`ompVerify`: Polyhedral analysis for the OpenMP Programmer

V. Basupalli[1], T. Yuki[1], S. Rajopadhye[1], A. Morvan[2], S. Derrien[2], P. Quinton[2], D. Wonnacott[3]

[1] Computer Science Department, Colorado State University
{basupall, yuki, svr}@cs.colostate.edu
[2] CAIRN, IRISA, Rennes, France
{antoine.morvan, sderrien, quinton}@irisa.fr
[3] Computer Science Department, Haverford College
davew@cs.haverford.edu

Abstract. We describe a static analysis tool for OpenMP programs integrated into the standard open source Eclipse IDE. It can detect an important class of common data-race errors in OpenMP parallel loop programs by flagging incorrectly specified `omp parallel for` directives and data races. The analysis is based on the polyhedral model, and covers a class of program fragments called Affine Control Loops (ACLs, or alternatively, Static Control Parts, SCoPs). `ompVerify` automatically extracts such ACLs from an input C program, and then flags the errors as specific and precise error messages reported to the user. We illustrate the power of our techniques through a number of simple but non-trivial examples with subtle parallelization errors that are difficult to detect, even for expert OpenMP programmers.

1 Introduction

Parallel programming is a difficult, and the semantic gap between sequential and parallel programming is huge. Automatic parallelization of sequential codes has seen significant headway on program analysis and transformation frameworks, but selection of the optimal transformation remains a difficult challenge. Most authors of parallel software retain "manual control" by parallelizing codes themselves with tools such as OpenMP.

In the sequential world, programmers are now used to type-safe languages and sophisticated tools and Integrated Development Environments (IDEs). They expect their IDEs to provide code refactoring, navigation, dynamic compilation, automatic builds, "instant" and rapid feedback through structured editors and static analysis, as well as debugging support. This paper shows how the infrastructure for automatic parallelization, specifically the "polyhedral model" for program analysis, can be employed to benefit OpenMP programmers. Our work complements recent work on parallel debugging tools[4] [1,2] with effective feedback to programmers about problems that can be identified statically.

Our analysis finds semantic errors of shared memory programs that parallelize loops with the OpenMP `omp for` work-sharing directive. Static bug-finding tools, like debuggers, cannot be trivial extensions of sequential tools: parallel programs have many other issues (e.g., deadlocks and data races), and familiar issues such as reads from uninitialized variables must be extended carefully (i.e., without treating an array as one big scalar). Specifically, we make two contributions.

- Static analysis of programs with parallel loops that verifies that parallel loops do not alter the program semantics, together with precise characterizations of where/when semantics are violated.

[4] See also DDT (`http://www.allinea.com/?page=48`) and the Sun Thread Analyzer (`http://docs.sun.com/app/docs/doc/820-0619`).

– Integration of this analysis and other instance/element wise warnings about the parallelization into the Eclipse IDE. In doing this, we are able to provide very precise and specific error messages, and we also have all the necessary information to provide concrete counterexamples.

The remainder of this paper is organized as follows. Section 2 describes the "polyhedral model", a mathematical framework that underlies our analysis. Section 3 motivates our work with a number of examples. Section 4 presents how polyhedral analysis is used for error detection and correction. Section 5 describes how we are integrating our analysis into Eclipse, a widely used, open source IDE. Section 7 contrasts our work with related work. Finally, we conclude with future directions in Section 8.

2 The Polyhedral Model

Our analysis is based on the polyhedral model, a formalism developed for reasoning about programs that manipulate dense arrays. This framework lets us reason at an appropriately "fine-grained" level, i.e., about specific *elements* of arrays and *instances* of statements. A detailed review of this work [3–11] is beyond the scope of this paper, but this section summarizes key, important concepts. A detailed comparison of the vocabularies and notations is probably the many-authored Wikipedia page frameworks [12].

2.1 Affine Control Loops

The polyhedral model provides powerful static analysis capabilities for a class of programs called Affine Control Loops (ACLs, also called Static Control Parts or *SCoPs*). ACLs include many scientific codes, such as dense linear algebra and stencil computations, as well as dynamic programming, and covers most of three of the "Berkeley View motifs" [13] proposed by researchers in multi-core parallel computation and shares much with a fourth.

Control flow in an ACL can include arbitrary nesting of `for` loops containing assignment statements; data references can include scalars and arrays. The definition of ACL requires that all loop step sizes must be known at compile time and all loop bound and subscript expressions must be affine functions (linear plus a known constant) of the surrounding loop index variables and a set of symbolic constants.

Fig. 2.1 shows an ACL from dense linear algebra, the forward substitution kernel. The labels S_1 etc. in the left margin are for reference and not part of the code itself. Note that the loops are not *perfectly nested*, with the outer (`i`) loop containing simple statements and a loop; The loop bound expressions and subscripts are not only affine, but very simple, though often symbolic rather than known constants (e.g., the bounds `0` and `N` in the outer loop, and `0` and `i` in the inner loop, as well as subscripts like `i` and `j`, or `i-1` and `j-1` in a later example).

