Polyhedral Optimizations of Explicitly Parallel Programs (PoPP)

Prasanth Chatarasi, Jun Shirako, Vivek Sarkar

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

Computer Science Graduate Seminar Course (COMP 600)

Practice talk: The 24th International Conference on Parallel Architectures and Compilation Techniques (PACT)

October 5, 2015

1. Introduction and Motivation
2. Background
3. Our framework
4. Evaluation
5. Related Work
6. Conclusions and Future work

Introduction - What is it ?

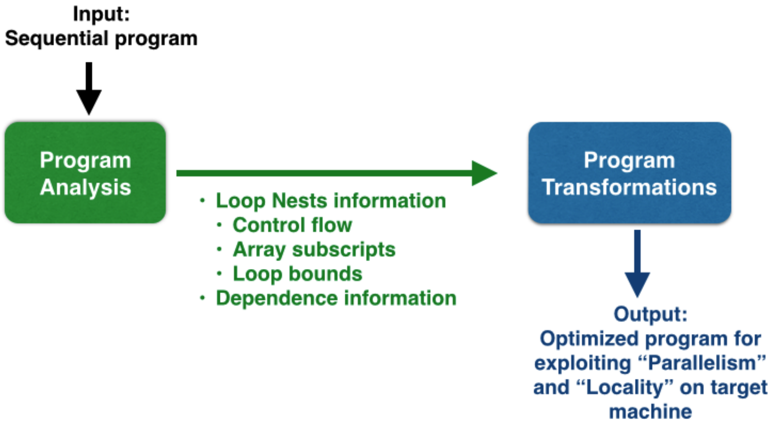
- Move towards Exa-scale computing
  - Billions of billions calculations per second
  - Expected to be processing power of human brain at neural level
- Enabling applications to fully exploit them is not easy !
- Then, How ??

Introduction - What is it ?

- Enabling applications to fully exploit them is not easy !
- Two approaches:
  - Automatically parallelize sequential programs using optimizers (PLuTo, PolyAST etc)
    - Easy ! But, limitations exist.
  - Manually parallelize using explicitly-parallel programming models (Ex: OpenMP, MPI, Habanero, CAF, HPF etc)
    - Tedious ! But, we can achieve high performance

Classical Approach

Automatic parallelization of sequential programs



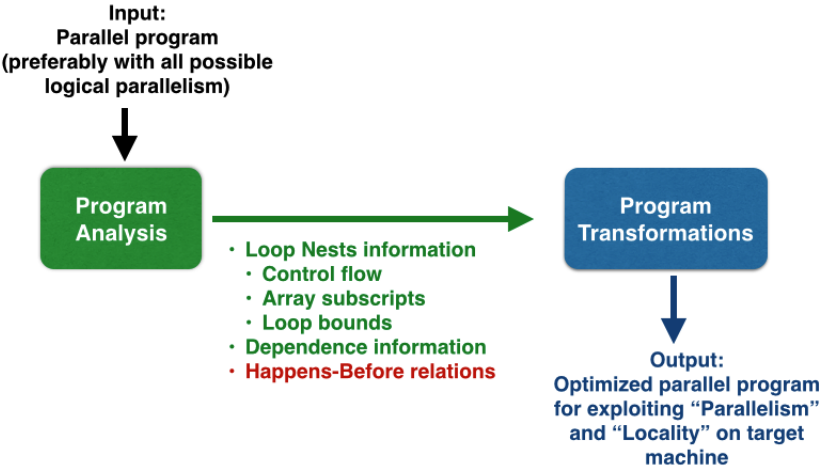
Where is the trend ??

- "Parallelism is oblivious; Let programmer expresses logical parallelism in the application and then let compiler do optimizations accordingly" - Charles Leiserson
  - So far, less attention paid !!

