Polyhedral Optimizations of Explicitly Parallel Programs

Prasanth Chatarasi, Jun Shirako, Vivek Sarkar

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

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

October 19, 2015

Introduction

- Moving towards Extreme-Scale and Exa-Scale computing systems
  - Billions of billions operations per second
- Enabling applications to fully exploit the systems is **not easy** !
- How ??

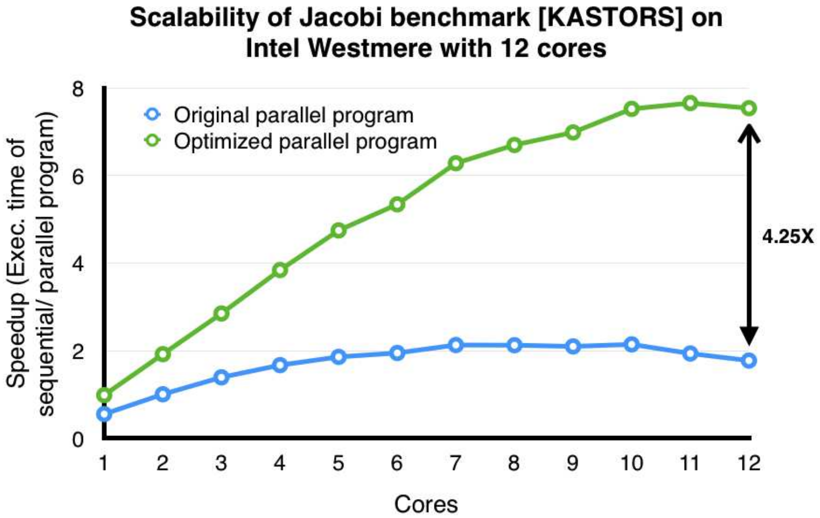
Introduction

- Two approaches from past work:
    - 1) Manually parallelize using explicitly-parallel programming models (E.g., CAF, Cilk, Habanero, MPI, OpenMP, UPC etc)
        - Optimizations performed by programmer not compiler !
        - Tedious ! But can deliver good performance, with sufficient effort
    - 2) Automatically parallelize sequential programs
        - Done by compilers not humans !
        - Easy ! But, limitations exist.

Motivation and Our Approach

- **Motivation**
  - Programmer expresses logical parallelism in the application and then let compiler perform optimizations accordingly
- **Our approach**
  - *Automatically optimize explicitly-parallel programs*

Glimpse of benefits

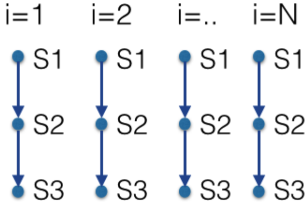


Explicit Parallelism - Loop level parallelism

- Major difference between Sequential and Parallel programs
  - Sequential programs - total execution order
  - Parallel programs - partial execution order

- Loop-level parallelism (since OpenMP 1.0)
  - Loop is annotated with 'omp parallel for'
  - Iterations of the loop can be run in parallel

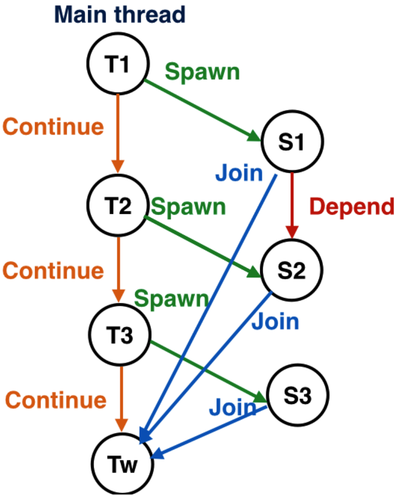
```
#pragma omp parallel for
    for (i−loop) {
            S1;
            S2;
            S3;
    }
```



Explicit Parallelism - Task level parallelism

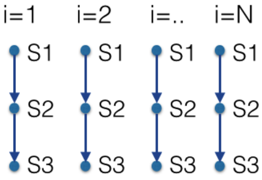
- Task-level parallelism (OpenMP 3.0 & 4.0)
  - Region of code is annotated with 'omp task'
  - Synchronization
    - B/w parent and children - 'omp taskwait'
    - B/w siblings - 'depend' clause