Our goal is to reason about specific *instances* (individual executions) of statements in ACLs. If we were given values for all symbolic constants, we could represent every statement instance explicitly or draw a diagram like that shown on the right of Fig. 2.1, and it is often easiest to visualize programs in these terms. To reason about statement instances in the presence of unknown symbolic constants, we will make use of operations on (potentially infinite) sets, which we will define in terms of affine constraints and manipulate with the ISL Library [14].

```
   #pragma omp parallel for private(j)
   for (i = 0; i < N; i++) {
S1:   x[i] = b[i];
      for (j = 0; j < i; j++)
S2:      x[i] = x[i] - L[i][j]*x[j];
S3:   x[i] = x[i]/L[i][i];
   }
```

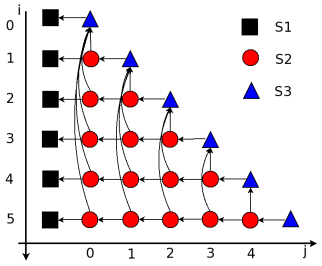

Fig. 1. Forward Substitution Code, and its Iteration Space for N=6. The parallelization is incorrect, but for now just think of this as a sequential program.

2.2 Statement Domains and Order of Execution

We use the term *domain* to describe the set of iteration points of the loops surrounding the statement. When the identity of the statement is clear from context, each point in this domain is defined by the values of the loop indices. For Fig. 2.1, if N=6, the domain of S_1 is $\{(0),(1),(2),(3),(4),(5)\}$, or more concisely $\{i|0 \leq i < 6\}$. That of S_2 is the triangle of red dots, i.e., $\{i,j|1 \leq i < 6 \wedge 0 < j < i\}$. Note that we often name set index variables for the corresponding loop indices, but this is not necessary.

If we wish to identify any statement in a program, we interleave these loop index values with constants defining the textual order at a given loop level, producing a vector of length $2k+1$ for a statement inside k nested loops. for example iteration (4) of S_1 can be identified $(0,4,0)$, meaning "initial statement (i.e., statement 0) of the program, iteration 4, initial statement therein". Iteration (i) of S_1 can likewise be identified $(0,i,0)$, and Iteration (x) of S_3 identified $(0,x,2)$, and Iteration (i,j) of S_2 as $(0,i,1,j,0)$.

The latter "$2k+1$" notation often contains information that is redundant with the program text: in this example the leading value will always be 0. However, this notation facilitates reasoning about the order of execution, as a domain element I is executed before J if and only if I precedes J in lexicographic (dictionary) order. In the less redundant notation, we cannot reason about the ordering of points in the space without reference to properties about the program structure, such as the number of loops surrounding both statements.

As noted above, we reason about sets of unknown size using constraints on symbolic variables, so the domain of S_1 is $\{(0,i,0)|0 \leq i < N\}$ (or $\{(i)|0 \leq i < N\}$ if we know from context we refer to S_1). Similarly, $Domain(S_2) = \{(0,i,1,j,0)|1 \leq i < N \wedge 0 < j < i\}$ and $Domain(S_3) = \{(0,i,2)|0 \leq i < N\}$.

2.3 Transforming Order of Execution

Equating execution order with lexicographic order of points in the iteration spaces, creates a simple mechanism for describing reordering transformations: simply rearrange the points in the iteration space. For example, if we wish to move S_2 to put it after S_3, we could replace the constant 1 in S_2's domain with a 3. More formally, we describe this as applying the *Space Time Map* $((0,i,1,j,0) \rightarrow (0,i,3,j,0))$. Note that some transformations (such as the above) may affect the *result*, i.e., may be illegal. The goal of parallelization is usually to improve performance *without* affecting result, and the polyhedral model can also be used to reason about legality.

A Space Time Map can be used to reorder iterations of a statement or split iterations into multiple statements as well as change the order of statements. For example, we could reverse order of

the `j` loop for S_2 with $((0, i, 1, j, 0) \rightarrow (0, i, 1, i - j, 0)$ or reverse the order of the `i` loop by replacing i with $N - i$ in all three Space Time Maps.

Concurrent execution can be indicated in a number of ways; we choose to simply flag certain dimensions of Space Time Maps as parallel (indicated in our documents with underlined dimensions). Thus, executing Fig. 2.1 in original order but with the outer loop parallel would be described as (we often use the same number of dimensions for all statements).

$S_1 : ((0, i, 0, 0, 0) \rightarrow (0, \underline{i}, 0, 0, 0))$
$S_2 : ((0, i, 1, j, 0) \rightarrow (0, \underline{i}, 1, j, 0))$
$S_3 : ((0, i, 2, 0, 0) \rightarrow (0, \underline{i}, 2, 0, 0))$

2.4 Memory Maps

We can also describe the relation of statement instances to the memory cells they read or update with a notation that is similar to that of a Space Time Map. We use the term *Memory Map*s for such mappings, and distinguish them visually from Space Time Maps by giving an array name in the range of the mapping. In Fig. 2.1, the memory maps for the array writes in S_1, S_2 and S_3 are (respectively) $((0, i, 0, 0, 0) \rightarrow x[i])$, $((0, i, 1, j, 0) \rightarrow x[i])$, and $((0, i, 2, 0, 0) \rightarrow x[i])$. Similar maps can describe reads. Note that, unlike Space Time Maps, Memory Maps are frequently many-to-one (since many iterations may read from and/or write to the same location). Memory Maps will play a key role in our detection of data races.

2.5 Dependences and Legality of Transformation

As noted above, program transformations may or may not affect the program result. The key to determining whether or not the result has been corrupted is the effect of the transformation on the program's *dependences*. Data dependences are ordering constraints arising from flow of information and/or reuse of memory. Traditional compilers reason about dependences among statements, but in the polyhedral model, we reason about dependences among statement *instances*. The iteration space diagram in Fig. 2.1 shows the inter-iteration ordering constraints that arise from the flow of information in the forward substitution code (assuming we do not allow reordering of floating-point additions).

