Polyhedral Optimizations of Explicitly Parallel Programs

Prasanth Chatarasi, Jun Shirako, Vivek Sarkar
Department of Computer Science
Rice University, Houston, Texas-77005, USA
{*prasanth,shirako,vsarkar*}*@rice.edu*

***Abstract*—The polyhedral model is a powerful algebraic framework that has enabled significant advances to analysis and transformation of sequential affine (sub)programs, relative to traditional AST-based approaches. However, given the rapid growth of parallel software, there is a need for increased attention to using polyhedral frameworks to optimize explicitly parallel programs. An interesting side effect of supporting explicitly parallel programs is that doing so can also enable optimization of programs with unanalyzable data accesses within a polyhedral framework. In this paper, we address the problem of extending polyhedral frameworks to enable analysis and transformation of programs that contain both explicit parallelism and unanalyzable data accesses. As a first step, we focus on OpenMP loop parallelism and task parallelism, including task dependences from OpenMP 4.0.**

Our approach first enables conservative dependence analysis of a given region of code. Next, we identify happens-before relations from the explicitly parallel constructs, such as tasks and parallel loops, and intersect them with the conservative dependences. Finally, the resulting set of dependences is passed on to a polyhedral optimizer, such as PLuTo and PolyAST, to enable transformation of explicitly parallel programs with unanalyzable data accesses.

We evaluate our approach using eleven OpenMP benchmark programs from the KASTORS and Rodinia benchmark suites. We show that 1) these benchmarks contain unanalyzable data accesses that prevent polyhedral frameworks from performing exact dependence analysis, 2) explicit parallelism can help mitigate the imprecision, and 3) polyhedral transformations with the resulting dependences can further improve the performance of the manually-parallelized OpenMP benchmarks. Our experimental results show geometric mean performance improvements of 1.62x and 2.75x on the Intel Westmere and IBM Power8 platforms respectively (relative to the original OpenMP versions).

***Keywords*-Explicit parallelism; Polyhedral transformations; Task parallelism; OpenMP; Happens-before relations**

I. Introduction

A key challenge for optimizing compilers is to keep up with the increasing complexity related to locality and parallelism in modern computers, especially as computer vendors head towards new designs for extreme-scale processors and exascale systems [1]. Classical AST-based optimizers typically focus on one particular objective at a time, such as vectorization, locality or parallelism. In contrast, polyhedral transformation frameworks support complex sequences of transformations of perfectly/imperfectly nested loops in a unified formulation. The advantages of this unified formulation are seen in polyhedral optimizers, such as PLuTo [2], [3] and PolyAST [4]. It has even been extended and specialized to integrate SIMD constraints [5]. Polyhedral frameworks achieve this generality in transformations by restricting the class of programs that do not have *unanalyzable* control or data flow. In the original formulation of polyhedral frameworks, all array subscripts, loop bounds, and branch conditions in *analyzable* programs were required to be affine functions of loop index variables and global parameters. However, decades of research since then have led to a great expansion of programs that can be considered analyzable by polyhedral frameworks. The main remaining constraints stem from restrictions on various program constructs including pointer aliasing, unknown function calls, non-affine expressions, recursion, and unstructured control flow.

Our work is motivated by the observation that software with explicit parallelism is on the rise. It can be used to enable larger set of polyhedral transformations (by mitigating conservative dependences), compared to what might have been possible if the input program is sequential. Our work focuses on explicitly-parallel programs that specify potential logical parallelism, rather than actual parallelism. Thus, explicit parallelism is simply a specification of a partial order, traditionally referred to as a happens-before relations [6]. Dependences can only occur among statement instances that are ordered by the happens-before relations. Hence, we can reduce spurious dependences arising from the unanalyzable constructs by intersecting happens-before relations with conservative dependences.

In this paper, we restrict our attention to explicitly-parallel programs that satisfy the *serial elision* property, i.e., the property that removal of all parallel constructs results in a sequential program that is a valid (albeit inefficient) implementation of the parallel program semantics [7]. We observe that loop-level and task-level parallelism form the core of modern parallel programming languages, such as OpenMP [8], Chapel [9], Cilk [10], and X10 [11]. So, we focus our attention on loop-level and task-level constructs in OpenMP that satisfy the serial elision property, while deferring support for SPMD constructs that do not satisfy this property to future work.

A summary of our approach is as follows. We first enable conservative dependence analysis of a given region of code. Next, we identify happens-before relations from the explicitly parallel constructs and intersect with the conservative dependences. Finally, the resulting set of dependences is passed to the polyhedral transformation tools, such as PLuTo [2], [3] and PolyAST [4], to enable the transformations of explicitly-parallel programs with unanalyzable data accesses. To the best of our knowledge, our work is the first to enable the polyhedral transformations of explicitly parallel OpenMP programs by combining the classical dependence analysis with happens-before analysis for explicit parallelism[1].

The rest of the paper is organized as follows. Section II summarizes background material, and Section III motivates the problem using OpenMP benchmark programs. Section IV provides an overview of our approach for enabling polyhedral transformations of explicitly parallel programs; we refer to our framework as the Polyhedral optimizer for Parallel Programs (PoPP). Section V presents experimental results to evaluate our approach on OpenMP benchmarks from the Kastors [13] and the Rodinia [14] benchmark suites on a 12-core Intel Westmere processor and a 24-core IBM Power8 system. Section VII and Section VIII summarize related work and our conclusions.

II. Background

We start with a brief overview of the polyhedral model, the basis of the proposed optimizing framework. Next, we briefly summarize explicit-parallelism including loop-level and task-level parallelism in the context of OpenMP [8], which is a widely used shared memory parallel programming model.

A. Polyhedral Model

The polyhedral model is a flexible representation for perfect and imperfect loop nests with static predictable control. Loop nests amenable to this algebraic representation are called *Static Control Parts* (SCoPs). It consists of a set of consecutive statements, and each statement contains three elements namely iteration domain, access relations, and schedule. The loop bounds, branch conditions, and array subscripts in the SCoP need to be affine functions of iterators and global parameters. A code region that does not strictly satisfy the above requirements can be also represented in the polyhedral model via conservative estimations.

Iteration domain, $\mathcal{D_S}$: A statement $\mathcal{S}$ enclosed by 'm' loops is represented by a m-dimensional polytope, referred to as an iteration domain of the statement [15]. Each element in the iteration domain of the statement is regarded as a statement instance.

Access relation: Each array expression in the statement is expressed through an access relation in the SCoP. An access relation maps the statement instance to one or more array elements [16]. It can conservatively support non-affine array expressions by mapping them to multiple array elements, perhaps even the entire range of the array. An example of a non-affine array access is shown below. The array reference to `x` is an indirect access via `col[j]` and is considered to read the entire range of `x[*]` to enable conservative estimations. In contrast, an access function maps a statement instance to a single array element, and cannot support non-affine accesses as a result.

```
for(i = 0; i < n; i++)
  for(j = index[i]; j < index[i+1]; j++)
        y[i] += A[j]*x[col[j]];
```

Schedule: is a function which associates a logical execution date (a timestamp) to each instance of a given statement. In the case of multidimensional schedules, this timestamp is a vector. In the program, statement instances will be executed according to the increasing lexicographic order of their timestamp.

Dependence Polyhedra, $\mathcal{P}^{\mathcal{S}\rightarrow\mathcal{T}}$: captures all possible dependences between statements $\mathcal{S}$ and $\mathcal{T}$. Two statement instances $\vec{X_s}$ and $\vec{X_t}$, which belong to the iteration domains of statements $\mathcal{S}$ and $\mathcal{T}$ respectively, are said to be in dependence if they access the same array location and at least one of them is a write. Multiple dependence polyhedra may be required to capture all dependent instances between two statements (scalars are simply treated as zero-dimensional arrays). For a given schedule, *depth* of a dependence polyhedron indicates the loop nest level where its dependence is carried. In other words, depth is the first non-zero dimension of the corresponding dependence vector.

A dependence polyhedron captures exact dependence information when each of the access relations is an access function or if the access relation models an exact read/write of an array range, e.g., a memset of an entire array. However, dependence polyhedra can be overestimated due to conservative access relations when array subscripts include unanalyzable accesses.

[1]An earlier version of this paper was informally presented at the IMPACT'15 workshop [12], which is a forum that does not include formal proceedings.

B. Explicit Parallelism

The major difference between sequential programs and explicitly-parallel programs is that sequential programs specify a total execution order, whereas the execution of an explicitly-parallel program can be viewed as a partial order, which is traditionally referred to as a happens-before relation. We briefly summarize the loop-level and task-level constructs in the context of OpenMP [8].

