Polyhedral Transformations of Explicitly Parallel Programs

Prasanth Chatarasi, Jun Shirako, Vivek Sarkar

Habanero Extreme Scale Software Research Group
Department of Computer Science
Rice University

January 19, 2015

1. Introduction
2. Explicit Parallelism and Motivation
3. Our Approach
4. Preliminary Results
5. Related Work
6. Conclusions, Future work and Acknowledgments

Introduction

- Software with explicit parallelism is on rise
- Two major compiler approaches for program optimizations
  - AST-based
  - Polyhedral-based
- Past work on transformations of parallel programs using AST-based approaches
  - E.g., [Nicolau et.al 2009], [Nandivada et.al 2013]

- **Polyhedral frameworks** for analysis and transformations of **explicitly parallel** programs ??

Introduction

- Explicit parallelism is different from sequential execution
    - Partial order instead of total order
    - No execution order among parallel portions → no dependence

- For the compiler, explicit parallelism can mitigate imprecision that accompanies unanalyzable data accesses from a variety of sources.
    - Unrestricted pointer aliasing
    - Unknown function calls
    - Non-affine constructs
        - Non-affine expressions in array subscripts
        - Indirect array subscripts
        - Non-affine loop bounds
        - Use of Structs

1. Introduction
2. Explicit Parallelism and Motivation
3. Our Approach
4. Preliminary Results
5. Related Work
6. Conclusions, Future work and Acknowledgments

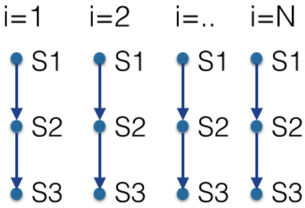
Explicit Parallelism

- Logical parallelism is a specification of a **partial order**, referred to as a **happens-before** relation
    - HB(S1, S2) = *true* ↔ S1 must happen before S2
- Currently, we focus on explicitly parallel programs that satisfy **serial-elision** property
    - Doall parallelism
    - Doacross parallelism

Explicit Parallelism - Doall (OpenMP)

- In case of OpenMP, **Doall** parallelism is equivalent to the **parallel for** clause.
- Happens-before relations exist only among statements in the same iteration
    - Guarantees no cross-iteration dependence

```
#pragma omp parallel for
    for (i-loop) {
            S1;
            S2;
            S3;
    }
```



Explicit Parallelism - Doall (OpenMP) - Example

- LU Decomposition - Rodinia benchmarks [Shuai et.al 09]

```
    for (i = 0; i < size; i++) {
        #pragma omp parallel for
        for (j = i; j < size; j++) {
            #pragma omp parallel for reduction(+:a)
            for (k = 0; k < i; k++) {
                a[i*size+j] -= a[i*size+k] * a[k*size+j];
            }
        }
      ....
    }
```

- `j,k-loops` are annotated as parallel loops and `k-loop` is parallel with a reduction on array a
- Poor spatial locality because of access pattern `k*size+j` for array a
- With happens-before relations from **doall**, loop permutation can be applied to improve spatial locality.

Explicit Parallelism - Doall (OpenMP) - Example

- Permuted kernel

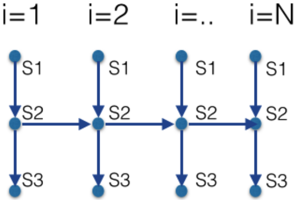
```
        for (i = 0; i < size; i++) {
            #pragma omp parallel for reduction(+:a) private(j)
            for (k = 0; k < i; k++) {
                for (j = i; j < size; j++) {
                    a[i*size+j] -= a[i*size+k] * a[k*size+j];
                }
            }
            ....
         }
```

- 1.25X performance on Intel Xeon Phi coprocessor with 228 threads and input size as 2K
- Array subscripts are non-affine (but can be made affine with delinearization and perform permutation) [Tobias et.al 15]

Explicit Parallelism - Doacross (OpenMP)

- In case of OpenMP, **Doacross** parallelism is equivalent to proposed extension [Shirako et.al 13] to the **ordered** clause (appears in OpenMP 4.1).
- To specify cross-iteration dependences of a parallelized loop

```
#pragma omp parallel for ordered(1)
      for (i-loop) {
        S1;
#pragma omp ordered depend(sink: i-1)
        S2;
#pragma omp ordered depend(source: i)
        S3;
     }
```