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

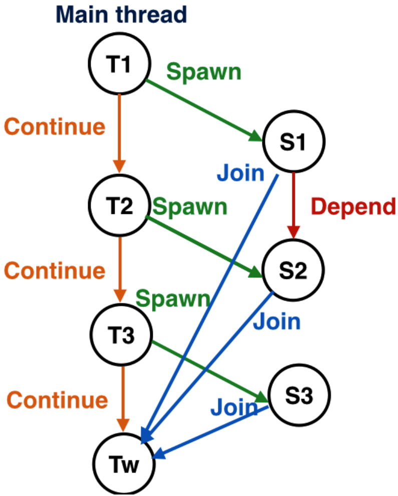


Explicit Parallelism - Happens before relation

- Happens-Before relation
  - Specification of partial order among dynamic statement instances
  - HB(S1, S2) = *true* ↔ S1 must happen before S2, where S1 and S2 are statement instances.



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



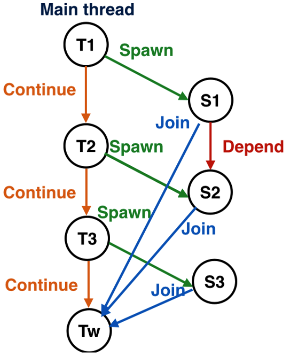
HB(S1, S2) = true, HB(S2, S3) = false

Explicit Parallelism - Serial elision property

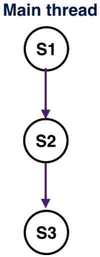
- Serial-Elision property
  - Removal of all parallel constructs results in a sequential program that is a valid (albeit inefficient) implementation of the parallel program semantics.

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

Satisfies serial-elision



Original task graph



Removing parallel constructs

Polyhedral Compilation Techniques

- Compiler techniques for analysis and transformation of codes with nested loops
- Algebraic framework for affine program optimizations
- Advantages over AST based frameworks
    - Reasoning at statement instance level
    - Unifies many complex loop transformations

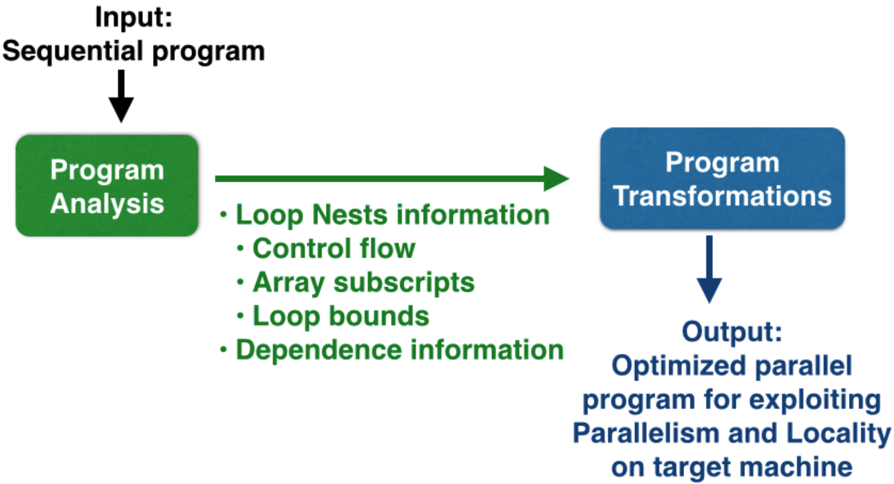
Polyhedral Representation (SCoP)

- A statment (S) in the program is represented as follows in Static Control Part (SCoP):
- 1) Iteration domain ($\mathcal{D}^{S}$)
  - Set of statement (S) instances
- 2) Schedule (Θ^{S})
  - Assigns logical time stamp to the statement instances (S)
  - Gives ordering information b/w statement instances
  - **Captures sequential execution order of a program**
  - Statement instances are executed in increasing order of schedules
- 3) Access function ($\mathcal{A}^{S}$)
  - Array subscripts in the statement (S)

Polyhedral Compilation Techniques - Summary

- Advantages
  - Precise data dependency computation
  - Unified formulation of complex set of loop transformations
- Limitations
  - Affine array subscripts
    - But, conservative approaches exist !
  - Static affine control flow
    - Control dependences are modeled in same way as data dependences.
  - **Assumes input is sequential program**
    - **Unaware of happens-before relation in input parallel program**