1) Loop-level parallelism: The OpenMP loop construct, `#pragma omp for`, is specified immediately before a `for` loop. This construct indicates that the iterations of the loop can be executed in parallel, which guarantees no happens-before relations among iterations. A barrier, i.e., an all-to-all synchronization point, is implied immediately after the parallel loop region.

The `private(`*op*`:` *list*`)` clause, which is attached to a `for` loop construct, indicates that each OpenMP thread has its own private copies of the variables specified in *list*.

2) Task-level parallelism including dependences: The OpenMP `task` construct, `#pragma omp task`, is specified on a code region and indicates the creation of an asynchronous task to process the region. Synchronization among the parent task and its child tasks (i.e., tasks spawned by the parent task) is supported by the `taskwait` construct, `#pragma omp taskwait`. This directive specifies a synchronization point at which the encountering task waits for all its child tasks to complete. Synchronization among the sibling tasks with the same parent task is supported by the `depend(`*type*`:` *vars*`)` clauses attached on a `task` construct. Here, *type* is `in`, `out`, or `inout` to imply read, write, or read-and-write access on *vars*, which is a list of variables that can include arrays[2]. The ordering constraints enforced by the depend clauses are as follows:

- `in` *dependence-type.* The generated task will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an `out` or `inout` depend clause.
- `out` *and* `inout` *dependence-types.* The generated task will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an `in`, `out`, or `inout` depend clause.

A task can start its execution only when all the dependent tasks have completed. These dependences on previous generated tasks enforce *serial elision* property. More details on these constructs can be found in [17].

III. Motivating Examples

To motivate the proposed approach, we discuss two explicitly parallel kernels with data accesses that are likely to be considered unanalyzable by many existing polyhedral frameworks. The first example uses C nested arrays which may have an unrestricted pointer aliasing, in general. The second example uses linearized (non-affine) array subscripts that would require a delinearization analysis to make them analyzable by polyhedral frameworks.

A. 2-D Jacobi

```
 1 jacobi (double *u_, double *unew_, double *f_)
 2 {
 3 double (*f)[nx][ny] = (double (*)[nx][ny])f_;
 4 double (*u)[nx][ny] = (double (*)[nx][ny])u_;
 5 double (*unew)[nx][ny] = \
 6              (double (*)[nx][ny])unew_;

 8 #pragma omp parallel
 9 #pragma omp single
10 {
11 for (int it = itold + 1; it <= itnew; it++) {
12   for (int i = 0; i < nx; i++) {
13 #pragma omp task depend(out: u[i]) \
14                  depend(in: unew[i])
15     for (int j = 0; j < ny; j++) {
16       (*u)[i][j] = (*unew)[i][j];
17   } }
18   for (int i = 0; i < nx; i++) {
19 #pragma omp task depend(out: unew[i]) \
20         depend(in: f[i], u[i-1], u[i], u[i+1])
21     for (int j = 0; j < ny; j++) {
22       if (i == 0 || j == 0 ||
23           i == nx - 1 || j == ny - 1) {
24         (*unew)[i][j] = (*f)[i][j];
25       } else {
26         (*unew)[i][j] = 0.25 * ((*u)[i-1][j]
27          + (*u)[i][j+1] + (*u)[i][j-1]
28          + (*u)[i+1][j] + (*f)[i][j] * dx * dy);
29  } } } }
30 #pragma omp taskwait
31 } }
```

Figure 1: 2-D Jacobi kernel from KASTORS suite.

The first example (in Figure 1) is a 2-dimensional Jacobi computation from the KASTORS suite [13]. This computation is parallelized using the OpenMP 4.0 task construct with depend clauses. Even though the loop nest has affine accesses on arrays `u` and `unew`, the possible aliasing of the flat array pointers can prevent a sound compiler analysis from detecting the exact cross-iteration dependences. However, the happens-before relations described through the `task depend` clauses (lines 13-14, lines 19-20) indicate uniform dependence patterns only among neighboring iterations (i.e., `u[i-1]`, `u[i]`, and `u[i+1]`), which enable skewing, tiling, and doacross pipelined parallelization. Section V shows how these transformations improve the data locality and the parallelism granularity and contribute the overall performance. However, there exist speculative approaches that add code to the program to check if all referenced arrays of a loop nest do not overlap and to generate optimized variants that can be selected at runtime [18].

[2]Any type or rank of arrays are permitted; in Figure 1, 1-D array of type `double (*)[ny]` are used.

B. Particle Filter

The second example (in Figure 2) is the `particle filter` kernel from the Rodinia suite [14]. The loop nests in the kernel contain linearized (non-affine) array subscripts such as `ind[x*countOnes+y]`, and indirect array subscript (`I[ind[x*countOnes+y]]`), that may pose challenges to the compiler for analysis.

```
 1 #define ALLOC(N) (double *) \
 2         malloc(sizeof(double)*N)
 3 void particleFilter(int *I, int Nparticles) {
 4 ....
 5 double *weights = ALLOC(Nparticles);
 6 double *arrayX = ALLOC(Nparticles);
 7 double *arrayY = ALLOC(Nparticles);
 8 double *likelihood = ALLOC(Nparticles);
 9 double *objxy = ALLOC(countOnes*2);
10 int *ind = (int*)malloc(sizeof(int) * \
11                 countOnes*Nparticles);

13 #pragma omp parallel for
14 for(x = 0; x < Nparticles; x++){
15   arrayX[x] += 1 + 5*randn(seed, x);
16   arrayY[x] += -2 + 2*randn(seed, x);
17 }

19 #pragma omp parallel for private(y, indX, indY)
20 for(x = 0; x < Nparticles; x++){
21   for(y = 0; y < countOnes; y++){
22     indX = roundDouble(arrayX[x]) + objxy[y*2+1];
23     indY = roundDouble(arrayY[x]) + objxy[y*2];
24     ind[x*countOnes+y] = fabs(indX ... indY ...);
25     ...
26     likelihood[x] += ...I[ind[x*countOnes+y]]...
27   }
28   ...}

30 #pragma omp parallel for
31 for(x = 0; x < Nparticles; x++){
32   weights[x] = weights[x] * exp(likelihood[x]);

34 #pragma omp parallel for private(x) \
35   reduction(+:sumWeights)
36 for(x = 0; x < Nparticles; x++)
37   sumWeights += weights[x];
38 }
39 ....
40 }
```

Figure 2: Particle filter kernel from Rodinia suite

Although de-linearization techniques [19] can handle the `ind[x*countOnes+y]` case, and the fact that array `I` is read-only in the kernel can be used to handle the `I[ind[x*countOnes+y]]` case, the use of parallel loop constructs can prune conservativeness in dependence analysis, even in the absence of techniques such as delinearization. The legality of loop fusion is easily established by the fact that all variables that cross multiple loops have affine accesses with no fusion-preventing dependences and the arrays don't alias each other, as these array are malloc(ed) in the same kernel. The key information needed from the parallel program is that the second loop (lines 19-28 in Figure 2) has no loop-carried dependence. This ensures that the resulting loop after fusing all four loops can also be made parallel.

IV. Polyhedral optimizer for Parallel Programs (PoPP)

In this section, we introduce our framework for automatically optimizing explicitly parallel programs.

Algorithm 1 Overall steps in PoPP

```
1: Input: Explicitly parallel program, $\mathcal{I}$
2: $\mathcal{P}$ := set of conservative dependences in $\mathcal{I}$
3: $\mathcal{HB}$ := Transitive closure of happens-before relations
   from parallel constructs in $\mathcal{I}$
4: $\mathcal{P}' := \mathcal{P} \cap \mathcal{HB}$ ,
5: Optimized schedules, $\mathcal{S}$ = Transform($\mathcal{I}$, $\mathcal{P}'$)
6: $\mathcal{I}'$ = CodeGen($\mathcal{I}$, $\mathcal{S}$, $\mathcal{P}'$ )
7: Output: Optimized explicitly parallel program, $\mathcal{I}'$
```

Algorithm 1 shows the overall approach to conservatively handle unanalyzable accesses (step 2), extract happens-before (HB) relations from explicit parallelism (step 3), and improve the accuracy of conservative dependences (step 4). Then the resulting dependences are passed to polyhedral optimizers, such as PolyAST and PLuTo, to leverage existing loop transformations (step 5). Finally, the code generator is invoked to generate the optimized parallel program (step 6).