Explicit Parallelism - Doacross (OpenMP) - Example

```
// Assume array A is a nested array
#pragma omp parallel for ordered(3)
for (t = 0; t <= _PB_TSTEPS - 1; t++) {
  for (i = 1; i<= _PB_N - 2; i++) {
    for (j = 1; j <= _PB_N - 2; j++) {
#pragma omp ordered  depend(sink: t,i-1,j+1) depend(sink: t,i,j-1) \
     depend(sink: t-1,i+1,j+1)
      A[i][j] = (A[i-1][j-1] + A[i-1][j] + A[i-1][j+1] + A[i][j-1]
               + A[i][j] + A[i][j+1] + A[i+1][j-1] + A[i+1][j]
               + A[i+1][j+1]) / 9.0;
#pragma omp ordered depend(source: t,i,j)
    }
  }
}
```

- 2-dimensional 9 point Gauss Seidel computation - [PolyBench]
- Annotated as 3-D Doacross loop nest
- Even though loop nest has affine accesses, C's unrestricted aliasing semantics for nested arrays can prevent a sound compiler analysis from detecting exact cross iteration dependences.

Explicit Parallelism - Doacross (OpenMP) - Example

```
// Assume array A is a nested array
#pragma omp parallel for ordered(3)
for (t = 0; t <= _PB_TSTEPS - 1; t++) {
  for (i = 1; i<= _PB_N - 2; i++) {
    for (j = 1; j <= _PB_N - 2; j++) {
#pragma omp ordered  depend(sink: t,i-1,j+1) depend(sink: t,i,j-1) \
     depend(sink: t-1,i+1,j+1)
      A[i][j] = (A[i-1][j-1] + A[i-1][j] + A[i-1][j+1] + A[i][j-1]
               + A[i][j] + A[i][j+1] + A[i+1][j-1] + A[i+1][j]
               + A[i+1][j+1]) / 9.0;
#pragma omp ordered depend(source: t,i,j)
    }
  }
}
```

- Through cross-iteration dependences via doacross, loop skewing and tiling can be performed to improve both locality and parallelism granularity.
- 2.2X performance on Intel Xeon Phi coprocessor with 228 threads and input for 100 time steps on a 2K X 2K matrix.

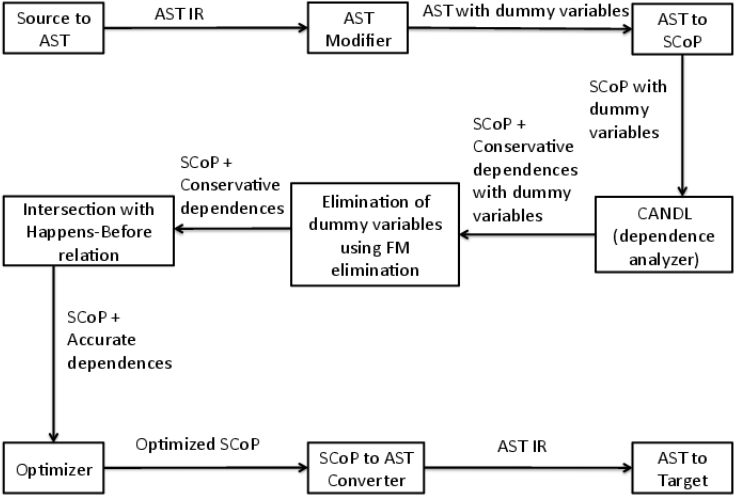
1. Introduction
2. Explicit Parallelism and Motivation
3. Our Approach
4. Preliminary Results
5. Related Work
6. Conclusions, Future work and Acknowledgments

Approach - Idea

- Overestimate dependences based on the sequential order
    - ignore parallel constructs
- Improve dependence accuracy via explicit parallelism
    - obtain happens-before relations from parallel constructs
    - **intersect HB relations with conservative dependences**
- Transformations via polyhedral optimizers
    - PLuTo [Bondhugula et.al 2008]
    - Poly+AST [Shirako et.al 2014]
- Code generation with parallel constructs