Automatic parallelization of sequential programs



Input:
Sequential program

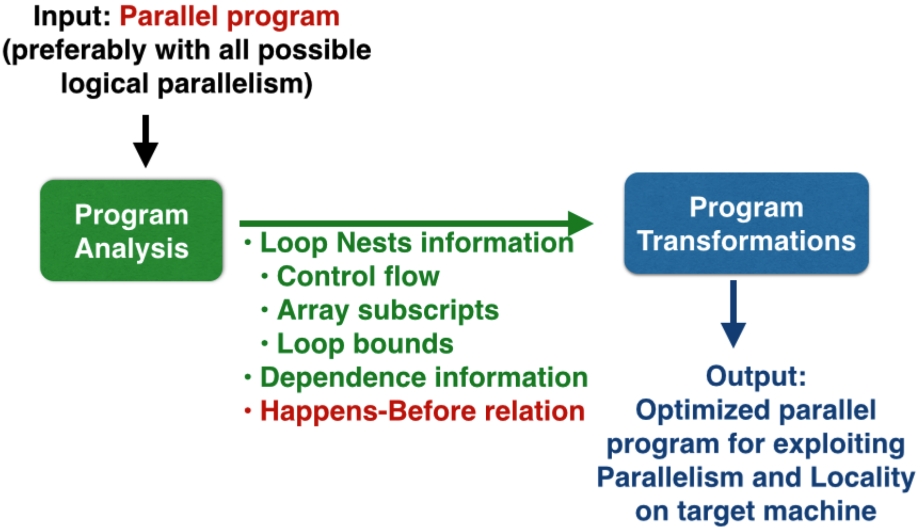
Program Analysis

- **Loop Nests information**
  - **Control flow**
  - **Array subscripts**
  - **Loop bounds**
- **Dependence information**

Program Transformations

Output:
Optimized parallel program for exploiting Parallelism and Locality on target machine

Polyhedral optimizations of Parallel Programs (PoPP)



Input: Parallel program (preferably with all possible logical parallelism)

Program Analysis

- **Loop Nests information**
  - **Control flow**
  - **Array subscripts**
  - **Loop bounds**
- **Dependence information**
- **Happens-Before relation**

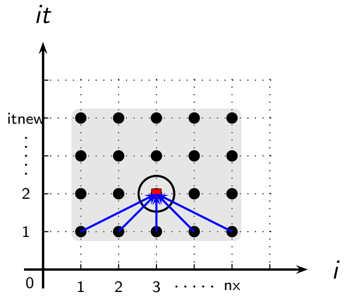
Program Transformations

Output: Optimized parallel program for exploiting Parallelism and Locality on target machine

PoPP - Program Analysis

- Step1: Compute dependences based on the sequential order (use serial elision and ignore parallel constructs)

```
#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
}
```



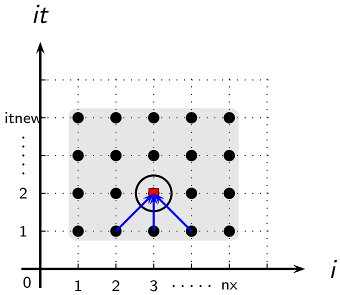
(S2 → S1) dependences across it & i loops

Conservative analysis, but may still capture vectorization possibility

PoPP - Program Analysis

- Step1: Compute dependences based on the sequential order (use serial elision and ignore parallel constructs)
- Step2: Compute happens-before relation

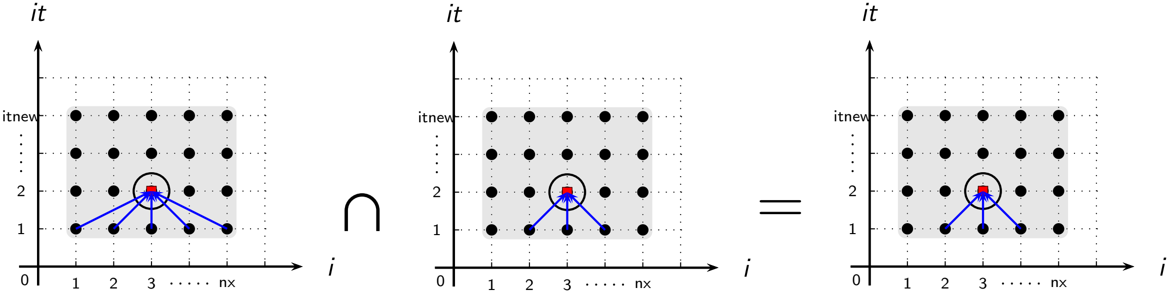
```
#pragma omp parallel
#pragma omp single
{
    for (int it = itold + 1; it <= itnew; it++) {
        for (int i = 0; i < nx; i++) {
#pragma omp task depend(out: u[i]) \
    depend(in: unew[i])              // T1
            for (int j = 0; j < ny; j++)
S1:             u[i][j] = unew[i][j];
        }
        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
}
```