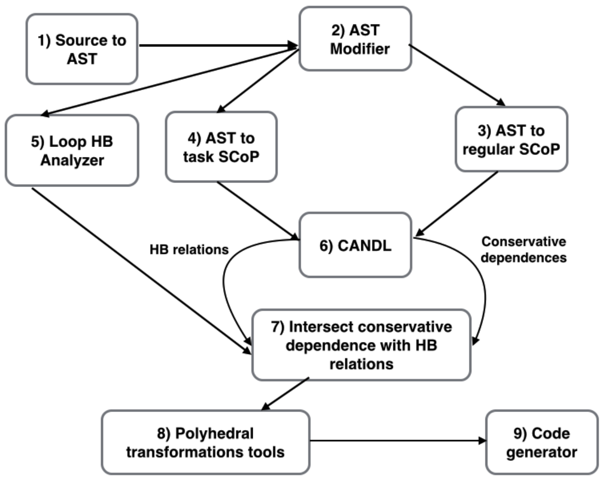


Figure 3: Overview of our approach

The overall approach is summarized in Figure 3, which is implemented as an extension to the PolyAST optimization framework [4] implemented in the ROSE compiler [20], and consists of the following components: 1) Conversion from source code to AST (with support for parallel-loop and parallel-task constructs), 2) AST Modifier (handling unanalyzable accesses), 3) AST to SCoP converter for regular statements, 4) AST to task SCoP converter (preprocessing of computing HB relations based on task parallelism), 5) Loop HB analyzer to compute HB relations based on loop level parallelism,

6) Use of CANDL [21] for both conservative dependence analysis and computing task-based HB relations, 7) Intersection of conservative dependences with HB relations, 8) Communication of the resulting set of dependences to a polyhedral transformation tool, such as PLuTo [2] and PolyAST [4], and 9) Code generator to produce automatically optimized code.

A. *Conservative analysis*

In case of unanalyzable data accesses, compilers must follow conservative dependence analysis that overestimates dependences and may report spurious dependences. In conservative dependence analysis, the basic assumption for a compiler is that all memory accesses of an array in a statement can potentially conflict with other memory accesses of that array, or perhaps even memory accesses in other arrays (in the case of unrestricted pointer aliasing). In this paper, we support such conservative assumptions by using the *dummy variable* approach described below.

Handling non-affine array subscripts. Non-affine subscripts such as linearized array subscripts and indirect array subscripts are common in regular benchmarks. These non-affine subscripts can be handled using the *access relations* in polyhedral extraction tools by assuming that they accesses the entire range instead of a single element in the array. Since our polyhedral framework in the infrastructure supports access functions but not access relations, we implemented similar functionality using a *dummy variable* approach. We treat non-affine subscript as a dummy variable and create affine inequalities such that these variables accesses the entire range of the array dimension [12]. There exists other conservative approaches such as array region analysis [22], fuzzy array data flow analysis [23] and other variants to approximate the access relations for arrays having non-affine subscripts.

Handling function calls. Function calls in the kernel pose challenges to polyhedral frameworks for analysis and transformations. We handle library/ user-defined function calls by treating them as regular statements and conservatively assume the statements read and write any array in the SCoP. But, there exists other sophisticated approaches such as array region analysis [24] used in PIPS compiler to approximate access relations and enhance dependence analysis in case of procedure calls.

Handling non-affine conditionals. Currently, polyhedral extraction tools have limitations in representing non-affine branch conditions in a polyhedral representation (SCoP). As a workaround, we handle an `if`-statement with a non-affine conditional and its corresponding `then` and `else` branches as a compound statement that inherits all the access relations in the condition, and the `then` and `else` branches[3]. Note that we allow multi-write per statement in the framework. For the benchmarks studied in Section V, such non-affine control flows are closed within a loop body; this approximation keeps the granularity of compound statements enough small to enable transformations/parallelizations. The Polyhedral Extraction Tool (PET) [25] also provides a way to represent data dependent assignments, data dependent accesses and data dependent conditions in the access relations.

After converting the given parallel program into polyhedral representation (SCoP) with above modifications, we use an existing polyhedral dependence analyzer (CANDL [21]) with the sequential schedule ignoring parallel constructs. The resulting dependence polyhedra can be directly used as conservative dependences. Adhering to polyhedral dependence notations, we use $\mathcal{P}_d^{S_i \to S_j}$ to represent the dependence between source statement S_i and target statement S_j at depth d where depth represents the loop nest level that carries the data dependence. The conservative dependences for the Jacobi kernel (in Figure 5(a)) are shown in Figure 5(c).

B. *Extraction of happens-before relations*

Happens-before (HB) relations [6] have been introduced in describing memory models. These relations can be defined as follows in the context of dependences between statements in the program.

Assume S_i and S_j are the statements in the program. If S_i **happens-before** *S_j, then the memory effects of S_i effectively become visible before statement S_j is executed.*

Explicit parallel constructs in the program specify the logical parallelism, which in turn describes the happens-before relations on the statements in the program. Let $\mathcal{U}_d^{S_i \to S_j}$ represent a given sequential ordering between source statement S_i and target statement S_j at depth d in the program when ignoring parallel constructs. Any happens-before relation is initialized to this sequential ordering:

$$\mathcal{HB}_d^{S_i \to S_j} = \mathcal{U}_d^{S_i \to S_j} \tag{1}$$

According to the explicit parallel constructs, the happens-before relations will be updated. This section introduces our approach to compute such happens-before relations in the cases of loop-parallel and task-parallel constructs, where the *serial-elision* property holds.

1) Loop-level parallelism: In the OpenMP, loop-level parallelism is expressed through the `#pragma omp parallel for` construct. This particular construct is an-

[3]This has been manually performed for programs with non-affine conditionals in Section V, but can be automated in future work.

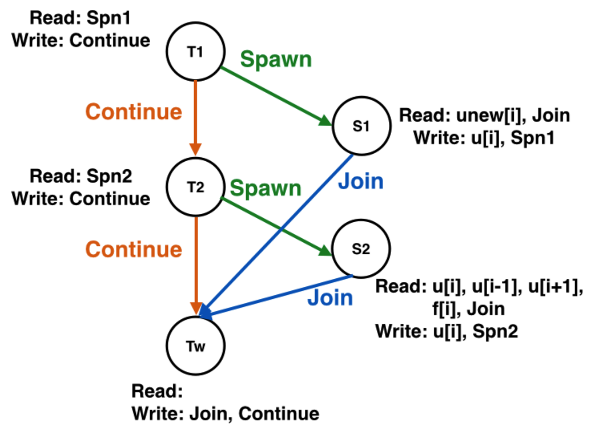


Figure 4: Happens-before relations for the Jacobi program in Figure 5(a) due to task-spawn, task-wait, and sequential ordering

notated on specific loops whose iterations can run in parallel, thereby it guarantees there are no happens-before relations among iterations of the annotated loop. Let S_i and S_j be statements enclosed in a `parallel for` loop at depth d, the corresponding happens-before relation is updated as:

$$\mathcal{HB}_d^{S_i \to S_j} = \phi \tag{2}$$

Note that the variables specified within `private` clause are expanded as arrays such that each parallel iteration accesses an unique element of the arrays, before the polyhedral compilation. In the post-polyhedral phase, the expanded arrays are replaced by the original variables with `private` attribute if they remain in parallel loops. This approach only applies to cases where each parallel loop is in an OpenMP parallel region by itself, and not to general OpenMP parallel regions (which are not supported by the approach in this paper).

2) Task parallelism including dependences: In OpenMP, task parallelism is specified through the `task`, `taskwait`, and `depend` constructs. As described in Section II-B2, these constructs specify ordering constraints 1) from parent task to child tasks via *task-spawn*, 2) from child tasks to parent task via *task-wait*, and 3) among sibling tasks via *inter-task dependence*. Computing happens-before relations in the presence of the inter-task dependences is challenging as it requires dependence analysis on the variables including arrays listed in the `depend(in/out)` clause.

In our approach, we encode these task-related constructs in the SCoP format by handling tasks as statements and `in`/`out` dependence type as read/write access; this is later processed by polyhedral dependence analyzers such as CANDL [21]. The resulting dependence polyhedra, $\mathcal{P}_{task_d}^{S_i \to S_j}$, are used as happens-before relations due to inter-task dependences among sibling tasks:

$$\mathcal{HB}_d^{S_i \to S_j} = \mathcal{P}_{task_d}^{S_i \to S_j} \tag{3}$$

The other relations among parent and child tasks are captured in the same manner by introducing special dependence variables shared by parent and children. We detail our approach in the rest of this section.

