-end is now capable of selecting the parallelization plan that best fits the underlying architecture.

3 PS-PDG DEFINITION

PPMs like OpenMP and Cilk enable programmers to make parallelization decisions explicit. A programmer can decide where to spawn threads or tasks, to distribute the computation of a loop between parallel threads and/or tasks, and how to synchronize their execution. These parallelization decisions define 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.

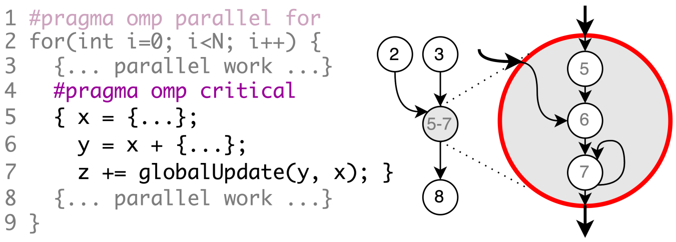
Table 1. Complete PS-PDG Definition

PS-PDG	::= (Node$^+$, Edge*, Variable*, VariableAccess*)
Node	::= (Instruction, Trait*) | (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*

The PS-PDG extends the PDG abstraction to capture the precise parallel constraints of a parallel OpenMP or Cilk program. Like the PDG, the PS-PDG has nodes to represent computation and edges to represent dependences within the computation, but it also includes variables to represent data and use/def edges to represent the relation between data and its computation. As shown in Table 1, a PS-PDG consists on one or more nodes with zero or more edges, variables and variable accesses. The rest of this Section will describe each extension in detail.

3.1 Hierarchical Nodes

Explicit parallel programming enables programmers to specify properties of a code region. Often such properties do not hold 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 (§3.2).



```
#pragma omp parallel for
for(int i=0; i<N; i++) {
  {... parallel work ...}
  #pragma omp critical
  { x = {...};
    y = x + {...};
    z += globalUpdate(y, x); }
  {... parallel work ...}
}
```

Fig. 5. Capturing properties of a region into hierarchical nodes with traits

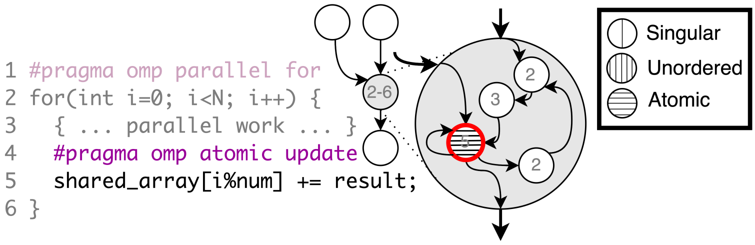
A node in the PS-PDG represents a non-empty set of instructions organizing the code hierarchically, shown in Fig. 5. For example, all instructions of a *critical* section in the PS-PDG is represented by a single node. More generally, a node N of the PS-PDG is a non-empty set of one or more instructions 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.

3.2 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. 6.

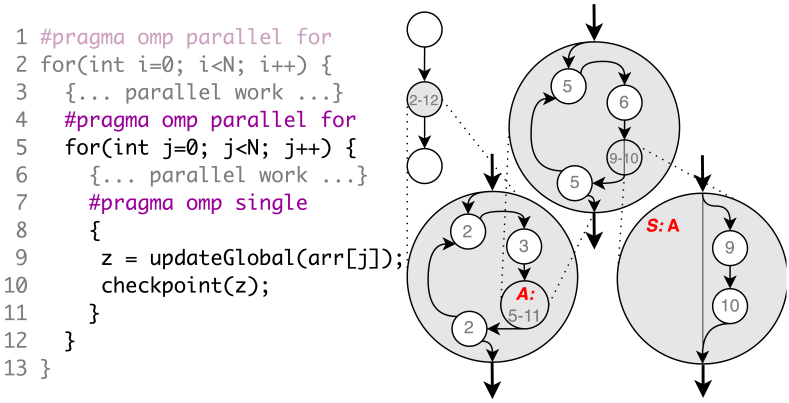


```
1 #pragma omp parallel for
2 for(int i=0; i<N; i++) {
3   { ... parallel work ... }
4   #pragma omp atomic update
5   shared_array[i%num] += result;
6 }
```

Fig. 6. How traits can be used to capture atomic updates.

3.3 Context

PPMs allow programmers 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



```
 1 #pragma omp parallel for
 2 for(int i=0; i<N; i++) {
 3   {... parallel work ...}
 4   #pragma omp parallel for
 5   for(int j=0; j<N; j++) {
 6     {... parallel work ...}
 7     #pragma omp single
 8     {
 9      z = updateGlobal(arr[j]);
10      checkpoint(z);
11     }
12   }
13 }
```

Fig. 7. Traits applied to context *A* captures the region's semantics.

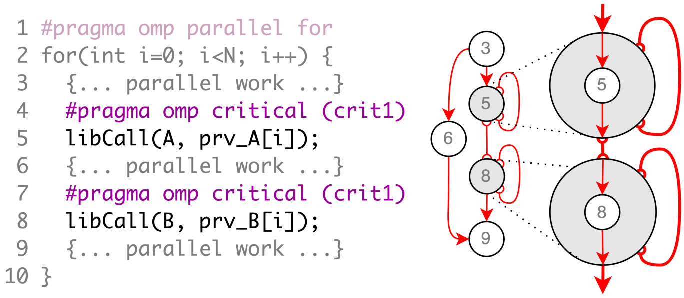
innermost loop that contains it. Because the contexts in which parallel semantics is valid cannot always be computed, the PS-PDG can specify contexts and their relation with parallel semantics.

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. 1. For example, the hierarchical node *S* of Fig. 7 that captures the *single* code section declares its semantics applies only to context A, which is its target loop.

3.4 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. 8).

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.



```
 1 #pragma omp parallel for
 2 for(int i=0; i<N; i++) {
 3   {... parallel work ...}
 4   #pragma omp critical (crit1)
 5   libCall(A, prv_A[i]);
 6   {... parallel work ...}
 7   #pragma omp critical (crit1)
 8   libCall(B, prv_B[i]);
 9   {... parallel work ...}
10 }
```

Fig. 8. Ordering constraints are captured with directed and undirected edges.