(S2→S1) HB edges across it & i loops

PoPP - Program Analysis

- Step1: Compute dependences
- Step2: Compute Happens-before relation
- Step3: Intersect 1 & 2 (Gives best of both worlds)



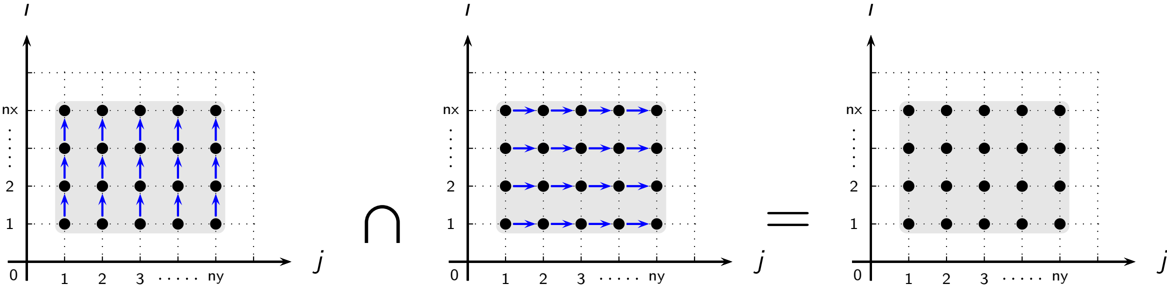
Conservative dependences $\mathcal{P}_1^{S2 \rightarrow S1}$

HB relation $\mathcal{HB}_1^{S2 \rightarrow S1}$

Refined dependences $\mathcal{P'}_1^{S2 \rightarrow S1}$

PoPP - Program Analysis

- Step1: Compute dependences
- Step2: Compute Happens-before relation
- Step3: Intersect 1 & 2 (Gives best of both worlds)



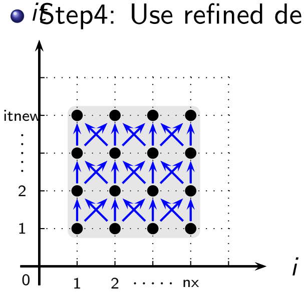
Conservative dependences $\mathcal{P}_1^{S1 \rightarrow S1}$ (j-loop is parallel for S1)

HB relation $\mathcal{HB}_1^{S1 \rightarrow S1}$ (i-loop is parallel for S1)

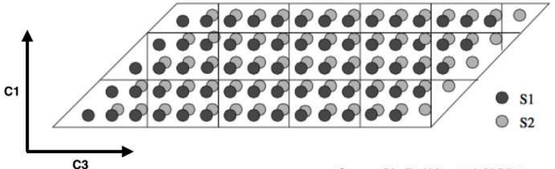
Refined dependences $\mathcal{P'}_1^{S1 \rightarrow S1}$ (No dependences for S1)