Task SCoP. Given a code region that contains `task` constructs, we define *task* SCoP that captures all required information to compute happens-before relations derived from tasks. As with regular SCoP, it includes set of statements and access relations which are modified as follows while domains and schedules are same as those in regular SCoP. **Statement.** The statements not enclosed in a `task` construct are handled in the same manner as regular SCoP statements. The `#pragma omp task` and `#pragma omp taskwait` are also handled as stand-alone statements that represent task-spawn and task-wait points, respectively. Finally, the body of `task` construct is handled as a compound statement, say *task-body* statement.

Figure 4 shows an example corresponding to the Jacobi kernel in Figure 5(a), where two task-spawn statements are represented as T1 and T2, a task-wait statement is Tw, and task-body statements are shown as S1 and S2.

Access relation. The `in` and `out` dependence types in `depend` clause are respectively handled as read and write accesses in the corresponding task-body statement (e.g., read: `unew[i]` / write: `u[i]` of S1 in Figure 4). In order to capture happens-before relations between parent and child tasks, we introduce the following special dependence variables and add to access relations.

- Spawn variable `Spn`*i* is added as a read access in *i*-th task-spawn statement and a write access in its task-body statement; the resulting Write-After-Read (WAR) dependence captures the ordering constraint on this specific task-spawn.
- Join variable `Join` is added as a read access in task-body statements and a write access in task-wait statements so that the WAR dependences capture ordering constraints on task-wait, which waits for all child tasks.
- Continue variable `Continue` is added as a write access in all statements by parent (i.e., task-spawn, task-wait, and regular statements) so that the Write-After-Write (WAW) dependences capture the sequential ordering.

Further, nested task graphs can be easily supported with the use of multiple join/continue variables for each level of nesting. The edges in Figure 4 represent the happens-before relations due to task-spawn, task-wait, and sequential ordering, which are computed by CANDL de-

```c
jacobi (double *u_, double *unew_, double *f_)
{
double (*f)[nx][ny] = (double (*)[nx][ny])f_;
double (*u)[nx][ny] = (double (*)[nx][ny])u_;
double (*unew)[nx][ny] = \
              (double (*)[nx][ny])unew_;

#pragma omp parallel
#pragma omp single
{
    for (int it = itold + 1; it <= itnew; it++) {
        for (int i = 0; i < nx; i++) {
#pragma omp task depend(out: u[i]) \
            depend(in: unew[i])                 // T1
            for (int j = 0; j < ny; j++)
S1:             u[i][j] = unew[i][j];
        }
        for (int i = 0; i < nx; i++) {
#pragma omp task depend(out: unew[i]) \
        depend(in: f[i], u[i-1], u[i], u[i+1]) // T2
            for (int j = 0; j < ny; j++)
S2:             cpd(i, j, unew, u, f);
         }
    }
    #pragma omp taskwait                        // Tw
}}
/* Original schedule
    S1: (0, it, 0, i, 0, j, 0)
    S2: (0, it, 1, i, 0, j, 0) */
```

(a) Input program: Jacobi from KASTORS suite; cpd represents compound statement.

```c
jacobi (double *u_, double *unew_, double *f_)
{
double (*f)[nx][ny] = (double (*)[nx][ny])f_;
double (*u)[nx][ny] = (double (*)[nx][ny])u_;
double (*unew)[nx][ny] = \
              (double (*)[nx][ny])unew_;


#pragma omp parallel for private(c3,c5) ordered(2)
for (c1 = itold + 1; c1 <= itnew; c1++) {
    for (c3 = 2 * c1; c3 <= 2 * c1 + nx; c3++) {
#pragma omp ordered depend(sink: c1-1, c3) \
                    depend(sink: c1, c3-1)
    if (c3 <= 2 * c1 + nx + -1) {
        for (c5 = 0; c5 < ny; c5++)
S1:         u[-2*c1+c3][c5] = unew[-2*c1+c3][c5];
    }
    if (c3 >= 2 * c1 + 1) {
        for (c5 = 0; c5 < ny; c5++)
S2:         cpd(-2*c1+c3-1, c5, unew, u, f);
    }
#pragma omp ordered depend(source)
  }
}
}

/* Transformed schedule
    S1: (0, it, 0, 2*it+i, 0, j, 0)
    S2: (0, it, 0, 2*it+i+1, 1, j, 0) */
```

(b) Jacobi after skewing and doacross pipelined parallelism; tiling was omitted due to space limitation.

$$
\begin{aligned}
&\mathcal{P}_1^{S1\to S1} : it' \geq it+1, i' = *, i = *, j' = *, j = *\\
&\mathcal{P}_2^{S1\to S1} : it' = it, i' \geq i+1, j' = *, j = *\\
&\mathcal{P}_3^{S1\to S1} : it' = it, i' = i, j' \geq j+1\\
&\mathcal{P}_1^{S1\to S2} : it' \geq it, i' = *, i = *, j' = *, j = *\\
&\mathcal{P}_1^{S2\to S1} : it' \geq it+1, i' = *, i = *, j' = *, j = *\\
&\mathcal{P}_1^{S2\to S2} : it' \geq it+1, i' = *, i = *, j' = *, j = *\\
&\mathcal{P}_2^{S2\to S2} : it' = it, i' \geq i+1, j' = *, j = *\\
&\mathcal{P}_3^{S2\to S2} : it' = it, i' = i, j' \geq j+1
\end{aligned}
$$

(c) Conservative dependences, $\mathcal{P}$

$$
\begin{aligned}
&\mathcal{HB}_1^{S1\to S1} : it' \geq it+1, i' = i\\
&\mathcal{HB}_2^{S1\to S1} : \phi\\
&\mathcal{HB}_3^{S1\to S1} : it' = it, i' = i, j' \geq j+1\\
&\mathcal{HB}_1^{S1\to S2} : it' \geq it, i' = i\\
&\qquad \cup\, it' \geq it, i' = i+1\\
&\qquad \cup\, it' \geq it, i' = i-1\\
&\mathcal{HB}_1^{S2\to S1} : it' \geq it+1, i' = i\\
&\qquad \cup\, it' \geq it+1, i' = i+1\\
&\qquad \cup\, it' \geq it+1, i' = i-1\\
&\mathcal{HB}_1^{S2\to S2} : it' \geq it+1, i' = i\\
&\mathcal{HB}_2^{S2\to S2} : \phi\\
&\mathcal{HB}_3^{S2\to S2} : it' = it, i' = i, j' \geq j+1
\end{aligned}
$$

(d) Happens-before relations, $\mathcal{HB}$

$$
\begin{aligned}
&\mathcal{P}_1^{\prime S1\to S1} : it' \geq it+1, i' = i, j' = *, j = *\\
&\mathcal{P}_2^{\prime S1\to S1} : \phi\\
&\mathcal{P}_3^{\prime S1\to S1} : it' = it, i' = i, j' \geq j+1\\
&\mathcal{P}_1^{\prime S1\to S2} : it' \geq it, i' = i, j' = *, j = *\\
&\qquad \cup\, it' \geq it, i' = i+1, j' = *, j = *\\
&\qquad \cup\, it' \geq it, i' = i-1, j' = *, j = *\\
&\mathcal{P}_1^{\prime S2\to S1} : it' \geq it+1, i' = i, j' = *, j = *\\
&\qquad \cup\, it' \geq it+1, i' = i+1, j' = *, j = *\\
&\qquad \cup\, it' \geq it+1, i' = i-1, j' = *, j = *\\
&\mathcal{P}_1^{\prime S2\to S2} : it' \geq it+1, i' = i, j' = *, j = *\\
&\mathcal{P}_2^{\prime S2\to S2} : \phi\\
&\mathcal{P}_3^{\prime S2\to S2} : it' = it, i' = i, j' \geq j+1
\end{aligned}
$$

(e) Accurate dependences, $\mathcal{P}' = \mathcal{P} \cap \mathcal{HB}$

Figure 5: Overall explanation of our framework on Jacobi benchmark from KASTORS suite.

pendence analyzer. As with Equation 3, the resulting dependence polyhedra are used as happens-before relations based on any task parallel constructs, after mapping task-body statements (i.e., compound statements) to regular statements. We use function inlining to handle tasks in non-recursive calls. However, handling of tasks in recursive calls is not currently supported by our approach.