- We introduce an end-to-end compiler framework (PoPP) to optimize parallel applications

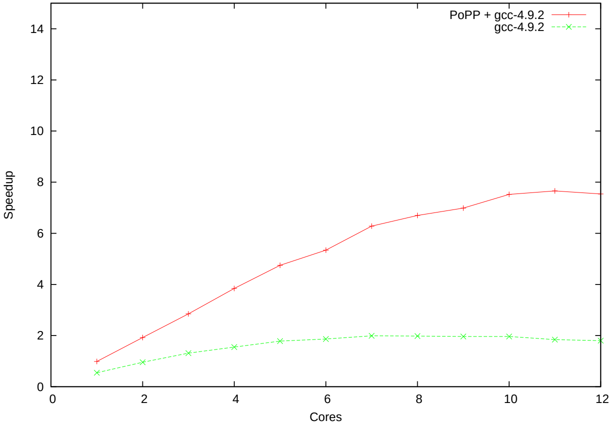
Introduction - What do we do ?

Optimizations of Explicitly-Parallel Programs



Motivation - Why do we need ?

- Jacobi benchmark (4 point 2D stencil) - OpenMP 4.0 Tasks
- Scalability on Intel (Westmere) with 12 cores
- Comparison with PoPP



1 Introduction and Motivation

2 Background

3 Our framework

4 Evaluation

5 Related Work

6 Conclusions and Future work

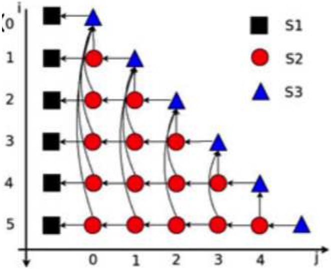
Polyhedral Compilation Techniques

- Compilers techniques for analysis and transformation of codes with nested loops
  - Algebraic framework for affine program optimizations
  - Reason about executions (iterations) of statements
  - Dependence analysis for arrays
  - Encode loop transformations
  - Generate transformed code efficiently

```
for (i = 0; i < N; i++) {
S1: X[i] = B[i];

     for(j = 0; j < i; j++) {
S2:        X[i] -= L[i][j] * X[j];
     }

S3: X[i] /= L[i][j];
}
```

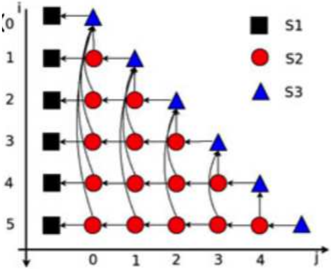


Polyhedral Compilation Techniques - Representation

```
for (i = 0; i < N; i++) {
S1: X[i] = B[i];

      for(j = 0; j < i; j++) {
S2:         X[i] -= L[i][j] * X[j];
      }

S3: X[i] /= L[i][j];
}
```



Iteration domain:

Set of statement instances

S1(i): { i | $0 \leq i \leq N$ }

S2(i,j): { (i,j) | $0 \leq i \leq N$ & $0 \leq j \leq i$ }

S3(i): { i | $0 \leq i \leq N$ }

Schedule:

Logical timestamp

S1(i): (0, i)

S2(i,j): (0, i, 1, j)

S3(i): (0, i, 2)

Access function:

Data accessed

S1(i): { X[i], B[i] }

S2(i,j): { X[i], X[j], L[i][j] }

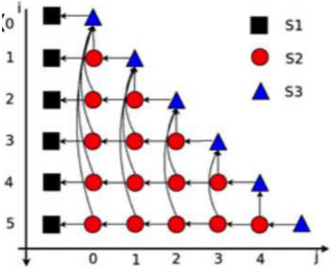
S3(i): { X[i], L[i][j] }

Polyhedral Compilation Techniques - Dependences

```
for (i = 0; i < N; i++) {
S1:  X[i] = B[i];

     for(j = 0; j < i; j++) {
S2:        X[i] -= L[i][j] * X[j];
     }

S3:  X[i] /= L[i][j];
}
```



Dependence b/w S1 & S2

Ex: S1(1) ← S2(1,0)

Dependence b/w S2 & S3

Ex: S2(2,1) ← S3(2)

Dependence b/w S1 & S3

Ex: S1(2) ← S3(2)

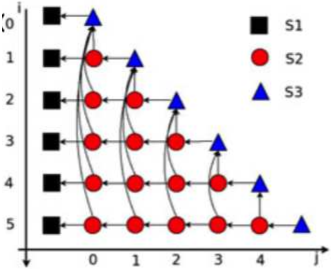
- Polyhedral analyzers capture the above dependences into system of equalities and inequalities.