PoPP - Program Transformations

- Step4: Use refined dependences in existing optimizations



→



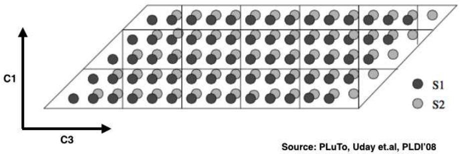
Source: PLuTo, Uday et.al, PLDI'08

Refined dependences, $\mathcal{P}'^{S2\rightarrow S1}_{1}$

- Skewing and tiling the iteration space

PoPP - Code generation

- Step5: Generate optimized code using fine grained synchronization



Source: PLuTo, Uday et.al, PLDI'08

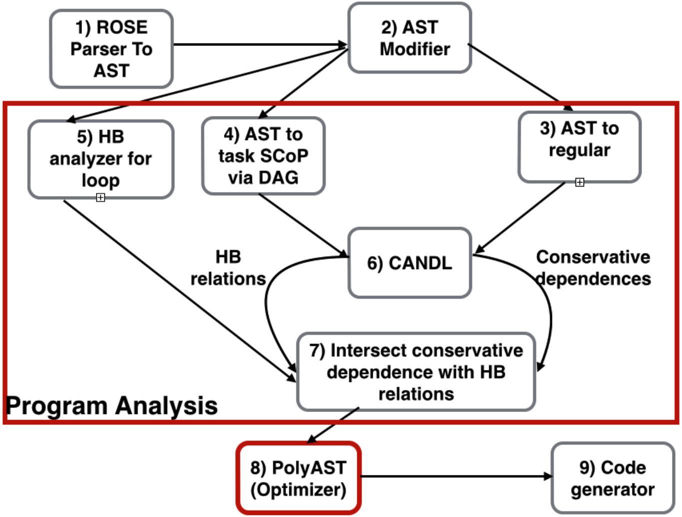
→

```
#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)
 }}
```

- Doacross loop synchronization - OpenMP 4.1

PoPP - Workflow (in ROSE Compiler)



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

Extensions to Polyhedral frameworks

- Correctness of Intersection approach
  - Serial-elision property makes it correct !
- Computing conservative dependences
  - Non-affine subscripts, Unknown function calls, Non-affine conditionals etc
  - Extended access functions to support
- Extracting and Encoding task-related constructs in polyhedral representation (SCoP)
  - Constructed task SCoP to compute HB relation

Evaluation: Benchmarks and 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 -4.9.2	gcc -4.9.2
Compiler flags	-O3 -fast(icc)	-O3

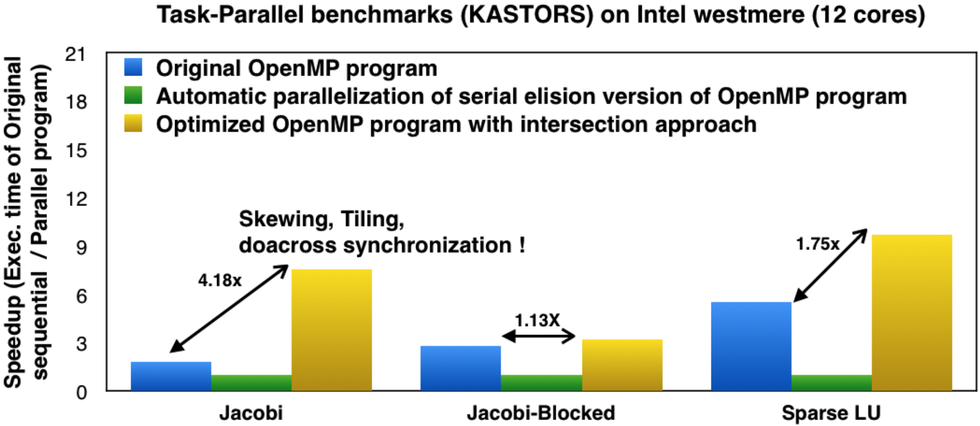
- KASTORS -Task parallel (3)
    - Jacobi, Jacobi-blocked, Sparse LU
- RODINIA -Loop parallel (8)
    - Back propagation, CFD solver, Hotspot, Kmeans, LUD, Needle-Wunch, Particle filter, Path finder
- Unanalyzable data access patterns - 7 benchmarks