We represent dependences as relations, for example from a statement instance that reads from a memory cell to the statement instances that write to that cell. Fig. 2.1, the relation $\{(0, i, 1, j, 0) \rightarrow (0, i', 0, 0, 0) | i' < i \wedge j = i'\}$ describes the relation from iteration (i, j) of S_2, which reads from `x[j]`, to those earlier iterations $(i' < i)$ of S_3 that write to the same element of `x` (as `x[i]`, so $(j = i')$). This corresponds to the vertical arrows in Fig. 2.1.

The polyhedral model can manipulate these memory-aliasing relations to compute the one-to-one *dependence function* that gives the source iteration from the domain of the dependence from the *domain* of the dependence. For the $S_2 \rightarrow S_3$ example, the dependence function is $((0, i, 1, j, 0) \rightarrow (0, j, 0, 0, 0))$ and the domain $i, j | 1 \leq i < N \wedge 0 < j < i$. This framework can also separate simple memory aliasing from actual flow of values, for example showing only a single chain of arrows in the horizontal dimension of Fig. 2.1 rather than arrows from each circle to all statement instances to its left (which all write to the same `x[i]`).

Table 1 gives the dependence functions and domains for the flow of information in Fig. 2.1 (note we have omitted the constant levels in dependences to save space). Entry 4 corresponds to the vertical arrows of Fig. 2.1, and Entries 2 and 3 to the horizontal arrows to the S_2 (circle) instances; Entries 6 and 7 show information flow to S_3 (triangles); and Entries 1, 5, and 8 show that all reads

$Number$	$Edge/Dependence$	$Dependence function$	$Domain$
1	$S_1 \rightarrow b(input)$	$((i) \rightarrow (i))$	$\{i \mid 0 \leq i < N\}$
2	$S_2 \rightarrow S_1$	$((i,j) \rightarrow (i))$	$\{i,j \mid 1 \leq i < N \wedge j = 0\}$
3	$S_2 \rightarrow S_2$	$((i,j) \rightarrow (i,j-1))$	$\{i,j \mid 1 \leq j < i < N\}$
4	$S_2 \rightarrow S_3$	$((i,j) \rightarrow (j))$	$\{i,j \mid 0 \leq j < i < N\}$
5	$S_2 \rightarrow L(input)$	$((i,j) \rightarrow (i,j))$	$\{i,j \mid 0 \leq j < i < N\}$
6	$S_3 \rightarrow S_1$	$((i) \rightarrow (i))$	$\{i \mid i = 0\}$
7	$S_3 \rightarrow S_2$	$((i) \rightarrow (i,i-1))$	$\{i \mid 1 \leq i < N\}$
8	$S_3 \rightarrow L(input)$	$((i) \rightarrow (i,i))$	$\{i \mid 0 \leq i < N\}$

Table 1. Edges ofthe PRDG for the Forward Substitution example of Fig. 2.1

to `L` and `b` are upward-exposed past the start of our ACL. This complete description of inter-instance data flow information is known as a *Polyhedral Reduced Dependency Graph* or PRDG.

There may be multiple PRDG edges for a single array read expression. For example, the value of `x[i]` read in S_2 may come from S_1 or S_2.

A program transformation will preserve the result (hopefully while improving performance) if it *satisfies* all dependences. A dependence is considered to be satisfied if the *the time stamp of the producer is before the time stamp of the consumer* in the transformed execution order (as it must have been, by definition, in the original sequential program). While polyhedral dependence analysis and program transformation were developed for automatic parallelization of sequential codes, these tools also let us reason about manual parallelization with OpenMP.

3 Motivating Examples

We now present a number of examples were we detect errors related to conflicting access to memory (data races) in array variables (we consider scalars as special, zero-dimensional arrays). In all the examples, the explanation is subtle and may require some careful analysis of the program by the reader.

We have already seen the forward substitution example (Fig. 2.1) where we claimed that the parallelization specified by the program on the first line was incorrect. The reason for this is that the reference to `x[j]` in S_2 would read values written by statement S_3 in different iteration(s) of the `i` loop. Thus, parallel execution of the iterations of the i loop creates a data race.

"Stencil computations" occur in many codes for modeling physical phenomena. The Jacobi stencil computation uses the values from the previous time step to update the current one. The Gauss-Seidel method converges faster (and also uses less memory) by storing a single array and using some values from the current time step. The example in Fig. 2 illustrates a hybrid Jacobi-Gauss-Seidel 5-pt stencil. The sequential program uses results from the current time step for only one of the dependences—in standard 5-pt Gauss-Seidel, two points are used from the current time step.

The parallelization shown in Fig. 2 is incorrect, as is evident from the inter-iteration dataflow. The access to `A[i-1][j]` in iteration (t,i,j), should read the value placed in element $i-1,j$ of `A` by the write to `A[i][j]` in iteration $(t,i-1,j)$. If the `i` loop is marked as parallel, as in Fig. 2, this value comes from a different thread, creating a data race. If, instead, the j loop were parallel, these (and all other) values would flow only within a thread, and the parallelism would be correct. Süß and Leopold describe such mistakes as the most common and severe form of OpenMP errors [15].

```
//Initialize B
for (t = 0; t < T; t++)
   #pragma omp parallel for private(j)
   for (i = 0; i < N; i++)
      for (j = 0; j < N; j++)
S1:          A[i][j] = (B[i][j] +
              A[i-1][j] + B[i+1][j] +
              B[i][j-1] + B[i][j+1])*0.2;
   //Swap A and B
```

Fig. 2. 5-pt Hybrid Jacobi-Gauss-Seidel stencil computation. The reference `A[i-1][j]` on the right hand side reads a memory location that is written by different iterations of the `i` loop. Hence the parallelization is illegal.