- Focus on
    - Doall and Doacross constructs
    - Non-affine subscripts and Indirect arrays subscripts

Algorithm - Framework



Algorithm - Motivation

- Conservative dependence analysis
    - May-information on access range of non-affine array subscripts
- Our existing implementation uses `scoplib` format for convenience (rather than `openscop`)
    - No support for access relations in `scoplib` format (to the best of our knowledge)

- What could potentially represent possible access range of non-affine subscript in polyhedral model?
    - Iterator ?
        - Cannot be part of loops
    - Parameter ?
        - Cannot be loop invariant

Approach - Dummy vector

- Approach use dummy variables to overestimate access range of non-affine subscripts
    - A dummy corresponds to a non-affine expression
    - Compute conservative dependences via dummy variables
- Dummy vector = vector of dummy variables from same scop
- Each dynamic instance of a statement S is uniquely identified by combination of:
    - its iteration vector ($\vec{i_S}$)
    - dummy vector ($\vec{d_S}$)
    - parameter vector ($\vec{p}$)

Approach - Dummy vector - Example

```
int A[N][N], x[N][N], y[N][N];
#pragma omp parallel for
for (i = 0; i < N; i++)
    for (j = 0; j < N; c++)
        A[j][i] = A[ x[j][i] ] [ y[j][i] ];
```

- Non-affine: Two indirect array subscripts (`x[j][i]`, `y[j][i]`)
- Replace non-affine constructs with dummy variables
- Iteration vector ($\vec{i_S}$) = (i, j), Parameter vector ($\vec{p}$) = (N)
- Dummy vector ($\vec{d_S}$) = (dmy_1, dmy_2) = (`x[j][i]`, `y[j][i]`)

Algorithm - Conservative Analysis

- Replace non-affine expressions in array subscripts with `dummy variables` as part of pre-processing
- Create affine inequalities for `dummy variables` based on array declarations and incorporate them into iteration domain
- In case of indirect array subscripts, also associate the index arrays into **read array** list
- Forward the SCoP to CANDL (dependence analyzer)

Algorithm - Conservative Analysis - Example

```
int A[N][N], x[N][N], y[N][N];
#pragma omp parallel for
for (i = 0; i < N; i++)
    for (j = 0; j < N; c++)
        A[j][i] = A[dmy1][dmy2]; // S
```

$$\mathcal{P}_1^{S \to S}(Depth = 1)$$

$$i \leq i' - 1$$

$$j = dmy_1, i = dmy_2$$

$$0 \leq i, j, i', j' \leq (N - 1)$$

$$0 \leq dmy_1, dmy_2 \leq (N - 1)$$

$$0 \leq dmy_1', dmy_2' \leq (N - 1)$$

$$\mathcal{P}_2^{S \to S}(Depth = 2)$$

$$i = i', j \leq j' - 1$$

$$j = dmy_1, i = dmy_2$$

$$0 \leq i, j, i', j' \leq (N - 1)$$

$$0 \leq dmy_1, dmy_2 \leq (N - 1)$$

$$0 \leq dmy_1', dmy_2' \leq (N - 1)$$

Source vector: (i, j, dmy$_1$, dmy$_2$, N)

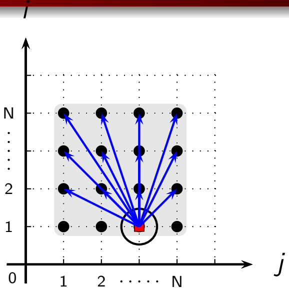
Sink vector: $(i', j', dmy_1', dmy_2', N)$

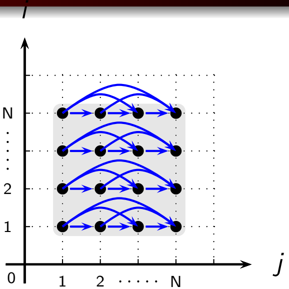
Algorithm - Conservative Analysis - Elimination

After computation of conservative dependences from CANDL, we eliminate dummy variables using Fourier-Motzkin elimination from