Polyhedral Compilation Techniques - Summary

```
for (i = 0; i < N; i++) {
S1: X[i] = B[i];

    for(j = 0; j < i; j++) {
S2:       X[i] -= L[i][j] * X[j];
    }

S3: X[i] /= L[i][j];
}
```



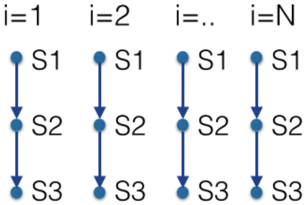
- Advantages
  - Precise data dependency computation
  - Unified formulation of complex set of loop transformations
- Limitations
  - Affine array subscripts, static affine control flow
  - But, conservative approaches exist !

Explicit Parallelism

- Major difference b/w Sequential and Parallel programs
  - Sequential programs - Total execution order
  - Parallel programs - Partial execution order

- Loop-level parallelism
  - Loop is annotated with 'omp parallel for'
  - Iterations of the loop can be run in parallel

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

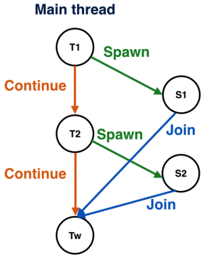


Explicit Parallelism

- Major difference b/w Sequential and Parallel programs
  - Sequential programs - Total execution order
  - Parallel programs - Partial execution order

- Task-level parallelism (OpenMP 3.0)
  - Region of code is annotated with 'omp task'
  - Synchronization is annotated with 'omp taskwait'

```
T1: #pragma omp task
        {S1}
T2: #pragma omp task
        {S2}
Tw: #pragma taskwait
```

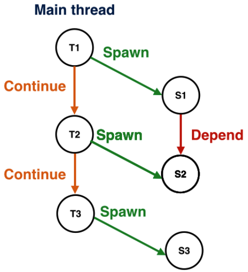


Explicit Parallelism

- Major difference b/w Sequential and Parallel programs
  - Sequential programs - Total execution order
  - Parallel programs - Partial execution order

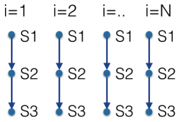
- Task-level parallelism with dependency (OpenMP 4.0)
  - Region of code is annotated with 'omp task depend'
  - Synchronization among **previously created** sibling tasks is achieved by 'depend' clauses

```
T1: #pragma omp task depend↩
    (out: A) {S1}
T2: #pragma omp task depend↩
    (in: A) {S2}
T3: #pragma omp task depend↩
    (out: B) {S3}
```

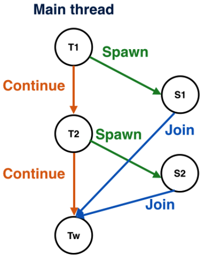


Explicit Parallelism

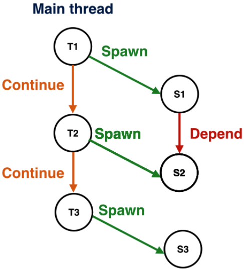
- Happens-Before relations
  - Specification of partial order
  - HB(S1, S2) $=$ *true* $\leftrightarrow$ S1 must happen before S2



HB(S1 (i), S2(i)) = true



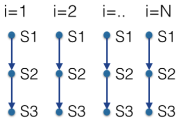
HB(S1, S2) = false



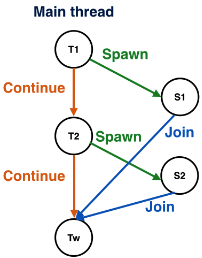
HB(S1, S2) = true

Explicit Parallelism

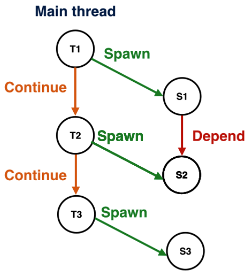
- Serial-Elision property
  - Removal of all parallel constructs results in a sequential program that is a valid (albeit inefficient) implementation of the parallel program semantics.