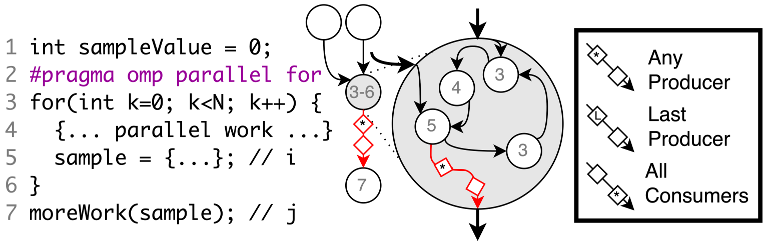
3.5 Data-Selector Directed Edge

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. 9 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, programmers can express richer semantics when developing a parallel program. For example, a programmer 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. So, 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.



```
1 int sampleValue = 0;
2 #pragma omp parallel for
3 for(int k=0; k<N; k++) {
4   {... parallel work ...}
5   sample = {...}; // i
6 }
7 moreWork(sample); // j
```

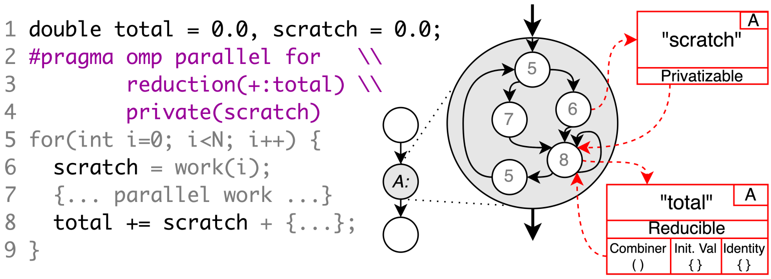
Fig. 9. Data-selector directed edges can capture non-trivial data relations.

3.6 Parallel Semantic Variables

Efficient parallel execution often requires programmers to express knowledge about the program's variables that go beyond their reads and writes and their data types. For example, programmers can express that a variable can be privatized in 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. 10.

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



```
1 double total = 0.0, scratch = 0.0;
2 #pragma omp parallel for   \\
3         reduction(+:total) \\
4         private(scratch)
5 for(int i=0; i<N; i++) {
6   scratch = work(i);
7   {... parallel work ...}
8   total += scratch + {...};
9 }
```

Fig. 10. Capturing programmer knowledge about data through privatizable and reducible parallel semantic variables.

chains [4]. 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.

4 THE NECESSITY OF EACH PS-PDG EXTENSION

This section demonstrates that each feature of the PS-PDG abstraction is necessary to capture the semantics expressible using the OpenMP programming model. The same result can be obtained similarly for Cilk. It does so for each PS-PDG extension by removing it from the proposed abstraction. This is done by showing two parallel programs that have different parallelization plans and semantics. By showing that these two programs translate to the same PS-PDG when the extension under evaluation is not available, we demonstrate that the feature is necessary. Additionally, this section provides an example that shows how each feature enables an important optimization. This shows the value of each PS-PDG extension.

4.1 Hierarchical Nodes and Undirected Edges

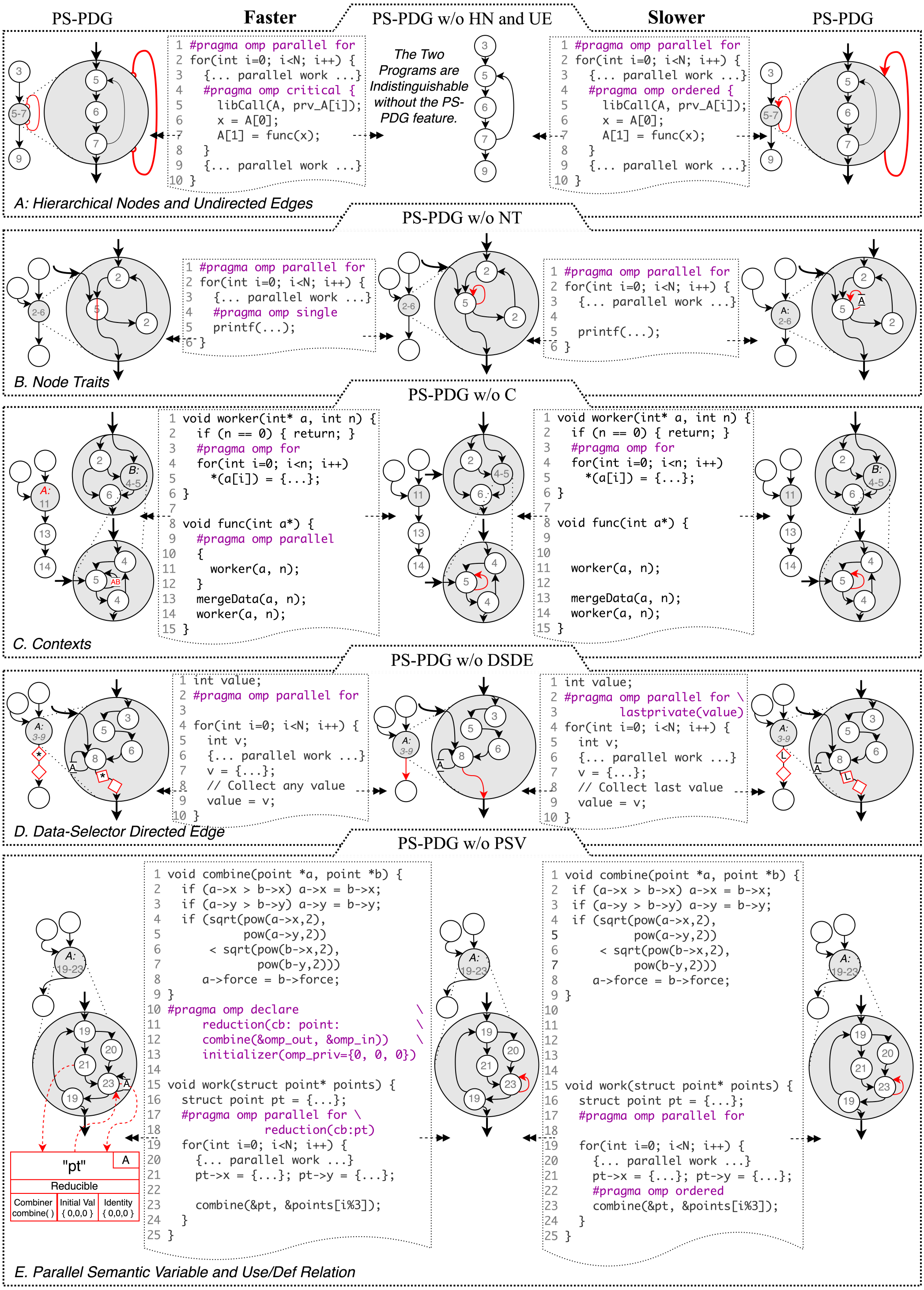
To understand the value of Hierarchical Nodes (HN) and Undirected Edges (UE), consider the two semantically different programs shown in Fig. 11-*A*. The program on the left requires avoiding overlapping dynamic instances of the critical section but puts no restriction on their order. In contrast, the 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 parallelization plan for each program different in this one 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. 11-*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.