Fig. 3 shows a matrix transpose program with nested parallelism. Here, element-wise dependence analysis reveals that there is no dependence carried by either of the two loops, and thus the given parallelization is legal.

```
#pragma omp parallel private(p1,p2,temp)
{
#pragma omp for
for (p1 = 0; p1 < N; p1++)
   #pragma omp for
   for (p2 = 0; p2 < p1; p2++)
S1:   temp = A[p1][p2];
S2:   A[p1][p2] = A[p2][p1];
S3:   A[p2][p1] = temp;
}
```

Fig. 3. Matrix transpose with nested parallelism. This parallelism is legal but requires element-wise analysis to validate.

Another common mistake [15] is forgetting to make variables private. This is a variation of the data race discussed above, where the access pattern exhibits specific characteristic so that privatization can be a solution.

Consider Fig. 4, a Jacobi stencil example from the OpenMP website (converted to C and simplified for brevity). If the `private` clause was removed, OpenMP treats `resid` as a shared variable. Then the value could be overwritten by other threads before it is used in subsequent statements S_2 and S_3. Similarly, if the `reduction` clause is removed, the value of `error` is not guaranteed to be reflected to other threads until the i loop is fully executed.

Privatization is a well studied concept, and other analyses can also detect variables that need to be privatized. However, the polyhedral analysis can also detect *arrays* that must be privatized. Fig. 5 is an example taken from [16], a 160 line loop from BT in NAS parallel benchmark. The array `TM` is written and used by every iteration of the k loop, but in an independent fashion. Thus, it is legal for the k loop to be parallel if each thread has its own copy of `TM`.

```
#pragma omp for private(j, resid) reduction(+:error)
for (i = 1; i < N; i++) {
   for (j = 1; j < M; j++) {
S1:   resid = (uold[i][j] +
           uold[i-1][j] + uold[i+1][j] +
           uold[i][j-1] + uold[i][j+1])/b;
S2:   u[i][j] = uold[i][j] - omega * resid;
S3:   error = error + resid*resid
   }
}
```

Fig. 4. Simplified version of an iteration of Jacobi stencil computation taken from OpenMP website. Removing `private` or `reduction` clause will cause data races for variables `resid` or `error` respectively.

```
#pragma omp parallel for private(m,n)
for (k = 2; k < NZ; k++) {
   for (m = 1; m <= 5; m++)
      for (n = 1; n <= 5; n++)
S1:          TM[1][m] = ...
S2:          TM[2][m] = ...
S3:          TM[3][m] = ...
S4:          TM[4][m] = ...
S5:          TM[5][m] = ...
   ...
   for (m = 1; m <= 5; m++)
      for (n = 1; n <= 5; n++)
 S6:         ... = TM[n][m];
}
```

Fig. 5. Example taken from [16], originally from BT in NAS parallel benchmark. Array `TM` must be declared private for the parallelization to be correct.

4 Analysis

During the initial dependence extraction, the work-sharing directives (i.e., `omp for`) are ignored, and the program is assumed to be purely sequential. This gives the analyzer the dependences in the original program, represented in a PRDG.

Next, we view OpenMP work-sharing directives as prescribing a program transformation that *changes* the execution order by assigning new time-stamps to the statement instances in the program. The main task of the verifier is to ensure that this transformation does not introduce any data-races. We address three types of races: (i) *causality*, i.e., violation of (true) dependences, (ii) *write conflicts*, i.e., the same memory location is written "simultaneously" by multiple threads, and (iii) *overwrite conflicts*, when a value read from a shared memory location is incorrect because of an intervening update to that location by an other thread.

Algorithm 1 Detection of dependence violation by parallel loops

Require:
E : edges of the PRDG of a loop nest
$P(e, d)$: Function to query if a loop is marked as parallel in Space Time Map
{returns true if dth dimension of the domain of the dependence e corresponds to a parallel loop}
Ensure: All dependences in E are satisfied
E' : list of edges that are violated
D_e : domain of an edge e (dependence polyhedron)
I_e : dependence function of an edge e

$E' \leftarrow \emptyset$
for all $e \in E$ **do**
  for $d = 1$ to max dimension of D_e **do**
    if $P(e, d)$ **then**
      if $\exists z \in D_e, z_d \neq I_e(z)_d$ **then**
        dependence is violated
        $E' \leftarrow E' + e$
        proceed to next e
      end if
    else
      if $\forall z \in D_e, z_d > I_e(z)_d$ **then**
        dependence is satisfied
        proceed to next e
      end if
    end if
  end for
end for
return E'

4.1 Causality

Obviously, the sequential program always satisfies all the dependences, because the loops are executed in lexicographic order. With the addition of a work-sharing directive, however, this legality condition may no longer be true, specifically, if the source of the dependence is from another iteration of the same *parallel* loop. For each dependence in the PRDG, the analyzer checks this condition using Algorithm 4.1.

We now illustrate this for the forward substitution example in Fig. 2.1. The edges of its PRDG are listed in Table 1. The Space Time Maps for each statement in the program are

$S_1 : ((0, i, 0, 0, 0) \rightarrow (0, \underline{i}, 0, 0, 0))$
$S_2 : ((0, i, 1, j, 0) \rightarrow (0, \underline{i}, 1, j, 0))$
$S_3 : ((0, i, 2, 0, 0) \rightarrow (0, \underline{i}, 2, 0, 0))$

To start with, except for dependences where the producer is *input*, the verifier marks all the dependences as unsatisfied. In our example, its the edges $1, 5, 8$ in Table 1 have producers as inputs and are excluded. Starting from the outer-most dimension in the space time map, the verifier checks if any of the dependences satisfy the legality condition. Only dependences that are not satisfied in the current dimension are further checked in subsequent dimensions.