Satisfies



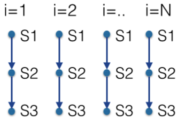
Satisfies



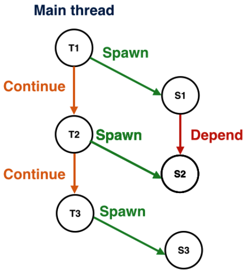
Satisfies

Explicit Parallelism

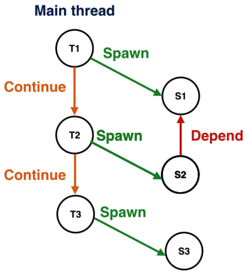
- Serial-Elision property
  - Removal of all parallel constructs results in a sequential program that is a valid (albeit inefficient) implementation of the parallel program semantics.



Satisfies



Satisfies



Not Satisfies (Not possible through depend clause)

1 Introduction and Motivation

2 Background

3 Our framework

4 Evaluation

5 Related Work

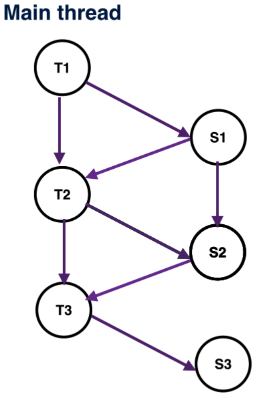
6 Conclusions and Future work

PoPP - Program Analysis

- Step1: Overestimate dependences based on the sequential order (Ignore parallel constructs)

```
T1: #pragma omp task depend(out: A)
    { S1: E[F[..]] =  ..}
T2: #pragma omp task depend(in: A)
    { S2: G[E[..]] =  ..}
T3: #pragma omp task depend(in: A)
    { S3:  B[C[..]] = ..}
```

  - Indirect array subscripts
  - Difficult to capture precise dependences
  - Going conservatively !!

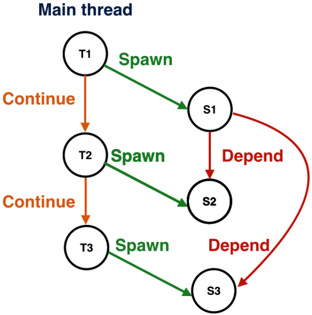


Conservative dependences

PoPP - Program Analysis

- Step1: Overestimate dependences based on the sequential order (Ignore parallel constructs)
- Step2: Compute HB relations

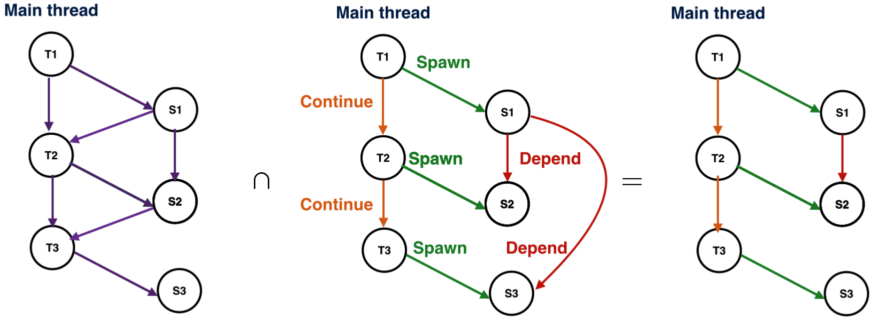
```
T1: #pragma omp task depend(out: A)
    { S1: E[F[..]] =  ..}
T2: #pragma omp task depend(in: A)
    { S2: G[E[..]] =  ..}
T3: #pragma omp task depend(in: A)
    { S3:  B[C[..]] = ..}
```