4.2 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. 11-*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 parallelization plan. Unfortunately, this is not possible when using the PS-PDG without Node Traits (“PS-PDG w/o NT"). Fig. 11-*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".

4.3 Contexts

To understand the value of Contexts (C), consider the two programs shown in Fig. 11-*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



```
#pragma omp parallel for
for(int i=0; i<N; i++) {
  {... parallel work ...}
  #pragma omp critical {
    libCall(A, prv_A[i]);
    x = A[0];
    A[1] = func(x);
  }
  {... parallel work ...}
}
```

```
#pragma omp parallel for
for(int i=0; i<N; i++) {
  {... parallel work ...}
  #pragma omp ordered {
    libCall(A, prv_A[i]);
    x = A[0];
    A[1] = func(x);
  }
  {... parallel work ...}
}
```

```
#pragma omp parallel for
for(int i=0; i<N; i++) {
  {... parallel work ...}
  #pragma omp single
  printf(...);
}
```

```
#pragma omp parallel for
for(int i=0; i<N; i++) {
  {... parallel work ...}

  printf(...);
}
```

```
void worker(int* a, int n) {
  if (n == 0) { return; }
  #pragma omp for
  for(int i=0; i<n; i++)
    *(a[i]) = {...};
}

void func(int a*) {
  #pragma omp parallel
  {
    worker(a, n);
  }
  mergeData(a, n);
  worker(a, n);
}
```

```
void worker(int* a, int n) {
  if (n == 0) { return; }
  #pragma omp for
  for(int i=0; i<n; i++)
    *(a[i]) = {...};
}

void func(int a*) {


  worker(a, n);

  mergeData(a, n);
  worker(a, n);
}
```

```
int value;
#pragma omp parallel for

for(int i=0; i<N; i++) {
  int v;
  {... parallel work ...}
  v = {...};
  // Collect any value
  value = v;
}
```

```
int value;
#pragma omp parallel for \
          lastprivate(value)
for(int i=0; i<N; i++) {
  int v;
  {... parallel work ...}
  v = {...};
  // Collect last value
  value = v;
}
```

```
void combine(point *a, point *b) {
  if (a->x > b->x) a->x = b->x;
  if (a->y > b->y) a->y = b->y;
  if (sqrt(pow(a->x,2),
           pow(a->y,2))
      < sqrt(pow(b->x,2),
             pow(b-y,2)))
    a->force = b->force;
}
#pragma omp declare                  \
      reduction(cb: point:           \
      combine(&omp_out, &omp_in))    \
      initializer(omp_priv={0, 0, 0})

void work(struct point* points) {
  struct point pt = {...};
  #pragma omp parallel for \
             reduction(cb:pt)
  for(int i=0; i<N; i++) {
    {... parallel work ...}
    pt->x = {...}; pt->y = {...};

    combine(&pt, &points[i%3]);
  }
}
```

```
void combine(point *a, point *b) {
  if (a->x > b->x) a->x = b->x;
  if (a->y > b->y) a->y = b->y;
  if (sqrt(pow(a->x,2),
           pow(a->y,2))
      < sqrt(pow(b->x,2),
             pow(b-y,2)))
    a->force = b->force;
}






void work(struct point* points) {
  struct point pt = {...};
  #pragma omp parallel for

  for(int i=0; i<N; i++) {
    {... parallel work ...}
    pt->x = {...}; pt->y = {...};
    #pragma omp ordered
    combine(&pt, &points[i%3]);
  }
}
```

Fig. 11. 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).

significantly faster 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.

4.4 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. 11-*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. 11-*D* allows the propagation of the value generated by any loop iteration. The program on the left 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. 11-*D* map to the same "PS-PDG w/o DSDE". Thus, for correctness, the parallelization plan generated from the "PS-PDG w/o DSDE" must enforce the 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".

4.5 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. 11-*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. 11-*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 parallelization plan to apply to the program on the left of Fig. 11-*E* needs to be aware of the ability to privatize and reduce the struct `pt` to generate the best parallelization 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. 11-*E* map to the same "PS-PDG w/o PSV". This means that the parallelization plan generated from the "PS-PDG w/o PSV" must enforce the stricter semantics of the program on the right of Fig. 11-*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".

5 THE SUFFICIENCY OF THE PS-PDG FOR OPENMP

This section demonstrates that the PS-PDG abstraction is sufficient to capture the precise parallel constraints of the OpenMP programming model (Appendix A demonstrates it similarly for the Cilk language). We target the OpenMP 5.0 specification [43] with the exclusions of execution control, the target offload abstraction, runtime calls, and tooling support. We excluded these features because the goal is to enable compilers to select a parallel execution plan for general-purpose CPUs. So, we exclude features that only control the amount of parallelism to generate without adding semantics (e.g., deciding the number of threads to use for a given loop).