In our example, in dimension 1, all the dependences have the same statement order value 0, implying that all statements have a common outer loop, so none of the dependences satisfy the legality condition. Dimension 2 is annotated as parallel, so according to the legality condition, for

Algorithm 2 Detection of Write Conflicts

Require:
N : nodes in PRDG with a many-to-one function as memory map
$T(n, d)$: Function to query the annotation of the loop (sequential/parallel/ordering)
Ensure: Multiple points in each node in N are not scheduled at same time
N' : list of edges that are violated
D_n : domain of an node

$N' \leftarrow \emptyset$
for all $n \in N, \exists\ z, z' \in D_n \wedge z \neq z' \wedge M(z) = M(z')$ **do**
  for $d = 1$ to max dimension of D_n **do**
    if $T(n, d) = SEQUENTIAL$ **then**
      if $z_d \neq z'_d$ **then**
        there are no write conflict
        $N \leftarrow N - n$
      end if
    else if $T(n, d) = PARALLEL$ **then**
      if $z_d \neq z'_d$ **then**
        there is a write conflict
        $N' \leftarrow N' + n$
      end if
    end if
  end for
end for
return N'

all the dependences, the producer and consumer should not cross processor boundaries, and in our case except for edge 4, all the other dependences satisfy this legality condition. For dependence $4 : S_2 \rightarrow S_3$, it can be observed that from the constraint $j < i$ in the domain of the dependence that producer of $x[j]$ is not from the same iteration of the loop i which is marked as parallel. So edge 4 is added to the violated dependences list, and all the other dependence have to be checked in subsequent dimensions of space time maps. In Dimension 3, edges $2, 6, 7$ satisfy the legality condition as in loop i, the producers in each of these dependences appears before the consumer in textual order. Only remaining to be satisfied is edge 3. This dependence is satisfied in dimension 4 of the space time map, as the producer is the previous iteration of the sequential loop j of $S2$.

4.2 Write Conflicts with Shared Variables

`ompVerify` checks that whenever multiple statement instances write into the same memory location, the execution of these instances cannot happen at a same time (in parallel). Consider the example in Fig. 5, where array ***TM*** is not declared as private. The memory mapping function for S_1 is $((0, k, 0, m, 0, n) \rightarrow TM[1, m])$. So we consider two distinct points in the program that write into the same memory location. In this example this will correspond to points from different iterations of the k loop but with the same m. Because loop k is parallel, multiple iterations of this loop will independently write into TM, which results in data race. `ompVerify` uses Algorithm 2 to perform this check.

4.3 Over-Write Conflicts with Shared Variables

Consider the simple example below:

```
#pragma omp parallel for private(j)
   for (i = 0; i < N; i++)
      for (j = 0; j < i; j++) {
S1:   A[j] = a[i,j];
S2:   B[i,j] = A[j];
   }
```

In this example, all dependences are satisfied within an instance of the loop j , i.e. there are no loop carried dependences, so parallelization of the loop i does not violate any of the dependences. However, since array A is shared by all iterations of loop i, there will be a data race involving A. Specifically, values of $A[j]$ read in S_2 may not be correct. For all the dependences where the producer has a many-to-one memory map, `ompVerify` checks that the time-stamp of consumer is no later than any of the over-writes to those memory locations. If the over-writes happen from multiple iterations of a parallel loop, then the above condition cannot be guaranteed, so `ompVerify` flags this as over-write conflict.

5 Integrating our Analysis

We now describe our prototype tool, its integration into the Eclipse IDE, and provide timing information to illustrate its run-time overhead.

We believe that IDEs will play a significant role in the adoption of multi-core programming. It is highly unlikely for parallel programming to become widely adopted without insightful programming environments. The Eclipse CDT/CODAN framework and the Parallel Tool Platform (PTP) project [17] are very interesting and promising initiatives in this direction.

CDT and CODAN together provides a light-weight static analysis framework, easing integration of our tool. PTP is a set of plug-ins for Eclipse with the goal of providing parallel programming environment. However, its support for OpenMP programs are still very limited. We see PTP as a perfect platform to integrate our analysis.

`ompVerify` builds on CDT, and utilizes two research compilers, GeCoS and AlphaZ, as back-end to perform the analysis. These results are returned to the user through CODAN.

5.1 Implementation of ompVerify

The flow of our prototype implementation[5] is depicted in Fig. 6. GeCoS [18] serves as the front-end that builds polyhedral intermediate representation (IR) from C programs. AlphaZ takes the IR and performs polyhedral analysis to detect errors that are reported to the user through CODAN. Internally, both GeCoS and AlphaZ utilize a number of existing tools for polyhedral analysis to extract ACLs and detect errors.

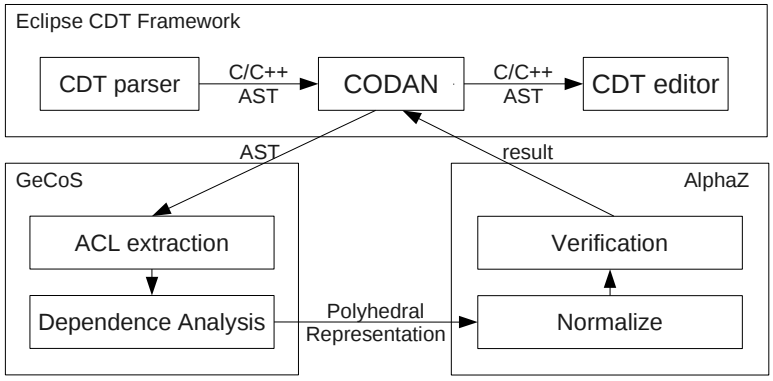

Fig. 6. Flow of `ompVerify`. C programs are parsed by CDT, sent to GeCoS for extracting polyhedral regions, and then analyzed by AlphaZ for its validity.