C. Reflection of happens-before relations

After the extraction of happens-before relations from parallel constructs such as loop-level and task level constructs, it is necessary to reflect the happens-before relations onto conservative dependences as it prunes the

Algorithm 2 Intersection of happens-before relations with conservative dependences.

```
 1: Input: Conservative dependences $\mathcal{P}$, Happens-Before relations $\mathcal{HB}$
 2: for each dependence $\mathcal{P}_d^{S_i\to S_j}$ in $\mathcal{P}$ do
 3:   for each HB relation $\mathcal{HB}_e^{S_k\to S_l}$ in $\mathcal{HB}$ do
 4:     if $S_i = S_k$ & $S_j = S_l$ & $d = e$ then
 5:       $\mathcal{P}_d^{\prime S_i\to S_j} = \mathcal{P}_d^{S_i\to S_j} \cap \mathcal{HB}_e^{S_k\to S_l}$;
 6:     end if
 7:   end for
 8:   Add the intersected polyhedron $\mathcal{P}_d^{\prime S_i\to S_j}$ to $\mathcal{P}'$;
 9: end for
10: Output: Accurate dependences after intersection $\mathcal{P}'$
```

spurious dependences from the program. Note that HB is more conservative than $\mathcal{P}$ in all program regions that do not contain explicit parallelism. Given conservative dependences $\mathcal{P} \ni \mathcal{P}_d^{S_i \to S_j}$ and HB relations $\mathcal{HB} \ni \mathcal{HB}_e^{S_k \to S_l}$, we define $\mathcal{P}' = \mathcal{P} \cap \mathcal{HB}$ where $\mathcal{P}_d^{S_i \to S_j} \cap \mathcal{HB}_e^{S_k \to S_l}$ is non-empty if and only if $S_i = S_k$ & $S_j = S_l$ & $d = e$. According to the definition, the happens-before relation must be transitive (like all binary relations); our approach removes dependences only for pairs of source and target instances that are not in the HB relation (including transitive dependences). Therefore, the intersection keeps any dependences between code portions that are not annotated as running in parallel.

Figure 5(e) shows the improved dependence information for the Jacobi kernel in Figure 5(a), by intersecting the conservative dependences shown in Figure 5(c) with happens-before relations shown in Figure 5(d). As shown in Figure 5(a), the whole `for-j` loops are enclosed in `task` constructs; the happens-before relations at depth $= 3$ (i.e., $\mathcal{HB}_3^{S1 \to S1}$ and $\mathcal{HB}_3^{S2 \to S2}$) are not subject to task ordering constraints and kept as the initial sequential order. Note it is also possible that some smart compilers detect parallelism in the original codes, e.g., Intel compiler could detect vector parallelism at the innermost level. Even in such cases, our approach can fully utilize explicit parallelism without missing any compiler-detected parallelism.

D. *PolyAST: a loop optimizer integrating polyhedral and AST-based transformations*

For the performance evaluation in Section V, we used the PolyAST [4] framework to perform loop optimizations, where the dependence information provided by the proposed approach is passed as input. PolyAST employs a hybrid approach of polyhedral and AST-based compilations; it detects reduction and doacross parallelism [26] in addition to regular doall parallelism. In the code generation stage, doacross parallelism can be efficiently expressed using the proposed doacross pragmas in OpenMP 4.1 [17], [27]. These pragmas allow for fine-grained synchronization in multidimensional loop nests, using an efficient synchronization library [28].

The transformed code of Jacobi kernel (Figure 5(a)) based on dependence polyhedra Figure 5(e) is shown in Figure 5(b). The `ordered(2)` at line 1 specifies the nest level to place `ordered depend` directives. The `ordered depend(sink:` *vec*`)` at line 4 can be viewed as a blocking operation that waits for the completion of iteration *vec*, e.g., `(c1, c3-1)`, while the `ordered depend(source)` at line 14 can be viewed as an unblocking operation to indicate that the current iteration `(c1, c3)` has completed. Thanks to the accurate dependence information at depths 1 and 2, outermost and secondary nested loops were skewed and parallelized using doacross extensions [27], [4] while the innermost loops were kept as the original because of the conservative dependence at depth 3. Due to space limitations, we omitted loop tiling at first and second nest levels although the permutability after skewing guarantees tiling.

V. Experimental Evaluation

In this section, we present the evaluation of our approach. We begin with an overview of the experimental setup and benchmark descriptions used in the evaluation. Then we discuss the experimental results and conclude with a summary.

A. *Experimental setup*

	Intel Xeon 5660 (Westmere)	**IBM Power 8E (Power 8)**
Microarch	Westmere	Power PC
Clock speed	2.80GHz	3.02GHz
Cores/socket	6	12
Total cores	12	24
L1 cache/core	32 KB	32 KB
L2 cache/core	256 KB	512 KB
L3 cache/socket	12 MB	8 MB
Compiler	gcc/g++ -4.9.2 icc/icpc -14.0	gcc/g++ -4.9.2
Compiler flags	-O3 -fast(icc)	-O3
Linux kernel	2.6.32	3.13.0

Table I: Details of architectures used for experiments.

Platform: Our evaluation uses two different multi-way SMP multicore setups: an Intel Westmere and a IBM Power8 system. Table I lists their hardware specifications. On both architectures, GCC-4.9.2 is used for all benchmarks as it supports OpenMP 4.0 specifications. On the Intel Westmere, the Intel C and C++ compiler (version-14.0) is also used for evaluation of the Rodinia suite. But, this compiler doesn't support OpenMP 4.0 `task depend` clauses and hence it is not used for evaluation of the KASTORS suite. On our IBM Power8 machine, the IBM XLC compiler is currently unavailable for the experiments. Note that our results include the -fast option for icc, but not the -Ofast option for gcc; this is not a significant issue because we do not use these results to compare icc vs. gcc performance.

Benchmarks and experimental variants: We used the KASTORS and the Rodinia suites to evaluate our approach. Benchmarks in these suites cover OpenMP loop and task constructs. Also, these benchmarks have various data access patterns such as affine array subscripts, linearized array subscripts, indirect array subscripts, unrestricted pointer aliasing and unknown function calls. Table II summarizes problem sizes used for each benchmark. The table also includes the sequential execution times for the benchmarks while using different

Suite	Benchmark name	Manual modifications to source	Problem Size	Sequential Exec time (Sec)			Transformations by PoPP
				Intel Westmere		IBM Power8	
				ICC	GCC	GCC	
Kastors	Jacobi	OF	Matrix size: 2K Time iterations: 200	-	4.412	4.914	F, S, T, D
	Jacobi-Blocked	D	Matrix size: 2K Time iterations: 200	-	5.838	6.241	F, S, D
	Sparse LU	D, OF	Matrix size: 100 Block size: 25	-	1.632	2.284	F, D
Rodinia	Back prop.	AP	Layer size: 5 Million	1.660	1.659	0.705	P
	CFD Solver	-	file: fvcorr.domn.097K	0.002	0.002	0.015	-
	Hotspot	AT, F	Matrix size: 8K Time iterations: 12	5.828	19.385	12.532	F, S, T, D
	Kmeans	-	Clusters: 5 Attributes: 34	2.484	4.914	7.061	-
	LUD	APR	Matrix size: 2K	7.866	8.633	30.471	P
	Needle-Wunch	AT	Matrix size: 8K	1.962	1.964	8.603	P, T, D
	Particle filter	R	Size: 10K	0.341	0.603	0.920	F
	Path finder	-	Size: 100K, Time iterations: 100	0.208	0.030	0.066	-

Table II: Sequential execution times of KASTORS and Rodinia on Intel Westmere and IBM Power 8 systems along with problem sizes. Intel ICC-14.0 compiler doesn't support OpenMP 4.0 task depend constructs. So, no execution time is reported for KASTORS on Intel platform with ICC compiler. Transformations exposed by PoPP: Permutation (P), Fusion (F), Skewing (S), Tiling (T), Doacross pipelined parallelism (D), No further optimizations (-). Manual modifications performed before passing to PoPP: Replace complex if-statements by closures i.e., outlined functions (OF), Delinearization on task-depend variables (D), Function inlining (F), Annotated inner loop as parallel (AP), Annotated inner loop as parallel with array reductions (APR), Annotated with task-depend constructs (AT), Removal of printf statements (R), No modifications (-).

compilers on each platform. In all experiments, we report the mean execution time measured over 10 runs repeated in the same environment for each data point.

In the following experiments, we compare two experimental variants: OpenMP to show the original OpenMP parallel version running with all cores - i.e., 12 cores on Westmere and 24 cores on Power8 - and PoPP to show the transformed version by our framework running with all cores. The speedup of a program is defined as the execution time of the serial version of the program divided by the execution time of the parallel version of the program.