- Conservative dependences
- Iteration domain

Algorithm - Conservative Analysis - Elimination - Example





Depth-1 dependences

$$i \leq i' - 1$$

$$\mathcal{P}'^{S\rightarrow S}_{1} : \begin{array}{l} 0 \leq i,j \leq (N-1) \\ 0 \leq i',j' \leq (N-1) \end{array}$$

Depth-2 dependences

$$i = i', j \leq j' - 1$$

$$\mathcal{P}'^{S\rightarrow S}_{2} : \begin{array}{l} 0 \leq i,j \leq (N-1) \\ 0 \leq i',j' \leq (N-1) \end{array}$$

Source vector: (i, j, N)
Sink vector: (i', j', N)

Algorithm - Reflection of happens-before relations

- Let C_d denote happens-before relations on loop at depth $= d$
    - C_d : constraint under which a dependence can exist
- If there are no explicit parallel constructs on a loop, then sequential order would be happens-before relations on that loop
- Happens-before relations in the following program

```
int A[N][N], x[N][N], y[N][N];
#pragma omp parallel for
for (i = 0; i < N; i++)
    for (j = 0; j < N; c++)
        A[j][i] = A[x[j][i]][y[j][i]]; // S
```

$C_1 : \; i = i'$

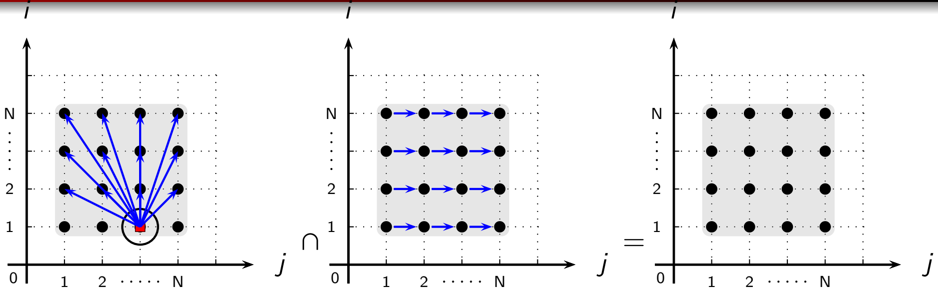
$C_2 : \; i = i', j = j' - 1$

Source vector: (i, j, N) Sink vector: (i', j', N)

Algorithm - Reflection of happens-before relations

```
Input: conservative dependences $\mathcal{P}'$ and constraints $\mathcal{C}$
for each dependence polyhedron ${\mathcal{P}'}_d^{S_i \rightarrow S_j}$ in $\mathcal{P}'$ do
    for each constraint $\mathcal{C}_e^{S_k \rightarrow S_l}$ in $\mathcal{C}$ do
        if $S_i = S_k$ & $S_j = S_l$ & $d = e$ then
            ${\mathcal{P}''}_d^{S_i \rightarrow S_j} := {\mathcal{P}'}_d^{S_i \rightarrow S_j} \cap \mathcal{C}_e^{S_k \rightarrow S_l}$;
        end if
    end for
    Add the reflected polyhedron ${\mathcal{P}''}_d^{S_i \rightarrow S_j}$ to $\mathcal{P}''$;
end for
Output: dependence polyhedra after reflection $\mathcal{P}''$
```

Algorithm - Reflection of happens-before relations - Example - (Depth = 1)



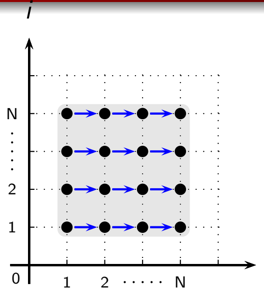
Conservative Dependences
$\mathcal{P}'^{S\rightarrow S}_{1}$:
$i \leq i' - 1$

Happens-Before Relations
$\mathcal{C}'^{S\rightarrow S}_{1}$:
$i = i'$

Final Dependences
$\mathcal{P}''^{S\rightarrow S}_{1}$:
ϕ

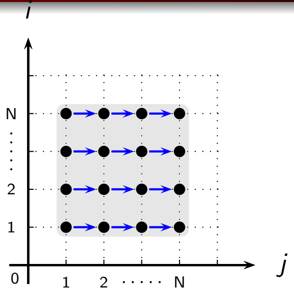
Source vector: (i, j, N) Sink vector: (i', j', N)