We group the parallel semantics expressible using the OpenMP 5.0 specification into three groups: declaration of independence, data properties, and computational ordering features. The OpenMP parallel semantics enabled by each group is mapped to a set of PS-PDG abstraction extensions. The parallel execution plan explicitly encoded by OpenMP programmer is within the parallelization plan space generated by a compiler using the PS-PDG. That is, the PS-PDG is sufficient to capture the precise parallel constraints of OpenMP

5.1 Declaration of Independence

Declaring the independence between code regions has a significant impact on the parallel semantics of the OpenMP programming language. This type of parallel semantics is the most used in OpenMP programs (empirically confirmed by well-established benchmark suites). The most typical example of this semantics is `omp for`, which declares the independence between loop iterations of all the code within the loop body not included in a critical section. Another example is `task`, which groups computation into multiple tasks, and their dependences (or lack of) are explicitly declared. Other examples with similar semantics are `taskloop`, `sections`, `simd`, and `workshare`; they all declare the existence of parallelism between code regions. Finally, OpenMP provides clauses that programmers use to declare when the declaration of parallelism (or dependence) is valid (in which contexts). For example, the programmer can declare that two code regions are independent only when executed within the context of a loop, but not when executed within the context of outer loops. These clauses include `barrier`, `flush`, `taskwait`, and `depobj`. These clauses, such as `barrier`, do not add additional information, rather they constrain the information provided by other clauses. This allows for the developer to encode synchronization in order to respect dependences which would instead be declared independent. The PS-PDG captures these constraints as dependeces.

The semantics encoded by all declarations of independence of the OpenMP language is captured by the PS-PDG abstraction extensions *hierarchical nodes* and *contexts*. Each code region targeted by an OpenMP pragma mentioned above (e.g., `task`) is mapped into a hierarchical node of the PS-PDG abstraction. Edges between hierarchical nodes (including between a node and itself) declare their dependences as specified by the OpenMP programmer. Finally, dependence edges include contexts in the PS-PDG abstraction to declare when they are valid (and therefore when they are not). This captures the precise parallel constraints of all declarations of independence that programmers can do using OpenMP.

For example, assume there is two-level nested loop where the inner loop has a call to a library function and that this call has a self-dependence only between iterations of the outer loop. An OpenMP programmer can parallelize this program by adding the pragma `for` to the inner loop. This semantics is mapped to PS-PDG creating a hierarchical node for the outer loop; hence, the outer loop becomes a context. Then, the dependence from the library call to itself includes the context of the outer loop, which declares that is valid only for the outer loop. A compiler can now generate the parallelization plan chosen by the OpenMP programmer using this PS-PDG as the inner loop has no loop-carried dependence and therefore can be parallelized using the parallelization plan selected by adding the pragma `for` to the inner loop.

5.2 Data and its Properties

The parallel semantics of data is essential for a PPM to be widely adopted. Parallel semantics of data allows programmers to declare properties of data, how to use them, and how to propagate them throughout the computation. An example of this parallel semantics is the `threadprivate` pragma, which declares that the data is attached to (e.g., an array) needs to be cloned such that threads use/define only their own private copy of it. Without access to privatization and reduction techniques the amount of expressible parallelism would be greatly limited. Therefore, the OpenMP model (like other PPMs) enables programmers to declare how the data can be privatized per-thread (`threadprivate`), how can the private copies be reduced at the end of a parallel code region (`reduction`), and which data needs to be propagated throughout the computation (`first/last private`).

The OpenMP semantics about privatizing data and reducing their private copies is captured by the *parallel semantics variable* of the PS-PDG abstraction. The PS-PDG variable declares its properties explicitly (e.g., per-thread private) and it declares how to reduce its private copies to a single one. The Use/Def relations included in PS-PDG declare how the code uses the related variable, and therefore when it needs to be privatized and when its private copies need to be reduced.

Finally, the OpenMP semantics about which data to propagate throughout the computation is captured by the *data selectors* of the PS-PDG abstraction. These selectors declare which data needs to propagate from a producer to its consumers.

5.3 Ordering

OpenMP programmers can express that two code regions are dependent, but their execution order is not important. This is more efficient than enforcing a pre-defined order. This semantics is expressed using the `critical` or `atomic` pragmas. The latter also imposes the need to execute atomically the code region wrapped in it. The PS-PDG *undirected edge* declares this lack of ordering between two interdependent code regions (or instructions), including self-dependences, and the OpenMP atomic semantic is captured by the *node trait* atomic.

6 EVALUATING THE PS-PDG ABSTRACTION

The PS-PDG abstraction captures the precise parallel constraints of a parallel program. The strength of an abstraction used within a compiler is in its ability to represent knowledge about a program not readily ascertainable from the IR. The value of a specific abstraction is in what it enables, and should be evaluated in this way. To this end, we evaluate the PS-PDG by what it enables for an *existing* automatic-parallelizing compiler rather than the end result of the parallelization. Notice that evaluating on the end result of a transformation would evaluate the transformation implemented using the PS-PDG, rather than the expressiveness of the PS-PDG itself. Hence, we evaluate the PS-PDG by performing two experiments that measure the power of the abstraction. The first experiment measures the size of the PS-PDG enabled expansion of options for the parallelizing compiler. The second experiment characterizes a bound on the potential of the parallel execution plans that the PS-PDG exposed. We find the PS-PDG enabled compiler has a richer set of options when determining a parallel execution plan and exposes significantly better parallel execution plans. Both experiments use a parallelizing compiler enhanced to use the PS-PDG in place of the PDG. We utilize the entire `NAS` Benchmark Suite [8] with class C inputs, with two exceptions (`BT`,`FT`: class B) due to gigabyte-size static variables.

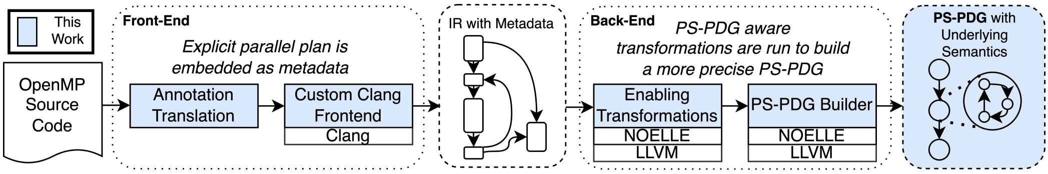


Fig. 12. Our custom compilation pipeline to generate the PS-PDG for a given OpenMP program.