B. KASTORS Suite

KASTORS suite is designed to evaluate the efficiency of OpenMP 4.0 task dependences [13]. This suite consists of five benchmarks namely Jacobi, Jacobi-Blocked, SparseLU, Strassen and Plasma. Our implementation is currently unable to compiler Strassen due to its use of recursive calls with tasks, and Plasma due to its use of C structs. As a result, we only provide results for Jacobi, Jacobi-Blocked and SparseLU from the KASTORS suite. Support for recursive task parallelism and for supporting C structs are topics for future work.

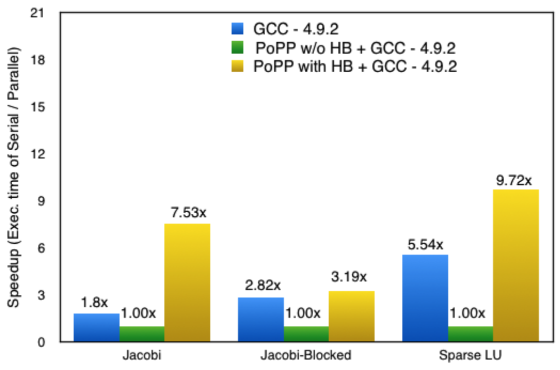
Jacobi & Jacobi-Blocked (Poisson2D): The kernel of Poisson2D is the Jacobi example discussed in Section IV-D and Poisson2D - Blocked is the version where loop tiling/blocking is already applied in the OpenMP version. In both versions, the PoPP framework utilized the explicit parallelism and applied loop fusion, skewing, tiling (only to non-blocked version) and doacross parallelization. Figures 6(a) and 6(b) show that PoPP has much better performance than OpenMP for the non-blocked version because of automatic loop tiling; it also gave some improvements for the blocked version thanks to doacross parallelization.

SparseLU. This benchmark computes LU decomposition of given sparse matrix. The computation kernel is a triply nested imperfect loop nest, which contains four kinds of function calls with linearized (i.e., non-affine) array subscripts. In the OpenMP version, each function call is annotated by `task depend` constructs to implement task parallelism with inter-task dependences.

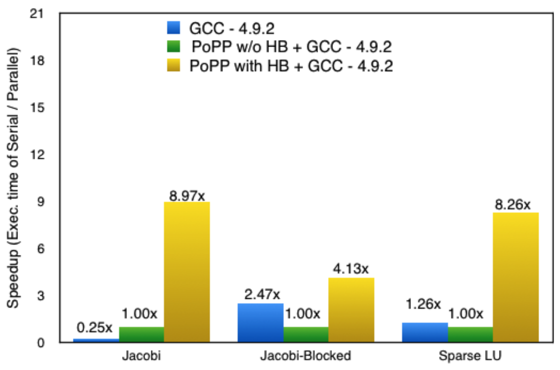
To the input kernel, we manually applied de-linearization [19] technique, which is not yet supported in the current framework. As described in Section IV-A, our dependence analyzer handled these function calls enclosed in non-affine `if`-statements[4] and provided conservative dependence information. Further, the proposed approach computed exact dependence information by intersecting with the happens-before relations obtained from `task depend` constructs. The PoPP framework applied loop fusion to make a perfect loop nest and parallelized the outermost loop as doacross, as with Jacobi kernel discussed in Section IV-D.

Figures 6(a) and 6(b) summarize the speedup compar-

[4]In preprocessing phase, `if`-statements are moved into the innermost levels so that loops are free from non-affine control flows.



(a) Intel Westmere with 12 cores



(b) IBM Power8 with 24 cores

Figure 6: Evaluation of the KASTORS suite (using GCC compiler). Sequential times are reported in Table II. Original benchmark speedup is compared against with optimized codes from PoPP with/ without considering happens-before (HB) relations.

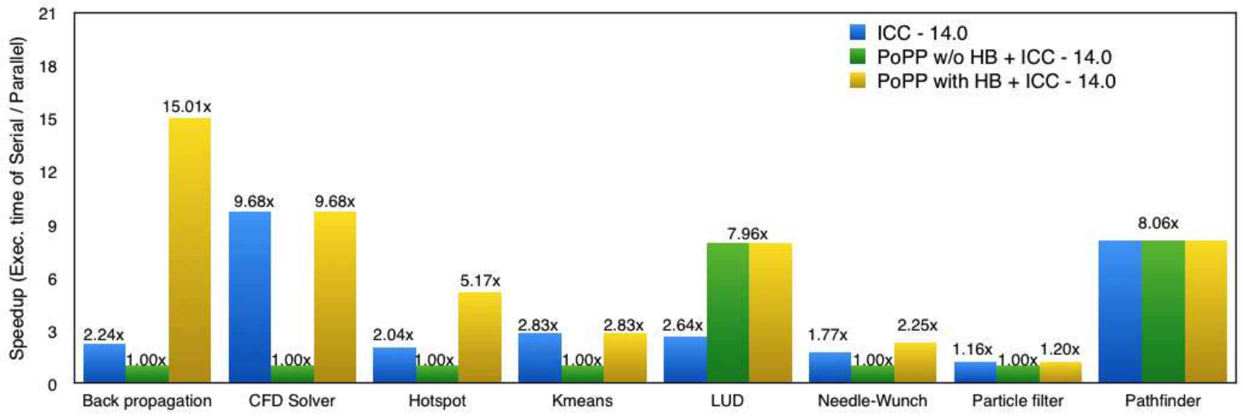


Figure 7: Evaluation of the Rodinia suite (using Intel compiler) on Intel Westmere with 12 cores. Sequential times are reported in Table II. Original benchmark speedup is compared against with optimized codes from PoPP with/ without considering happens-before (HB) relations.

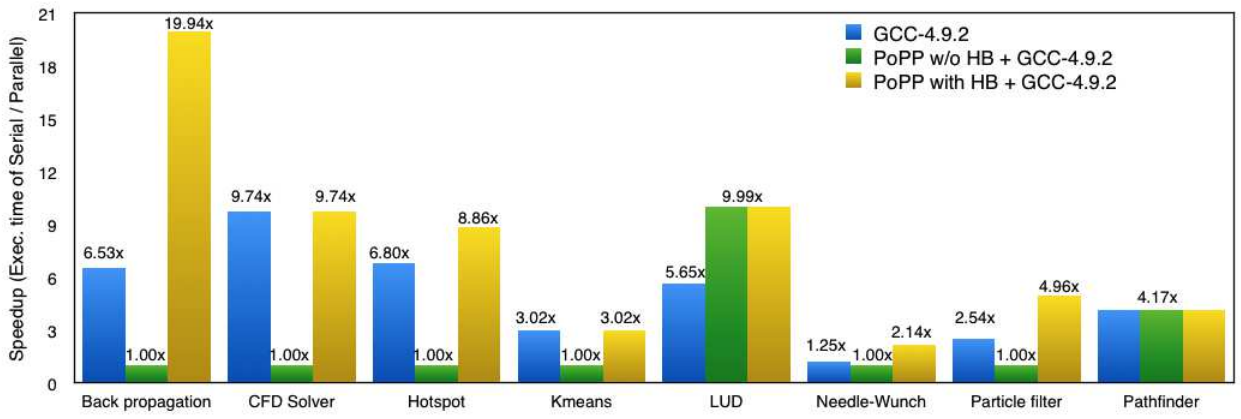


Figure 8: Evaluation of Rodinia suite (using GCC compiler) on Intel Westmere with 12 cores. Sequential times are reported in Table II. Original benchmark speedup is compared against with optimized codes from PoPP with/ without considering happens-before (HB) relations.