Algorithm - Reflection of happens-before relations - Example - (Depth = 2)



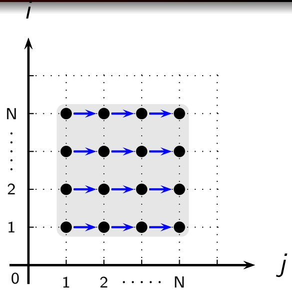
Conservative Dependences $\mathcal{P}'^{S\rightarrow S}_{2}$:
$i = i', j \leq j' - 1$

$\cap$



Happens-Before Relations $\mathcal{C}'^{S\rightarrow S}_{2}$:
$i = i', j = j' - 1$

$=$



Final Dependences $\mathcal{P}''^{S\rightarrow S}_{2}$:
$i = i', j = j' - 1$

Source vector: (i, j, N) Sink vector: (i', j', N)

Algorithm - Code generation

- Transformed kernel after loop permutation

```
int A[N][N], x[N][N], y[N][N];
for (j = 0; j < N; i++)
#pragma omp parallel for
    for (i = 0; i < N; c++)
        A[j][i] = A[x[j][i]][y[j][i]]; // S
```

Algorithm - Implementation

- Implementation is in-progress
- Completed modules:
  - AST Modifier, AST to SCoP converter, Elimination of dummy variables
  - Intersection with Happens-before relations
  - AST to Target
- In-progress modules:
  - Integration with optimizers such as PLuTo, Poly+AST
  - Code generation for do-across

1. Introduction
2. Explicit Parallelism and Motivation
3. Our Approach
4. **Preliminary Results**
5. Related Work
6. Conclusions, Future work and Acknowledgments

Rodinia Benchmarks

- Studied 18 explicitly parallel OpenMP-C Rodinia benchmarks
- Identified non-affine constructs used in the benchmarks that limit the use of polyhedral frameworks
  - Indirect Array Subscript (IAS), Non-affine Array Subscript(NAS), Use of Structs (S), Functions (F)
- Potential opportunities for polyhedral loop transformations that can be enabled through our approach
  - Loop permutation, Fusion, Skewing, Tiling, Doacross parallelism, Vectorization

Rodinia Benchmarks

Limitations

Kernel	NAS	IAS	S	F	Transformations
b+ tree		✓	✓		
backprop				✓	perm, fuse, vect
bfs		✓	✓		
cfd	✓			✓	
heartwall				✓	
hotspot	✓				doacross, fuse, skew, tile, vect
kmean				✓	perm, fuse, vect
lavaMD	✓	✓	✓		
leukocyte				✓	fuse, vect

Table: Limitations and possible transformations in Rodinia benchmarks (NAS: Non-affine Array Subscript, IAS: Indirect Array Subscript, S: Struct, F: Function, and perm/fuse/skew/tile/doacross/vect: loop permutation/fusion/skewing/tiling/doacross parallelism/vectorization)

Rodinia Benchmarks (Continued)

Limitations

Kernel	NAS	IAS	S	F	Transformations
lud	✓				perm, vect
mummergpu			✓	✓	
myocyte	✓			✓	
nn	✓			✓	
nw	✓			✓	doacross, skew, perm
particle filter	✓			✓	fuse, vect
path finder					doacross, skew, tile
srad	✓				
streamcluster	✓	✓	✓	✓	

Table: Limitations and possible transformations in Rodinia benchmarks (NAS: Non-affine Array Subscript, IAS: Indirect Array Subscript, S: Struct, F: Function, and perm/fuse/skew/tile/doacross/vect: loop permutation/fusion/skewing/tiling/doacross parallelism/vectorization)

Preliminary results

$$\text{Speedup} = \frac{\text{Exec time of optimized parallel code}}{\text{Exec time of input parallel code}}$$

Kernel	Benchmark	Best Speedup	Transformation
Backprop	Rodinia	28X	Permutation, Vect
Hotspot	Rodinia	2.25X	Skewing, Tiling, Doacross
Lud	Rodinia	1.15X	Permutation, Vect
Particlefilter	Rodinia	1.05X	Fusion

Table: Performance improvements on Intel Xeon Phi with 228 threads[1]

---

[1] Some steps (e.g., code gen) were done manually.