6.1 Implementing the PS-PDG in an existing compiler

To perform the two experiments described above, we implemented a custom compilation pipeline (Fig. 12). This pipeline is built upon the NOELLE [40] compilation framework, which extends LLVM [35]. NOELLE provides an automatic-parallelizing compiler working at the IR level, which supports three loop-based parallelization techniques: DOALL [26], DSWP [55], HELIX [14, 15]. First, we added a tool that translates OpenMP annotations into pragmas that are amenable to being lowered by our custom front-end. Then, our custom clang-based front-end that generates LLVM IR with custom metadata from these pragmas. This IR with metadata feeds a series of code transformations that are designed to make the code more amenable to parallelization while maintaining the metadata for the precise parallel constraints of the parallel program. Finally, the resulting IR with metadata is used to build the PS-PDG. The PS-PDG is then used by our extensions of NOELLE's compiler, which originally used the PDG.

Both our parallelizing compiler and NOELLE's original one consider the parallelization of each loop with at least 1% run-time coverage. The subset of a dependence graph (PS-PDG or PDG) for a given loop is analyzed to identify strongly-connected components (SCC) with loop-carried dependences. For these SCCs, we utilize any PS-PDG features within the SCC to determine if the loop-carried dependences can be removed (e.g., privatization). If a loop can be parallelized as DOALL (i.e., no loop-carried dependences with a known trip count), then it is only considered as DOALL. For non-DOALL loops, the compiler considers HELIX and DSWP.

6.2 The PS-PDG gives the compiler more choices

To understand the potential added by PS-PDG, we automatically enumerate the options our compiler considers when determining a parallel execution plan of a loop. Then, we compare it with those that the PDG-based compiler included in NOELLE has. The PDG-based compiler utilizes the sequential version of the benchmarks. Additionally, we include the results from utilizing the OpenMP worksharing loop information for improved loop dependence analysis as in [28], labeled as "J&K". This approach enables the worksharing loop information to remove loop-carried dependences in the PDG. Finally, we include the number of corresponding options available to the source code OpenMP parallelization through environment variables.

We automatically enumerate the options for a 56 core machine while following the existing parallelization process in the compiler. For DOALL loops, the number of options is at most 56 (cores) × 8 (chunk sizes considered). For non-DOALL loops, the SCCs of a loop are categorized as sequential or parallel. The options available to HELIX is the possible number of sequential segments of that loop (a sequential segment is a slice of the loop that includes at least one sequential SCC). Furthermore, we consider running these sequential segments in parallel up to 56 cores. The options available to DSWP is the number of pipeline stages (each stage has at least one SCC) up to 56 cores.

Fig. 13 shows the number of options available to the compiler. The PS-PDG enables the automatic-parallelizing compiler to explore more options than the PDG, resulting in a large increase in the number of parallelization plans when combined. The compiler is able to consider all loops which

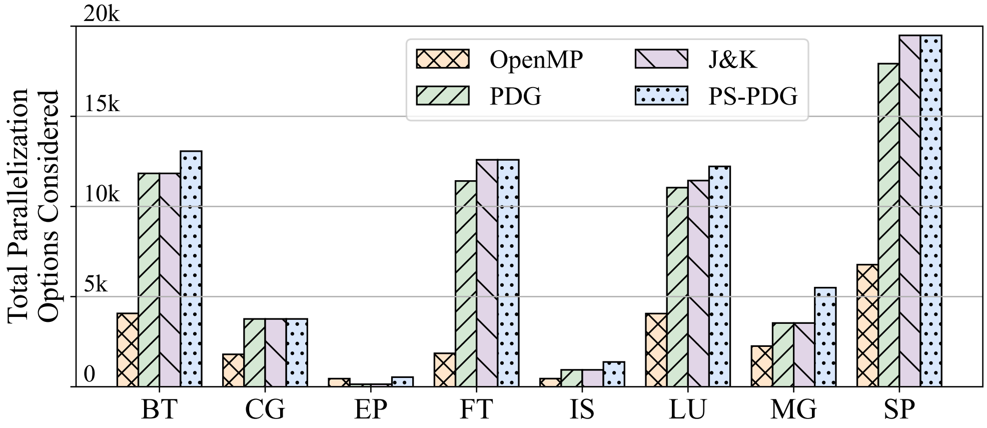


Fig. 13. Number of parallelization options available to the compiler.

meet the parallelization requirements while the programmer-encoded parallelization is static, the PS-PDG pushes beyond the limitations of the PDG-based compiler. For benchmarks with few loops which are parallelized well by the programmer (e.g., EP), the increase in options stays low. Additionally, we find that utilizing the PDG with workshare improved loop dependence analysis is insufficient to match the PS-PDG, as seen in the MG benchmark. Through the PS-PDG, the compiler can leverage the precise parallel constraints of the explicit parallelism, while leveraging compiler analysis for a larger space of parallelization plans.

6.3 The parallelization potential exposed by the PS-PDG

Next we evaluate the quality of the additional parallelization plans these options create. To evaluate the potential of the parallelization plans, we measure, via an emulator, the critical path of the available parallelism on an ideal machine with unlimited cores, zero cost communication, and perfect memory access. This enables us to characterize the limit of the additional parallelism unlocked by PS-PDG. We do so by comparing the limit of the parallelism expressed by programmers, with the one obtainable using PDG, and with the parallelism obtainable using the new PS-PDG abstraction. The critical path is computed as the number of dynamic LLVM instructions that must run sequentially given a parallelization plan. The PDG measurements assume that every outermost loop is parallelized using DOALL, HELIX, or DSWP using the SCCs generated from the PDG. The J&K measurements assume that every outermost loop is parallelized using the SCCs from the PDG along with inner developer-expressed loops. Lastly, the PS-PDG measurements assume that every outer loop is parallelized using the SCCs from the PS-PDG, as well as inner developer-expressed loops. We only consider the hierarchical parallelism possible with existing parallelizing compilers or as expressed by the developer. This methodology is consistent with that proposed by others [57].

Fig. 14 shows the critical path speedup over the programmer encoded parallelization for an ideal machine. By utilizing the PS-PDG, the automatic-parallelizing compiler can leverage the developer encoded parallel semantics as well as the advanced compiler techniques, HELIX/DSWP. The PS-PDG allows the compiler to discover plans with far more parallelization potential than with the PDG. Additionally, for benchmarks with good parallelization coverage by the programmer (e.g., EP), the PS-PDG ensures no loss of parallelism since it captures the precise parallel constraints of the parallel program. Finally, we find that utilizing workshare improved loop dependence analysis with the PDG (J&K) is unable to unlock as much parallelization potential as the PS-PDG (e.g., IS).