HB Relations

PoPP - Program Analysis

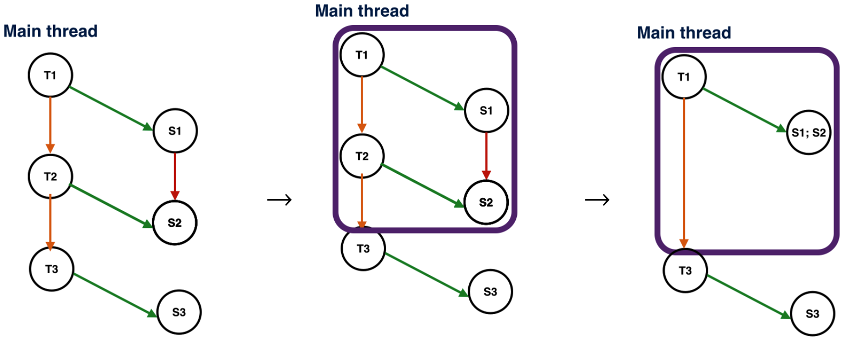
- Step1: Overestimate dependences based on the sequential order (Ignore parallel constructs)
- Step2: Compute HB relations
- Step3: Intersect 1 & 2 (Gives best of both worlds)



Conservative dependences HB Relations Refined dependences

PoPP - Program Transformations

- Step4: Use refined dependences to existing optimizers



Refined dependences

Fusion of S1 & S2 for better data locality

Optimized code

PoPP - Strategy

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, $\mathcal{I}'$

PoPP - Transformations & Code Generation

- Transformations - PolyAST framework [Shirako et.al SC'2014]
  - To perform loop optimizations
  - Hybrid approach of polyhedral and AST-based transformations
  - Detects reduction, doacross and doall parallelism from dependences
- Code Generation
  - Doall parallelism - `omp parallel for`
  - Doacross parallelism - `omp doacross`
    - Proposed in OpenMP 4.1 [Shirako et.al IWOMP'11]
    - Allows fine grained synchronization in multidimensional loop nests

1. Introduction and Motivation
2. Background
3. Our framework
4. Evaluation
5. Related Work
6. Conclusions and Future work

PoPP Evaluation

- Two different SMP platforms

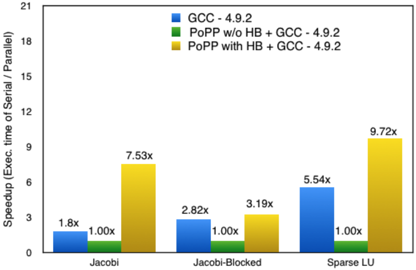
	Intel Xeon 5660 (Westmere)	**IBM Power 8E (Power 8)**
Microarch	Westmere	Power PC
Clock speed	2.80GHz	3.02GHz
Cores/socket	6	12
Total cores	12	24
Compiler	gcc/g++ -4.9.2 icc/icpc -14.0	gcc/g++ -4.9.2
Compiler flags	-O3 -fast(icc)	-O3

Benchmarks

- KASTORS - Task parallel benchmarks (3)
  - Jacobi, Jacobi-blocked, Sparse LU
  - OpenMP 4.0 task, task-depend, taskwait constructs
- RODINIA - Loop parallel benchmarks (8)
  - Back propagation, CFD solver, Hotspot, Kmeans
  - LUD, Needle-Wunch, Particle filter, Path finder
  - OpenMP 3.0 parallel for, Doacross (Modified) constructs
- Unanalyzable data access patterns
  - Non-affine array subscripts
  - Linearized array subscripts
  - Indirect array subscripts
  - Unrestricted pointer aliasing
  - Unknown function calls

PoPP performance on KASTORS Benchmarks - Intel

- Optimization variants
  - Original, PoPP w/o HB, PoPP with HB



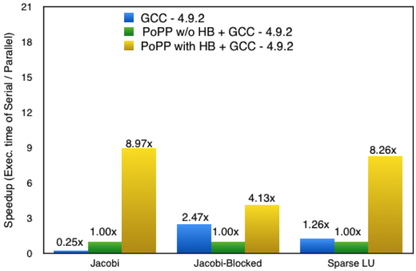
Jacobi: F, S, T, D
Jacobi-blocked: F, S, D
Sparse LU: F, D

Intel Westmere with 12 cores
Optimizations: Fusion(F), Skewing(S), Tiling(T), Doacross sync (D)