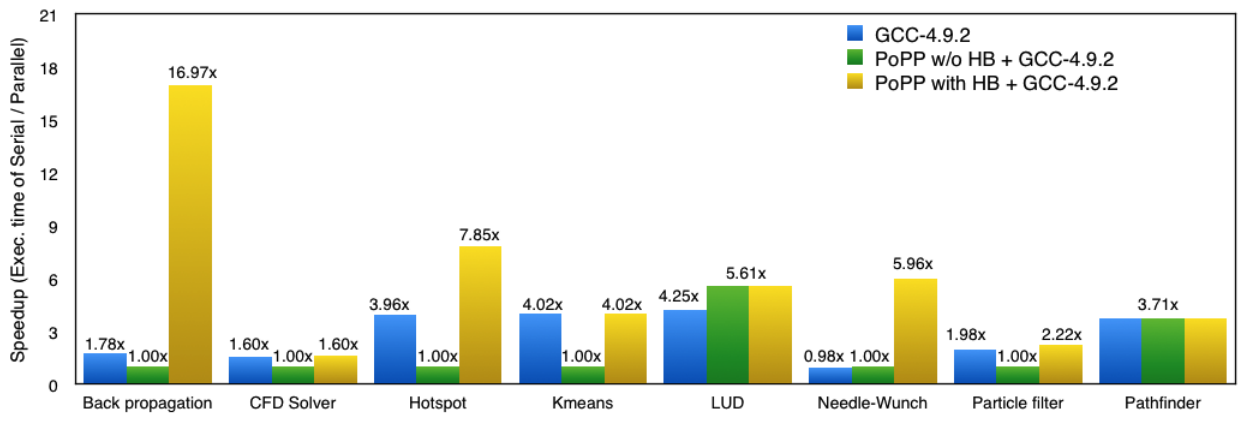


Figure 9: Evaluation of the Rodinia suite (using GCC compiler) on IBM Power8 with 24 cores. Sequential times are reported in Table II. Original benchmark speedup is compared against with optimized codes from PoPP with/ without considering happens-before (HB) relations.

ing with sequential execution on Westmere and Power8, which show that PoPP improved performances from 3.19× to 9.72× on Westmere and from 4.13× to 8.97× on Power8, respectively. This improvement is due to the synchronization efficiency of doacross parallelization. Although the possible dependence patterns of doacross parallelism is subset of the task constructs, the loop-based point-to-point synchronizations of doacross generally have quite small synchronization overhead. By using our framework, programmers can specify ideal task dependences regardless of the overhead and the framework applies a sequence of transformations and converts into the efficient doacross implementations when possible.

C. *Rodinia Suite*

Rodinia suite is designed for heterogeneous computing and it includes kernels which target towards multi-core CPUs and GPU platforms [14]. The suite consists of 18 benchmarks and include diverse applications such as dynamic programming techniques, linear algebra kernels, graph traversals, structure grid, unstructured grid, etc. In the current evaluation, we consider eight benchmarks namely Back propagation, CFD Solver, Hotspot, Kmeans, LU decomposition, Needleman-Wunch, Particle filter, and Pathfinder. The other 10 benchmarks contain C structs, which are not yet supported in the proposed polyhedral framework. We will address these benchmarks in future work.

Back propagation. This benchmark is a machine-learning algorithm that trains the weights of connecting nodes on a layered neural network. This benchmark has two functions each of which contains an OpenMP parallelized doubly nested loops[5], which is the source of parallelism in this benchmark. However, because of the unrestricted pointer aliasing among function arguments (2D pointer-to-pointer arrays), this loop parallelism is impossible to detect without sound inter-procedural pointer analysis. Run-time checking for the absence of aliasing without inter-procedural analysis can be expensive in this benchmark even though there exists speculative approaches [18] and compiler flags to identify pointer aliases. Alternatively, our framework utilized the happens-before relations derived from parallel loop constructs; based on the improved dependence information, our framework applied loop permutation to the kernel loops so that the resulting kernels have better spatial data locality to enhance cache reuse and vectorization. As can be seen from Figures 7, 8, and 9, PoPP versions show much better speedup than the OpenMP versions on both systems.

[5]For evaluation, we also specified the inner loop's parallelism.

Other benchmarks. As shown in the figures, we also observed performance improvements for Hotspot, LU Decomposition, Needle-Wunch, and Particle filter, by the PoPP framework. Based on the improved dependence information via happens-before relations, the PoPP applied loop fusion and/or permutation to improve spatial data locality to these five benchmarks. Note that Rodinia benchmarks aim at parallelization on accelerators such as GPUs while the optimization criteria in PolyAST framework are customized for CPU cache architectures and thereby enabled these transformations. Also, we removed some print statements as summarized in Table II, which provided more opportunities for transformations. For evaluation of Hotspot and Needle-Wunch, although the OpenMP variants reported in Figures 7, 8, and 9 are the original loop parallel versions, we converted these loop constructs into `task depend` constructs and passed to PoPP framework. Loop skewing (only to Hotspot), tiling and doacross parallelization were applied automatically on these benchmarks by PoPP to enable better data reuse and synchronization efficiency. No transformations were applied to CFD Solver, Kmeans, and Pathfinder; same performance was observed between OpenMP and PoPP. The overall experimental results show geometric mean performance improvements of 1.62x and 2.75x on the Intel Westmere and IBM Power8 platforms respectively, relative to the original OpenMP versions.

VI. Limitations

This section describes the limitations imposed on the current framework and briefly discusses how we can address such restrictions in the future work.

The algorithmic limitations are as follows:

- Intersection of conservative dependence with happens-before relations is applicable only in the case of programs that satisfy serial-elision property. This limitation is also due to the underlying polyhedral representations that support only sequential - i.e., total ordered - schedules. In our future work, we will address the parallel constructs that don't satisfy serial-elision property, e.g., *SPMDized code with explicit OpenMP threads and barriers*, by extending data dependence definition with happens-before relations for ordering. There is certain amount of related work in this direction [29], [30].
- In this work, we consider only nested tasks all of which are included in the same lexical scope without recursive calls. On the other hand, the generic OpenMP task parallel constructs support wider range of parallelism including arbitrary patterns of dynamic task parallelism. We plan to address recursive task patterns (e.g., observed in Strassen benchmark in KASTORS suite) by an hybrid approach of polyhedral and graph-based optimizations, which also have a long history including the Sisal language [31].
- In the current framework, we implement the detected doall and doacross parallelism by the `#pragma omp for` and `#pragma omp for ordered` constructs in the code generation phase. In our future work, we will also support task parallel constructs including inter-task dependences in the codegen phase and appropriate cost models to determine which construct of task dependence or doacross is more beneficial in the parallelization phase. In general, doacross construct has less synchronization overhead while task dependence is more robust over load unbalance.

The implementation limitations are as follows:

- In this paper, we limit our analysis to support only doall (`#pragma omp for`) parallelism and task parallelism including inter-task dependence (`task depend` constructs). We will extend our framework so that other parallel constructs in OpenMP, such as *sections* that do satisfy serial-elision property, can be expressed as happens-before relations. Once they are converted into HB relations in step 3 of Algorithm 1, the remaining steps seamlessly reflect such parallelism in the final dependence information.
- We extend our approach to handle *C structs* by encoding the fields of *structure* onto separate names and extend dependence analysis accordingly, as with access relations in ISL library [32].

VII. Related Work

There is an extensive body of literature on applying polyhedral transformations to non-affine static program regions. We focus on past contributions that are most closely related to this paper. The comparison between our approach and other related work is discussed in [12].

PENCIL [33], a platform-neutral compute intermediate language, aimed at facilitating automatic parallelization and optimization on multi-threaded SIMD hardware for domain specific languages. The language allows users to supply information about dependences and memory access patterns to enable better optimizations. PENCIL provides directives such as *independent, reductions* to remove data dependences on the loop, but doesn't have support for task directives as in our approach. Another key difference from our approach is that we are interested in general-purpose parallel languages such as OpenMP while PENCIL is focused on supporting DSLs in which certain coding rules are enforced related to pointer aliasing, recursion, unstructured control flow. There is a similarity in the semantics of the *independent* pragma from PENCIL and the *parallel for* pragma from OpenMP, as they both indicate no dependences among loop iterations.

Pop and Cohen have presented a preliminary approach to increase optimization opportunities for parallel programs by extracting the semantics of the parallel annotations [34]. This extracted information is brought into compiler's intermediate representation and leverage existing polyhedral frameworks for optimizations. They envisaged on considering streaming OpenMP extensions carrying explicit dependence information, to enhance the accuracy of data dependence analyses.

A number of papers addressed the problem of dataflow analysis of explicitly parallel programs, including extensions of array data-flow analysis to data-parallel and/or task-parallel programs [29], and adaptation of array data-flow analysis to the X10 programs with finish/async parallelism [30]. In these approaches, the happens-before relations are first analyzed and the data-flow is computed based on the partial order imposed by happen-before relations. On the other hand, our approach first overestimates dependences based on the sequential order and intersect the happen-before relations with the conservative dependences. The main focus of our work is on transformations of explicitly parallel programs for improved performance, whereas the work in [29] and [30] is only focused on analysis.

There has also been work done in partitioned global address space languages such as Co-Array FORTRAN (CAF) and Unified Parallel C (UPC), where certain compiler optimizations have been enabled by introducing language extensions and new synchronization constructs [35]. There has been significant effort to handle certain subsets of non-affine accesses, including delinearization techniques [19] for linearized subscripts, polynomial accesses [36] in the polyhedral model for array dependence analysis and loop transformations.