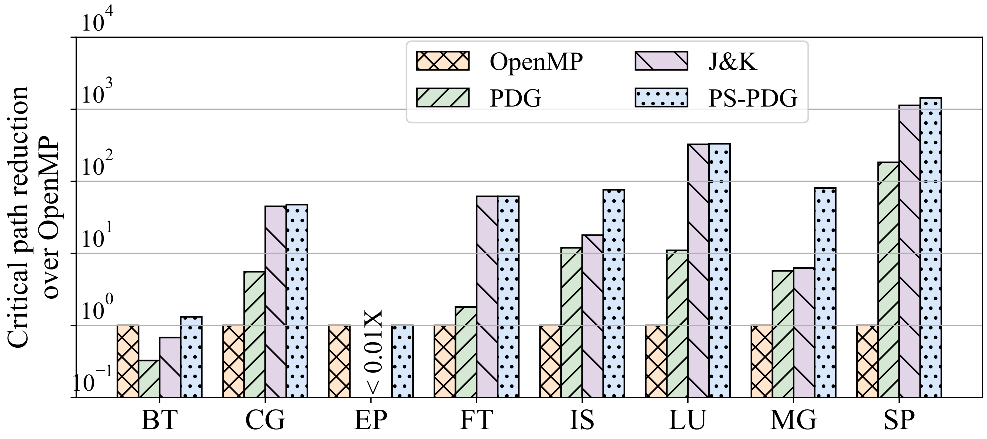


Fig. 14. Critical path reduction from abstraction-enabled parallelism.

7 RELATED WORK

Previous work [49, 50, 53] proposed representations of the explicit parallelism encoded in a program to help programmers understand their parallelization. These representations directly capture the parallel control flow encoded in the parallel program. Other prior work [2, 29, 30, 41, 51] 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 [28, 38] 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 [32] 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 [33] and the Kinetic Dependence Graph [25] 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) [7, 10, 11, 21, 23, 24, 31, 37, 39, 44, 46, 52, 56]. 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).

8 CONCLUSION

This work presents the PS-PDG, a novel abstraction that captures the precise parallel constraints of modern parallel programs. The PS-PDG is shown to be necessary and sufficient for capturing the precise parallel constraints found in OpenMP and Cilk. Our PS-PDG enabled compiler is capable of enabling advanced parallel execution plan optimizations for explicitly parallel programs on modern many-core systems.

REFERENCES

[1] 2022. Write fast code with C/C++ and OpenCilk. https://www.opencilk.org/

[2] 2023. LLVM/OpenMP design and overview. https://www.https://openmp.llvm.org/

[3] 2023. RTL Representation. https://gcc.gnu.org/onlinedocs/gccint/RTL.html

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

[5] Sotiris Apostolakis, Ziyang Xu, Greg Chan, Simone Campanoni, and David I. August. 2020. 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* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 351–367. https://doi.org/10.1145/3373376.3378458

[6] 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* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 638–654. https://doi.org/10.1145/3385412.3386028

[7] Arvind Arvind, Rishiyur Nikhil, and Keshav Pingali. 1989. I-Structures: Data Structures for Parallel Computing. *ACM Trans. Program. Lang. Syst.* 11 (Oct. 1989), 598–632. https://doi.org/10.1145/69558.69562

[8] 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. 1991. The NAS Parallel Benchmarks—Summary and Preliminary Results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing* (Albuquerque, New Mexico, USA) *(Supercomputing '91)*. Association for Computing Machinery, New York, NY, USA, 158–165. https://doi.org/10.1145/125826.125925

[9] David A. Beckingsale, Jason Burmark, Rich Hornung, Holger Jones, William Killian, Adam J. Kunen, Olga Pearce, Peter Robinson, Brian S. Ryujin, and Thomas RW Scogland. 2019. RAJA: Portable Performance for Large-Scale Scientific Applications. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 71–81. https://doi.org/10.1109/P3HPC49587.2019.00012

[10] Guy Blelloch and John Greiner. 1995. Parallelism in sequential functional languages. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture - FPCA '95*. ACM Press, La Jolla, California, United States, 226–237. https://doi.org/10.1145/224164.224210

[11] Guy E. Blelloch, Jonathan C. Hardwick, Siddhartha Chatterjee, Jay Sipelstein, and Marco Zagha. 1993. Implementation of a portable nested data-parallel language. *ACM SIGPLAN Notices* 28, 7 (July 1993), 102–111. https://doi.org/10.1145/173284.155343

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

[13] Simone Campanoni and Stefano Crespi Reghizzi. 2009. Traces of Control-Flow Graphs. In *Developments in Language Theory*, Volker Diekert and Dirk Nowotka (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 156–169.

[14] Simone Campanoni, Timothy Jones, Glenn Holloway, Vijay Janapa Reddi, Gu-Yeon Wei, and David Brooks. 2012. HELIX: Automatic Parallelization of Irregular Programs for Chip Multiprocessing. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization* (San Jose, California) *(CGO '12)*. ACM, New York, NY, USA, 84–93. https://doi.org/10.1145/2259016.2259028

[15] Simone Campanoni, Timothy M Jones, Glenn Holloway, Gu-Yeon Wei, and David Brooks. 2012. HELIX: Making the extraction of thread-level parallelism mainstream. *IEEE Micro* 32, 4 (2012), 8–18.