Time (s)	CDT Front End	ACL Extraction	Dependence Analysis	Normalization	Verification
ProdMat	0.80	0.16	0.59	1.23	1.92
Gauss	0.76	0.31	1.26	0.66	0.77
Examples (Fig 1 to4)	0.89	1.20	1.38	17.34	6.10
SOR 2D	0.78	0.44	1.62	198.25	7.76

Table 2. `ompVerify` overhead

5.2 Evaluation of Overhead

For best user experience, it is important that the analysis remains fast enough so that the user is not inconvenienced. Polyhedral operations are known to be expensive (most of them are NP complete), and thus understanding the overhead of our analysis is important.

Table 2 shows the execution time of various components of our analysis for several examples, including those from Section 3. Our prototype implementation provides nearly instantaneous response for small examples, but gets somewhat slow as the input program becomes complicated. The breakdown shows that the bottleneck is in Normalization and Verification.

Normalization is a pre-processing step that repeatedly updates the IR using local rewrite rules. We have not optimized this, and there are significant savings to be achieved as we go beyond research prototypes. For instance, our IR is currently in a form that needs many passes of tree rewriting. With some effort, it can be incrementally closer to normal form *during* its construction.

The verification also takes some time with large programs, but is also in very early stage of development. The verifier engine that we currently use is designed for a more general need—verifying proposed parallelizations of equational programs, and could be specialized to `ompVerify`.

[5] `ompVerify` is not yet integrated to the PTP static analysis framework.

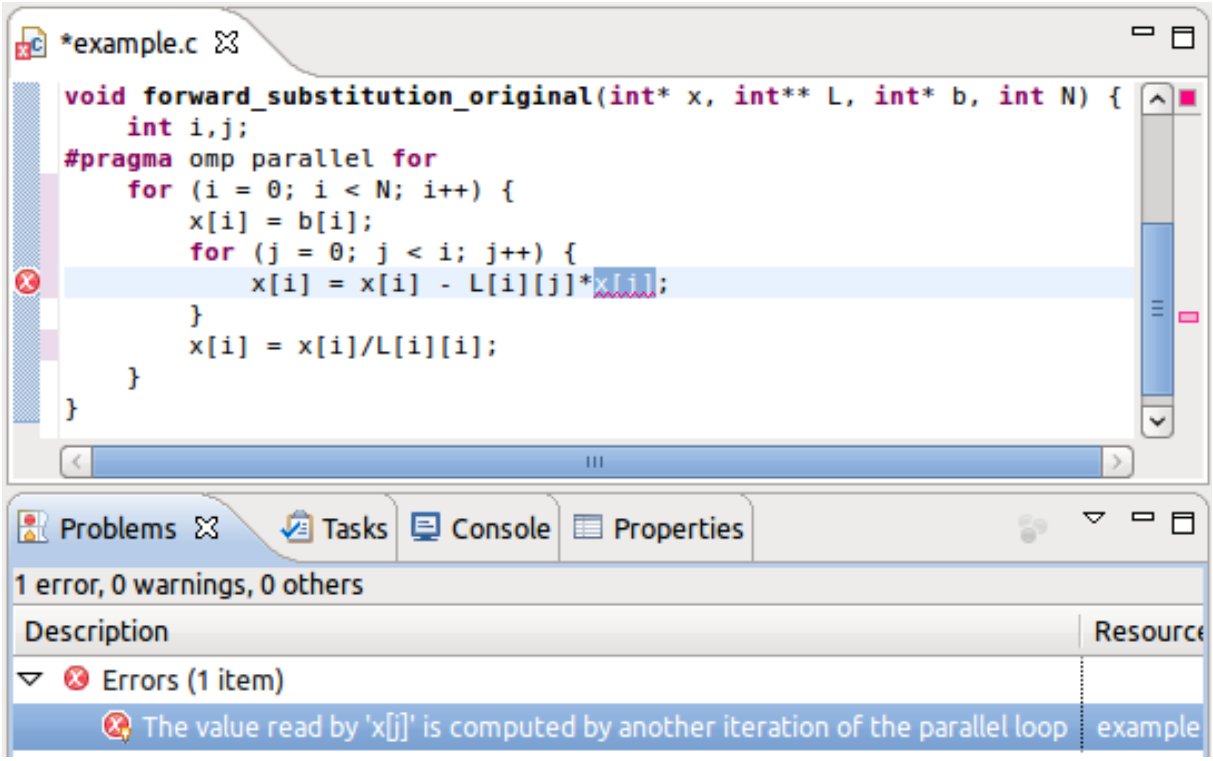

Fig. 7. Screenshot of `ompVerify` detecting incorrect parallelization of forward substitution (Fig. 2.1). We are working on providing more precise information found by our analysis to CODAN.

6 Discussion, Limitations and Extensions

We have presented an analysis that statically detects parallelism violations and data races in OpenMP programs. We believe that statically detecting data races are important and it would greatly help OpenMP programmers, even though our analysis is limited to SCoPs. If the user was willing to see warnings rather than just errors, our analysis could also be easily adapted to approximate information produced by recently proposed extension to the polyhedral model [19, 20] to handle richer set of programs.