VIII. Conclusions and Future Work

This work is motivated by the observation that software with explicit parallelism is on the rise. This explicit parallelism can be used to enable larger set of polyhedral transformations by mitigating conservative dependences, compared to what might have been possible if the input program had been sequential. We introduced an approach that reduces spurious dependences from the conservative dependence analysis by intersecting them with the happens-before relations from parallel constructs. The final set of the dependences can then be passed on to a polyhedral transformation tool, such as PLuTo or PolyAST, to enable transformations of explicitly parallel programs.

We evaluated our approach using OpenMP benchmark programs from the KASTORS and the Rodinia benchmark suites. The approach reduced spurious dependences from the conservative analysis of these benchmarks and the resulting dependence information broadened the range of legal transformations in the polyhedral optimization phase. Overall, our experimental results show geometric mean performance improvements of 1.62x and 2.75x on the 12-core Intel Westmere and 24-core IBM Power8 platforms respectively, relative to the original OpenMP versions. The main focus of our future work will be to address the limitations summarized in Section VI.

Acknowledgments

This research was supported in part by the DOE Office of Science Advanced Scientific Computing Research program through collaborative agreement DE-SC0008882. We are also very grateful to the anonymous reviewers on the PACT'15 program committee, and members of the Habanero Extreme Scale Software Research Group at Rice University, for their detailed comments and suggestions on earlier drafts of this paper.

References

[1] S. Borkar and A. A. Chien, "The future of microprocessors," *Commun. ACM*, vol. 54, no. 5, pp. 67–77, May 2011.

[2] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, "Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model," in *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th International Conference on Compiler Construction*, ser. CC'08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 132–146.

[3] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '08. New York, NY, USA: ACM, 2008, pp. 101–113.

[4] J. Shirako, L.-N. Pouchet, and V. Sarkar, "Oil and Water Can Mix: An Integration of Polyhedral and AST-based Transformations." in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '14, 2014.

[5] M. Kong, R. Veras, K. Stock, F. Franchetti, L.-N. Pouchet, and P. Sadayappan, "When polyhedral transformations meet simd code generation," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13. New York, NY, USA: ACM, 2013, pp. 127–138.

[6] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.

[7] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the cilk-5 multithreaded language," in *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, ser. PLDI '98. New York, NY, USA: ACM, 1998, pp. 212–223.

[8] "OpenMP Specifications," http://openmp.org/wp/openmp-specifications.

[9] B. Chamberlain, D. Callahan, and H. Zima, "Parallel programmability and the chapel language," *International Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, 2007.

[10] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," in *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP '95. New York, NY, USA: ACM, 1995, pp. 207–216.

[11] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: An object-oriented approach to non-uniform cluster computing," *SIGPLAN Not.*, vol. 40, no. 10, pp. 519–538, Oct. 2005.

[12] P. Chatarasi, J. Shirako, and V. Sarkar, "Polyhedral transformations of explicitly parallel programs," in *5th International Workshop on Polyhedral Compilation Techniques (IMPACT)*, Amsterdam, Netherlands, Jan. 2015.

[13] P. Virouleau, P. Brunet, F. Broquedis, N. Furmento, S. Thibault, O. Aumage, and T. Gautier, "Evaluation of OpenMP Dependent Tasks with the KASTORS Benchmark Suite," in *Using and Improving OpenMP for Devices, Tasks, and More - 10th International Workshop on OpenMP, IWOMP 2014, Salvador, Brazil, September 28-30, 2014. Proceedings*, 2014, pp. 16–29.

[14] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, ser. IISWC '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 44–54.

[15] P. Feautrier and C. Lengauer, "Polyhedron model," in *Encyclopedia of Parallel Computing*, D. A. Padua, Ed. Springer, 2011, pp. 1581–1592.

[16] D. G. Wonnacott, "Constraint-based array dependence analysis," Ph.D. dissertation, College Park, MD, USA, 1995, uMI Order No. GAX96-22167.

[17] "OpenMP Technical Report 3 on OpenMP 4.0 enhancements," http://openmp.org/TR3.pdf.

[18] J. Doerfert, C. Hammacher, K. Streit, and S. Hack, "SPolly: Speculative Optimizations in the Polyhedral Model," in *Proc. 3rd International Workshop on Polyhedral Compilation Techniques (IMPACT)*, Berlin, Germany, Jan. 2013, pp. 55–61.

[19] T. Grosser, J. Ramanujam, L.-N. Pouchet, P. Sadayappan, and S. Pop, "Optimistic delinearization of parametrically sized arrays," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, ser. ICS '15. New York, NY, USA: ACM, 2015, pp. 351–360.

[20] C. Liao, D. J. Quinlan, T. Panas, and B. R. de Supinski, "A rose-based openmp 3.0 research compiler supporting multiple runtime libraries," in *Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More, 6th Internationan Workshop on OpenMP, IWOMP 2010, Tsukuba, Japan, June 14-16, 2010, Proceedings*, ser. Lecture Notes in Computer Science, M. Sato, T. Hanawa, M. S. Müller, B. M. Chapman, and B. R. de Supinski, Eds., vol. 6132. Springer, 2010, pp. 15–28.

[21] "CANDL: Data dependence analysis tool in the polyhedral model," http://icps.u-strasbg.fr/ bastoul/development/candl.

[22] B. Creusillet and F. Irigoin, "Exact versus approximate array region analyses," in *Proceedings of the 9th International Workshop on Languages and Compilers for Parallel Computing*, ser. LCPC '96. London, UK, UK: Springer-Verlag, 1997, pp. 86–100.

[23] D. Barthou *et al.*, "Fuzzy Array Dataflow Analysis," *J. Parallel Distrib. Comput.*, vol. 40, no. 2, pp. 210–226, 1997.

[24] B. Creusillet and F. Irigoin, "Interprocedural array region analyses," *Int. J. Parallel Program.*, vol. 24, no. 6, pp. 513–546, Dec. 1996.

[25] S. Verdoolaege and T. Grosser, "Polyhedral extraction tool," *Second International Workshop on Polyhedral Compilation Techniques (IMPACT 12), Paris, France*, 2012.

[26] R. Cytron, "Doacross: Beyond Vectorization for Multiprocessors," in *ICPP'86*, 1986, pp. 836–844.

[27] J. Shirako, P. Unnikrishnan, S. Chatterjee, K. Li, and V. Sarkar, "Expressing DOACROSS Loop Dependencies in OpenMP," in *9th International Workshop on OpenMP (IWOMP)*, 2011.

[28] P. Unnikrishnan, J. Shirako, K. Barton, S. Chatterjee, R. Silvera, and V. Sarkar, "A practical approach to DOACROSS parallelization," in *Euro-Par*, 2012, pp. 219–231.

[29] J.-F. Collard and M. Griebl, "Array Dataflow Analysis for Explicitly Parallel Programs," in *Proceedings of the Second International Euro-Par Conference on Parallel Processing*, ser. Euro-Par '96, 1996.

[30] T. Yuki, P. Feautrier, S. Rajopadhye, and V. Saraswat, "Array Dataflow Analysis for Polyhedral X10 Programs," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '07, 2013.

[31] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors*. Cambridge, MA, USA: MIT Press, 1989.

[32] "Integer set library," http://isl.gforge.inria.fr.

[33] R. Baghdadi, A. Cohen, S. Guelton, S. Verdoolaege, J. Inoue, T. Grosser, G. Kouveli, A. Kravets, A. Lokhmotov, C. Nugteren, F. Waters, and A. F. Donaldson, "PENCIL: towards a platform-neutral compute intermediate language for dsls," *CoRR*, vol. abs/1302.5586, 2013.

[34] A. Pop and A. Cohen, "Preserving high-level semantics of parallel programming annotations through the compilation flow of optimizing compilers," in *Proceedings of the 15th Workshop on Compilers for Parallel Computers (CPC'10)*, 2010.

[35] C. Coarfa, Y. Dotsenko, J. Mellor-Crummey, F. Cantonnet, T. El-Ghazawi, A. Mohanti, Y. Yao, and D. Chavarría-Miranda, "An evaluation of global address space languages: Co-array fortran and unified parallel c," in *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '05. New York, NY, USA: ACM, 2005, pp. 36–47.

[36] V. Maslov and W. Pugh, "Simplifying Polynomial Constraints Over Integers to Make Dependence Analysis More Precise," In CONPAR 94 - VAPP VI, Int. Conf. on Parallel and Vector Processing, Tech. Rep., 1994.