[16] Li-Wen Chang, Izzat El Hajj, Christopher Rodrigues, Juan Gómez-Luna, and Wen-mei Hwu. 2016. Efficient kernel synthesis for performance portable programming. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–13. https://doi.org/10.1109/MICRO.2016.7783715

[17] Tom Deakin, Simon McIntosh-Smith, James Price, Andrei Poenaru, Patrick Atkinson, Codrin Popa, and Justin Salmon. 2019. Performance Portability across Diverse Computer Architectures. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 1–13. https://doi.org/10.1109/P3HPC49587.2019.00006

[18] Yuanbo Fan, Simone Campanoni, and Russ Joseph. 2019. Time squeezing for tiny devices. In *Proceedings of the 46th International Symposium on Computer Architecture, ISCA 2019, Phoenix, AZ, USA, June 22-26, 2019*. 657–670. https://doi.org/10.1145/3307650.3322268

[19] Yuanbo Fan, Tianyu Jia, Jie Gu, Simone Campanoni, and Russ Joseph. 2018. Compiler-guided Instruction-level Clock Scheduling for Timing Speculative Processors. In *Proceedings of the 55th Annual Design Automation Conference* (San Francisco, California) *(DAC '18)*. ACM, New York, NY, USA, Article 40, 6 pages. https://doi.org/10.1145/3195970.3196013

[20] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.* 9, 3 (jul 1987), 319–349. https://doi.org/10.1145/24039.24041

[21] Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. [n. d.]. Implicitly-threaded Parallelism in Manticore. ([n. d.]), 12.

[22] Saturnino Garcia, Donghwan Jeon, Christopher M. Louie, and Michael Bedford Taylor. 2011. Kremlin: Rethinking and Rebooting Gprof for the Multicore Age. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 458–469. https://doi.org/10.1145/1993498.1993553

[23] Adrien Guatto, Sam Westrick, Ram Raghunathan, Umut Acar, and Matthew Fluet. 2018. Hierarchical memory management for mutable state. *ACM SIGPLAN Notices* 53, 1 (Feb. 2018), 81–93. https://doi.org/10.1145/3200691.3178494

[24] Robert H. Halstead. 1984. Implementation of multilisp: Lisp on a multiprocessor. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming (LFP '84)*. Association for Computing Machinery, New York, NY, USA, 9–17. https://doi.org/10.1145/800055.802017

[25] Muhammad Amber Hassaan, Donald D. Nguyen, and Keshav K. Pingali. 2015. Kinetic Dependence Graphs. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (Istanbul, Turkey) *(ASPLOS '15)*. Association for Computing Machinery, New York, NY, USA, 457–471. https://doi.org/10.1145/2694344.2694363

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

[27] ISO JTC1/SC22/WG14 - N1665 2012. *Intel® Cilk™ Plus Language Extension Specification*. Technical Report. International Organization for Standardization. https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1665.htm

[28] Nicklas Bo Jensen and Sven Karlsson. 2017. Improving Loop Dependence Analysis. *ACM Trans. Archit. Code Optim.* 14, 3, Article 22 (aug 2017), 24 pages. https://doi.org/10.1145/3095754

[29] Richard Johnson, David Pearson, and Keshav Pingali. 1994. The Program Structure Tree: Computing Control Regions in Linear Time. *SIGPLAN Not.* 29, 6 (jun 1994), 171–185. https://doi.org/10.1145/773473.178258

[30] Herbert Jordan, Simone Pellegrini, Peter Thoman, Klaus Kofler, and Thomas Fahringer. 2013. INSPIRE: The insieme parallel intermediate representation. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*. 7–17. https://doi.org/10.1109/PACT.2013.6618799

[31] Ulrike Klusik, Rita Loogen, Steffen Priebe, and Fernando Rubio. 2001. Implementation Skeletons in Eden: Low-Effort Parallel Programming *(Lecture Notes in Computer Science)*, Markus Mohnen and Pieter Koopman (Eds.). Springer, Berlin, Heidelberg, 71–88. https://doi.org/10.1007/3-540-45361-X_5

[32] 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* (Vienna, Austria) *(PPoPP '18)*. Association for Computing Machinery, New York, NY, USA, 68–80. https://doi.org/10.1145/3178487.3178493

[33] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. 2007. Optimistic Parallelism Requires Abstractions. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '07)*. Association for Computing Machinery, New York, NY, USA, 211–222. https://doi.org/10.1145/1250734.1250759

[34] Monica Lam. 1988. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*. 318–328.

[35] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 75.

[36] Chingren Lee, Jenq Kuen Lee, and TingTing Hwang. 2000. Compiler optimization on instruction scheduling for low power. In *Proceedings 13th International Symposium on System Synthesis*. IEEE, 55–60.

[37] Peng Li, Simon Marlow, Simon Peyton Jones, and Andrew Tolmach. 2007. Lightweight concurrency primitives for GHC. In *Proceedings of the ACM SIGPLAN workshop on Haskell workshop (Haskell '07)*. Association for Computing Machinery, New York, NY, USA, 107–118. https://doi.org/10.1145/1291201.1291217

[38] Jiacheng Ma, Wenyi Wang, Aaron Nelson, Michael Cuevas, Brian Homerding, Conghao Liu, Zhen Huang, Simone Campanoni, Kyle C. Hale, and Peter A. Dinda. 2021. Paths to OpenMP in the kernel. In *SC '21: The International Conference for High Performance Computing, Networking, Storage and Analysis, St. Louis, Missouri, USA, November 14 - 19, 2021*, Bronis R. de Supinski, Mary W. Hall, and Todd Gamblin (Eds.). ACM, 65:1–65:17. https://doi.org/10.1145/3458817.3476183

[39] Simon Marlow. 2012. Parallel and Concurrent Programming in Haskell. In *Central European Functional Programming School: 4th Summer School, CEFP 2011, Budapest, Hungary, June 14-24, 2011, Revised Selected Papers*, Viktória Zsók, Zoltán Horváth, and Rinus Plasmeijer (Eds.). Springer, Berlin, Heidelberg, 339–401. https://doi.org/10.1007/978-3-642-32096-5_7

[40] 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 *Proceedings of the 20th IEEE/ACM International Symposium on Code Generation and Optimization* (Virtual Event, Republic of Korea) *(CGO '22)*. IEEE Press, 179–192. https://doi.org/10.1109/CGO53902.2022.9741276

[41] V. Krishna Nandivada, Jun Shirako, Jisheng Zhao, and Vivek Sarkar. 2013. A Transformation Framework for Optimizing Task-Parallel Programs. *ACM Trans. Program. Lang. Syst.* 35, 1, Article 3 (apr 2013), 48 pages. https://doi.org/10.1145/2450136.2450138