1. Introduction
2. Explicit Parallelism and Motivation
3. Our Approach
4. Preliminary Results
5. **Related Work**
6. Conclusions, Future work and Acknowledgments

Related Work - Explicitly parallel programs

- Extension of array data-flow analysis to data-parallel and/or task-parallel languages [Collard et.al 96]
- Adaptation of array data-flow analysis to the X10 programs with finish/async parallelism [Yuki et.al 13]
- In these approaches, happens-before relations are first analyzed and data-flow is computed based on the partial order imposed by happens-before relations.

- Our approach first overestimates dependences based on the sequential order and intersect with the happens-before relations from explicitly parallel constructs.
- Our work focuses on transformation of explicitly parallel programs for improved performance where as above works are focused on analysis.

Compile time Approaches for non-affine constructs

- Pugh et.al 91, Maslov et.al 94, ...
- Uses uninterpreted function symbols to represent non-affine constructs
- Generates dependence relations by approximating with affine dependence relations

- We prune conservative dependences using happens-before relations from explicit parallel constructs

Run time Approaches for non-affine constructs

- Doerfert et.al 13, Simburger et.al 14, ...
- Speculative polyhedral optimization techniques, Auto tuning
- Modeling using semi-algebraic sets and real algebra (POLLY)
    - Worst case doubly exponential complexity

- Inspector/ Executor: Strout et.al 03, Basumallik et.al 06, Venkat et.al 14, ...
- Integration into Polyhedral compiler collection chain

- We perform analysis and transformations at compile time

Related Work - PENCIL

- Platform Neutral Compute Intermediate Language
- Automatic parallelization on multi-threaded SIMD hardware for DSL's
- Provides extensions and directives that allow users to supply dependence information

- We are interested in leveraging happens-before relations from programs written in general purpose languages like OpenMP, X10, Habanero-C whereas PENCIL is focused on supporting DSL's in which certain coding rules are enforced related to pointer aliasing, recursion and unstructured control flow.

1 Introduction

2 Explicit Parallelism and Motivation

3 Our Approach

4 Preliminary Results

5 Related Work

6 Conclusions, Future work and Acknowledgments

Conclusions

- Introduced an approach that reflects **happens-before** relations from explicitly parallel constructs in the dependence polyhedra to mitigate conservative dependence analysis.
- Studied 18 explicitly-parallel OpenMP benchmarks from Rodinia suite.
- Shown that the use of explicit parallelism enables larger set of polyhedral transformations, compared to what might have been possible if the input program was sequential.

Future work and Acknowledgments

- Future work
    - Incorporate additional explicit parallel constructs such as barrier and task parallelism
    - Additional transformations for explicit parallel programs

- Acknowledgments
    - Rice Habanero Extreme Scale Software Research Group
    - IMPACT 2015 chairs, reviewers and shepherd

Backup - Conservative Analysis

- Access relations [Wonnacott thesis] and uninterpreted function symbols [Omega library] could have been used instead of dummy variables, but our implementation is heavily dependent on `scoplib` format instead of `openscop` format.
- Our existing implementation uses `scoplib` format for convenience (rather than `openscop`)
  - No support for access relations in `scoplib` format (to the best of our knowledge)

Backup - References

- Nicolau et.al 2009 : Alexandru Nicolau, Guangqiang Li, Alexander V. Veidenbaum, and Arun Kejariwal. 2009. Synchronization optimizations for efficient execution on multi-cores. In Proceedings of the 23rd international conference on Supercomputing (ICS '09). ACM, New York, NY, USA, 169-180. DOI=10.1145/1542275.1542303 http://doi.acm.org/10.1145/1542275.1542303
- Nandivada et.al 2013 : A Transformation Framework for Optimizing Task-Parallel Programs . V. Krishna Nandivada, Jun Shirako, Jisheng Zhao, Vivek Sarkar. ACM Transactions on Programming Languages and Systems (TOPLAS), Volume 35, May 2013.