In addition to extending the scope of our analysis to larger class of programs, there are a number of simple extensions to the types of OpenMP directives that can be handled. These include the `nowait` option, explicit synchronization directives such as `barrier` and variations of `private` such as `firstprivate` or `lastprivate`. We are actively working on these.

We have explored one bug finding tool based on the polyhedral framework, but others, such as straightforward extensions of many scalar warnings, are also possible. Wonnacott [21] described instance-wise extensions of a number of standard scalar analyses such as dead-code elimination. This work did not demonstrate any value of these optimizations in practice; Wonnacott later surmised that no significant optimization would be useful on existing codes, as dead array elements (for

example) would likely be so costly that the programmer would have avoided them in the original code.

This supposition presumes the programmer has an accurate understanding of the code; the very presence of dead array elements suggests otherwise. Dead array element (or dead loop iteration) analysis could provide a potentially useful analog of the scalar "unused variable" or "potential unused variable" warnings. These can be identified with the full analysis suggested in [21], or (more quickly but perhaps almost as accurately) by simply flagging any statement for which any element of the iteration domain is not the source of any dependence. Analogously, "array element may be used before set" warnings can be produced for any statement in which any element of the iteration domain includes an iteration that is not the sink of any dependence for every variable read there.

We have not implemented these warnings or measured their value, but believe a user interface for it could be analogous to our work described above. We believe such tests would flag as problematic some, but not all, of the erroneous codes of Section 3. For example, Figure 2 should exhibit these warnings, since `A[i][j]` may go unused in some circumstances, and `A[i-1][j]` may be used before it is set.

7 Related Work

There is a long history of research on the polyhedral model [3–7, 10], including work on foundations, scheduling, memory analysis and code generation. The model is now finding its way into production compilers, both commercial [22] and open source [23]. Nevertheless, automatic parallelization is very difficult, and progress is slow. Our work therefore complements these efforts, since explicit, hand parallelization is still the preferred option for most programmers.

Since OpenMP is not a language, there has been relatively little work on analyzing OpenMP programs. Satoh et. al [24] were the first to address this. In the context of developing an OpenMP compiler, they showed how to extend many compiler optimizations and analyses to explicitly parallel programs. They addressed "cross-loop" data dependences but this analysis appears to be limited to sequences of perfectly nested loops. Moreover, they state that, "Parallel loops in OpenMP programs are `doall` type loops, i.e., there are no loop-carried data dependencies without explicit synchronization. Therefore, data dependence analysis within a single parallel loop is not so important." Strictly speaking, this is true —such a program is incorrect, and the compiler is free to do whatever it wants. However, it is equally, if not more, important to report such errors to the user. This is what `ompVerify` seeks to do, and that too, using the most advanced compilations and dependence extraction techniques available.

Lin [25] describes techniques to perform non-concurrency analysis of OpenMP programs, i.e., to detect when statements in a program with OpenMP pragmas *must* be executed serially. The analysis is for "scalar" programs in the sense that even if an instance-wise, element-wise analysis could be provably race-free the analysis may flag a potential race. Huang et. al [26] also present a compiler framework for, again scalar, static analysis of OpenMP. The approach can be used for dead-code and barrier elimination.

Basumallik and Eigenmann [27] describe how OpenMP's memory consistency model can be incorporated into conventional data-flow analysis. This again provides an important bridge between traditional and parallel analyses, and is complementary to our work. Similarly, Liao et. al [28] describe how the Rose system was extended to handle OpenMP programs. Again, this complements our work. Some authors have discussed common mistakes in OpenMP programs [15, 29]. Most of these are either syntactic errors (e.g., missing/miss-spelling directives), or relatively easy to flag

(e.g., shared loop iterators). We focus on errors that are non-trivial to Other errors are detected only after the program has executed, through an analysis of the execution trace [15].

Many tools have been proposed to debug and analyze parallel programs, but mostly targeted to HPC and restricted to distributed memory (MPI).

8 Conclusions

Polyhedral analysis and parallelization methods have an important contribution to parallel computation. In the past,the effort has always been on automatic parallelization. In this paper, we have shown that with a slight change in perspective much of the powerful machinery can be channeled towards (i) static analysis to validate parallelization, (ii) provide debugging/analysis feedback to the programmer, and (iii) even as a pedagogical tool.

We showed that the analysis for automatic parallelization can also be used for static analysis of OpenMP programs, and in pragmatic terms, this may be even more important. Although automatic parallelization is powerful and advancing, it has not yet been adopted by the mainstream programmers, but OpenMP provides methods for incremental parallelization of existing code, and has a much wider user base. It is clear, even from OpenMP compilation efforts that the program directed approach provides a lot of leeway (rope) to the user, and it may result in either very powerful results (rope tricks) or disaster (programmer tripping up). Since such errors are difficult to detect, we believe that it is crucially important to develop tools like ours that verify the correctness of a given OpenMP parallelization.

There are a number of open problems and ways in which our tools can be improved. We have already indicated some of the standard ones: incorporating a wider class of programs by sacrificing precision (warnings rather than errors), simple extensions to the class of programs described here, etc. In the future, we are also planning to extend the analysis to other OpenMP constructs such as barriers, critical sections etc.