[42] Ton Ngo, Lawrence Snyder, and Bradford Chamberlain. 1997. Portable Performance of Data Parallel Languages. In *Proceedings of the 1997 ACM/IEEE Conference on Supercomputing* (San Jose, CA) *(SC '97)*. Association for Computing Machinery, New York, NY, USA, 1–20. https://doi.org/10.1145/509593.509611

[43] OpenMP Architecture Review Board. 2018. OpenMP Application Program Interface Version 5.0. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf

[44] Simon Peyton Jones, Roman Leshchinskiy, Gabriele Keller, and Manuel Chakravarty. 2008. Harnessing the Multicores: Nested Data Parallelism in Haskell. *Leibniz International Proceedings in Informatics, LIPIcs* 2 (Dec. 2008). https://doi.org/10.4230/LIPIcs.FSTTCS.2008.1769

[45] Phitchaya Mangpo Phothilimthana, Jason Ansel, Jonathan Ragan-Kelley, and Saman Amarasinghe. 2013. Portable Performance on Heterogeneous Architectures. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Houston, Texas, USA) *(ASPLOS '13)*. Association for Computing Machinery, New York, NY, USA, 431–444. https://doi.org/10.1145/2451116.2451162

[46] Ram Raghunathan, Stefan K. Muller, Umut A. Acar, and Guy Blelloch. 2016. Hierarchical memory management for parallel programs. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP 2016)*. Association for Computing Machinery, New York, NY, USA, 392–406. https://doi.org/10.1145/2951913.2951935

[47] Vijay Janapa Reddi, Simone Campanoni, Meeta S. Gupta, Michael D. Smith, Gu-Yeon Wei, David Brooks, and Kim Hazelwood. 2010. Eliminating Voltage Emergencies via Software-guided Code Transformations. *ACM Trans. Archit. Code Optim.* 7, 2, Article 12 (Oct. 2010), 28 pages. https://doi.org/10.1145/1839667.1839674

[48] Vivek Sarkar. 1989. *Partitioning and Scheduling Parallel Programs for Multiprocessors.* MIT Press, Cambridge, MA, USA.

[49] Vivek Sarkar. 1997. Analysis and Optimization of Explicitly Parallel Programs Using the Parallel Program Graph Representation. In *LCPC*.

[50] Vivek Sarkar and Barbara Simons. 1993. Parallel Program Graphs and Their Classification. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*. Springer-Verlag, Berlin, Heidelberg, 633–655.

[51] 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* (Austin, Texas, USA) *(PPoPP '17)*. Association for Computing Machinery, New York, NY, USA, 249–265. https://doi.org/10.1145/3018743.3018758

[52] K. C. Sivaramakrishnan, Lukasz Ziarek, and Suresh Jagannathan. 2014. MultiMLton: A multicore-aware runtime for standard ML. *Journal of Functional Programming* 24, 6 (Nov. 2014), 613–674. https://doi.org/10.1017/S0956796814000161 Publisher: Cambridge University Press.

[53] Harini Srinivasan and Michael Wolfe. 1991. Analyzing Programs with Explicit Parallelism. In *Proceedings of the Fourth International Workshop on Languages and Compilers for Parallel Computing*. Springer-Verlag, Berlin, Heidelberg, 405–419.

[54] Kim-Anh Tran, Trevor E Carlson, Konstantinos Koukos, Magnus Själander, Vasileios Spiliopoulos, Stefanos Kaxiras, and Alexandra Jimborean. 2017. Clairvoyance: Look-ahead compile-time scheduling. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 171–184.

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

[56] Sam Westrick, Rohan Yadav, Matthew Fluet, and Umut A. Acar. 2020. Disentanglement in nested-parallel programs. *Proceedings of the ACM on Programming Languages* 4, POPL (Jan. 2020), 1–32. https://doi.org/10.1145/3371115

[57] 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)*. 151–162. https://doi.org/10.1109/IISWC53511.2021.00024

A SUFFICIENCY OF PS-PDG FOR CILK

This Appendix describes how to map the Cilk parallel programming model to PS-PDG. In particular, we refer to OpenCilk 2.0 [1] as Cilk as it is the only actively maintained implementation as of this writing. Following OpenCilk 2.0, we exclude features such as inlets, array operations, elemental functions, and the `simd` pragma. Note that inlets were removed as early as Intel Cilk Plus [27]. Additionally, since array operations and elemental functions are language constructs to exploit data parallelism, their semantics can be equivalently expressed as a `cilk_for` if desired. Similar to OpenMP, we do not consider Cilk clauses that control the amount of parallelism to generate such as `grainsize`. Note however that the Cilk `simd` is semantically identical to the OpenMP `simd`.

The execution model of Cilk is expressed in the PS-PDG as follows. The `cilk_spawn` construct is represented as a hierarchical single-entry single-exit (SESE) node that contains two nodes: an inner entry node (referred to as a knot in the Cilk community) with two directed outgoing edges: one to a node within the hierarchical node representing the function call within the spawn construct, and the other exiting the hierarchical node. Each directed outgoing edge from the knot represents an outgoing strand of execution (thread) from the spawn knot. That is, the node representing the spawned function call represents forking a thread to call that function to be joined at the next synchronization point. Similar to `omp barrier`, `cilk_sync` is represented by a node with incoming edges from all nodes that contain spawned function calls in the smallest hierarchical node that contains it. The construct `cilk_scope` is represented by a SESE hierarchical node that contains a `cilk_sync` as the exit node and contains the nodes needed to represent the contents of the Cilk scope. The `cilk_sync` node is necessary as there is an implicit synchronization at the end of every Cilk scope. Finally, `cilk_for` is represented identically to `omp parallel for`.

Cilk hyperobjects are represented as reducible parallel semantic variables. Recall that Cilk hyperobjects are copied from the parent function into the child strand, and after the spawned child is joined back into the parent, the parent's view of the object is reduced with the child's view using the given reducer operation by the programmer. At this point, the child's view of the object is destroyed along with the spawned strand. By design, this semantic also provides support for other hyperobjects such as holders, which are a special case of reducers. For these classes of hyperobjects, parallel semantic variables and their properties provide the relevant semantics for Cilk. Therefore, we conclude that the PS-PDG abstraction possesses all the features needed to capture the Cilk programming model, as well as OpenMP.