- PoPP improved performance from 3.19X to 9.72X

PoPP performance on KASTORS Benchmarks - IBM

- Optimization variants
  - Original, PoPP w/o HB, PoPP with HB



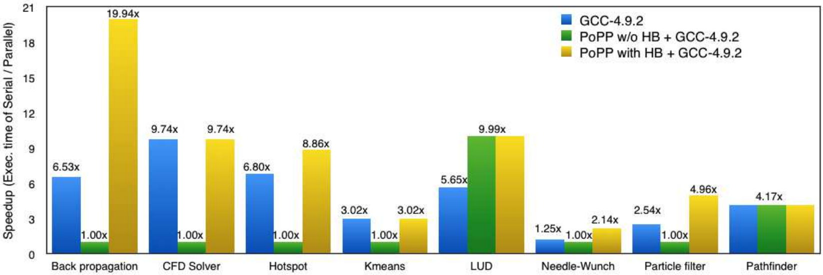
Jacobi: F, S, T, D
Jacobi-blocked: F, S, D
Sparse LU: F, D

IBM Power8 with 24 cores
Optimizations: Fusion(F), Skewing(S), Tiling(T), Doacross sync (D)

- PoPP improved performance from 4.13X to 8.97X

PoPP performance on RODINIA Benchmarks - Intel

- Optimization variants
  - Original, PoPP w/o HB, PoPP with HB

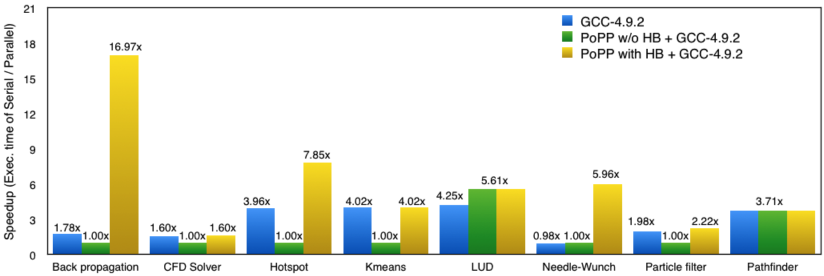


Intel Westmere with 12 cores

- PoPP improved performance from 2.14X to 19.94X

PoPP performance on RODINIA Benchmarks - IBM

- Optimization variants
  - Original, PoPP w/o HB, PoPP with HB



IBM Power8 with 24 cores

- PoPP improved performance from 1.60X to 16.97X

1 Introduction and Motivation

2 Background

3 Our framework

4 Evaluation

5 Related Work

6 Conclusions and Future work

Related work

- Dataflow analysis of explicitly parallel programs
  - Extensions to data-parallel/ task-parallel languages [J.F.Collard et.al Europar'96]
  - Extensions to X10 programs with async-finish languages [T. Yuki et.al PPoPP'13]

  - In these approaches, HB relations are analyzed and data-flow is computed based on partial order imposed by HB relations.
  - We focus on transformations of explicitly parallel programs where as above works only focus in analysis

Related work

- PENCIL - Platform Neutral Compute Intermediate Language [Baghdadi et.al. PACT'15]
  - Automatic parallelization for DSL's
  - Prunes data-dependence relations on parallel loops
  - No support for task parallel constructs
  - Enforces certain coding rules related to aliasing, recursion etc.
- Preliminary approach to optimize parallel programs [Pop and Cohen CPC'10]
  - Extract parallel semantics into compiler IR and perform polyhedral optimizations
  - Envisaged on considering OpenMP streaming extensions

1 Introduction and Motivation

2 Background

3 Our framework

4 Evaluation

5 Related Work

6 Conclusions and Future work

PoPP - Conclusions and Future work

- Conclusions: Our approach
  - Introduced a new analysis for parallel programs
  - Reduced spurious dependences from conservative analysis by intersection with happens-before relations
  - Broadened the range of legal transformations for parallel programs

- Future work:
  - Parallel constructs that don't satisfy serial-elision property
  - Code generation with *task* constructs

- Acknowledgments
  - Rice Habanero Extreme Scale Software Research Group
  - PACT 2015 program committee