References

1. P. Petersen and S. Shah, "OpenMP support in the Intel® thread checker," *OpenMP Shared Memory Parallel Programming*, pp. 1–12, 2003.
2. J. Cownie, S. Moore *et al.*, "Portable OpenMP debugging with totalview," in *Proceedings of the Second European Workshop on OpenMP (EWOMP'2000)*. Citeseer, 2000.
3. S. V. Rajopadhye, S. Purushothaman, and R. M. Fujimoto, "On synthesizing systolic arrays from recurrence equations with linear dependencies," in *Proceedings, Sixth Conference on Foundations of Software Technology and Theoretical Computer Science*. New Delhi, India: Springer Verlag, LNCS 241, December 1986, pp. 488–503, later appeared in Parallel Computing, June 1990.
4. P. Feautrier, "Some efficient solutions to the affine scheduling problem. I. One-dimensional time," *International Journal of Parallel Programming*, vol. 21, no. 5, pp. 313–347, 1992.
5. ——, "Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time," *International Journal of Parallel Programming*, vol. 21, no. 6, pp. 389–420, 1992.
6. ——, "Dataflow analysis of array and scalar references," *International Journal of Parallel Programming*, vol. 20, no. 1, pp. 23–53, 1991.
7. W. Pugh, "The Omega test: a fast and practical integer programming algorithm for dependence analysis," in *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*. ACM, 1991, p. 13.
8. W. Pugh and D. Wonnacott, "Eliminating false data dependences using the Omega test," in *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, ser. PLDI '92. New York, NY, USA: ACM, 1992, pp. 140–151. [Online]. Available: http://doi.acm.org/10.1145/143095.143129

9. ——, “Constraint-based array dependence analysis,” *ACM Trans. Program. Lang. Syst.*, vol. 20, pp. 635–678, May 1998. [Online]. Available: http://doi.acm.org/10.1145/291889.291900
10. F. Quilleré, S. Rajopadhye, and D. Wilde, “Generation of efficient nested loops from polyhedra,” *International Journal of Parallel Programming*, vol. 28, no. 5, pp. 469–498, October 2000.
11. C. Bastoul, “Code generation in the polyhedral model is easier than you think,” in *PACT’13: IEEE International Conference on Parallel Architectures and Compilation and Techniques*, Juan-les-Pins, September 2004, pp. 7–16.
12. Wikipedia, “Frameworks supporting the polyhedral model — wikipedia, the free encyclopedia,” 2011, [Online; accessed 21-March-2011]. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Frameworks supporting the polyhedral model
13. K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, , and K. Yelick, “The landscape of parallel computing research: A view from berkeley,” EECS, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, Dec 2006.
14. S. Verdoolaege, “ISL,” http://freshmeat.net/projects/isl.
15. M. Süß and C. Leopold, “Common mistakes in OpenMP and how to avoid them,” *OpenMP Shared Memory Parallel Programming*, pp. 312–323, 2008.
16. S. A. Amarasinghe, “Paralelizing compiler techniques based on linear inequalities,” Ph.D. dissertation, Stanford University, 1997.
17. “Eclipse parallel tools platform,” http://www.eclipse.org/ptp/.
18. CAIRN, IRISA, “Generic compiler suite,” http://gecos.gforge.inria.fr/.
19. M. W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul, “The polyhedral model is more widely applicable than you think,” in *Compiler Construction*. Springer, 2010, pp. 283–303.
20. W. Pugh and D. Wonnacott, “Nonlinear array dependence analysis,” in *Third Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, Troy, New York, May 1995.
21. D. Wonnacott, “Extending scalar optimizations for arrays,” in *Proceedings of the 13th International Workshop on Languages and Compilers for Parallel Computing-Revised Papers*, ser. LCPC ’00. London, UK: Springer-Verlag, 2001, pp. 97–111. [Online]. Available: http://portal.acm.org/citation.cfm?id=645678.663949
22. B. Meister, A. Leung, N. Vasilache, D. Wohlford, C. Bastoul, and R. Lethin, “Productivity via automatic code generation for PGAS platforms with the R-Stream compiler,” in *APGAS’09 Workshop on Asynchrony in the PGAS Programming Model*, Yorktown Heights, New York, Jun. 2009.
23. S. Pop, A. Cohen, C. Bastoul, S. Girbal, G.-A. Silber, and N. Vasilache, “Graphite: Polyhedral analyses and optimizations for gcc,” in *In Proceedings of the 2006 GCC Developers Summit*, 2006, p. 2006.
24. S. Satoh, K. Kusano, and M. Sato, “Compiler optimization techniques for OpenMP programs,” *Scientific Programming*, vol. 9, no. 203, pp. 131–142, 2001.
25. Y. Lin, “Static nonconcurrency analysis of OpenMP programs,” in *First International Workshop on OpenMP*, M. S. Mueller, B. M. Chapman, B. R. de Supinski, A. D. Malony, and M. Voss, Eds., vol. LNCO 4315. Eugene, OR: Springer-Verlag, June 2005.
26. L. Huang, G. Sethuraman, and B. Chapman, “Parallel data flow analysis for OpenMP programs,” in *IWOMP 2007: International Workshop on OpenMP*, B. Chapman, W. Zhewng, G. Gao, M. Sato, E. Ayguade, and D. Wang, Eds., vol. LNCO 4935. Beijing, China: Springer, June 2007.
27. A. Basumallik and R. Eigenmann, “Incorporation of OpenMP memory consistency into conventional dataflow analysis,” in *Proc. of the International Workshop on OpenMP, IWOMP*. Springer Verlag, 2008.
28. C. Liao, D. Quinlan, T. Panas, and B. de Supinski, “A ROSE-based OpenMP 3.0 research compiler supporting multiple runtime libraries,” in *IWOMP 2010: International Workshop on OpenMP*. Tsukuba, Japan: Springer-Verlag, June 2010.
29. A. Kolosov, E. Ryzhkov, and K. A., “32 OpenMP traps for C++ developers,” 11 2009, http://www.viva64.com/art-3-2-1023467288.html.