Variants

Variants in the experiments

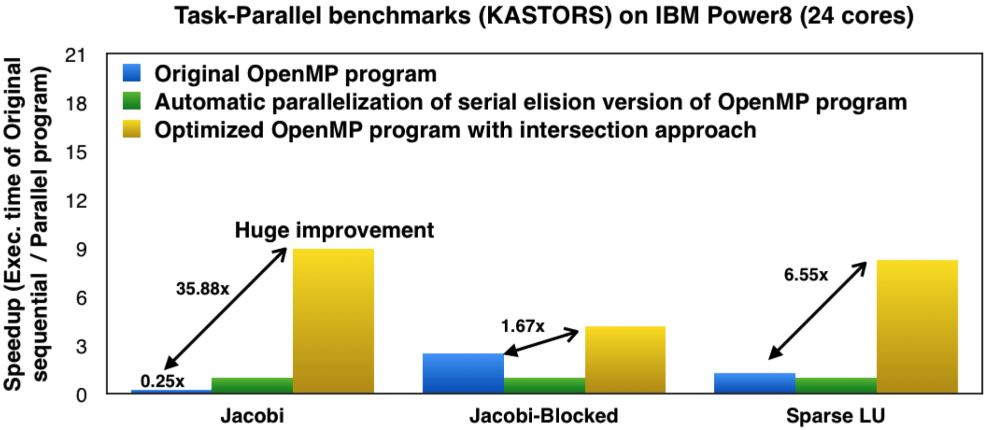
- **Original OpenMP program (Blue bars)**
  - Written by programmer
- **Automatic parallelization and optimization of serial elision version of OpenMP program (Green bars)**
  - Automatic optimizers
- **Optimized OpenMP program with intersection approach (Yellow bars)**
  - Our framework (PoPP)

KASTORS suite + Intel Westmere (12 cores)



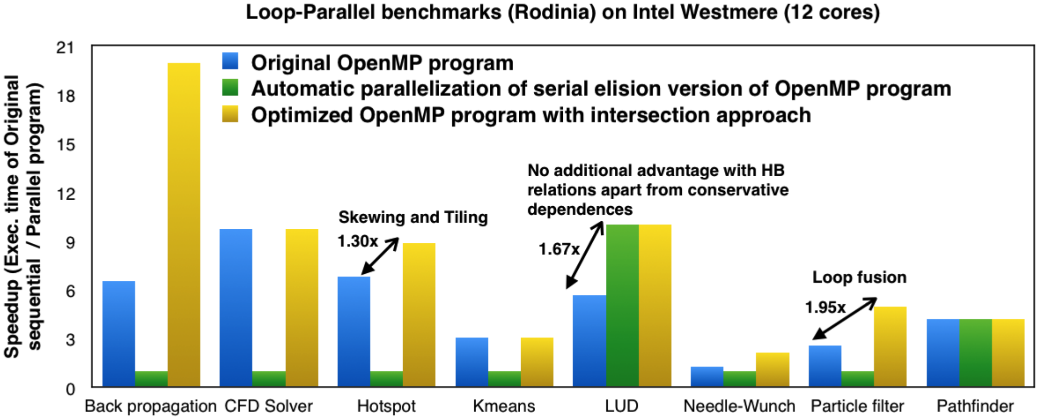
Geometric mean improvement - 2.02x

KASTORS suite + IBM Power8 (24 cores)



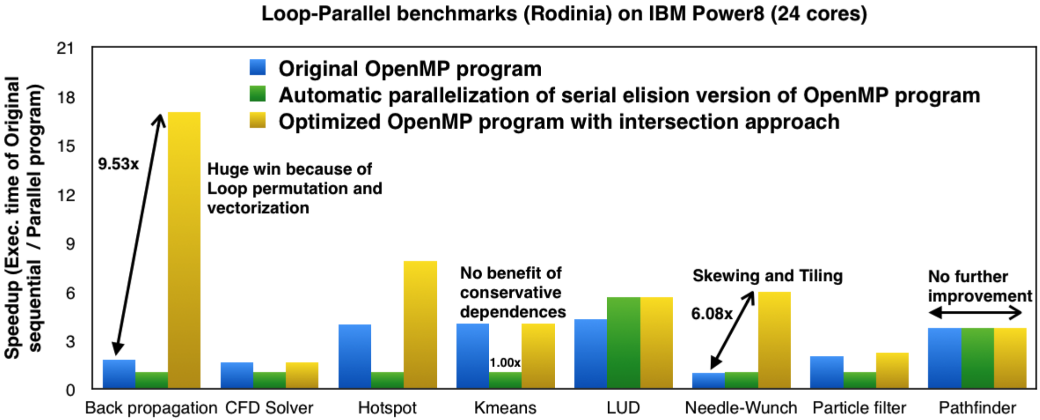
Geometric mean improvement - 7.32x

RODINIA suite + Intel westmere (12 cores)



Geometric mean improvement - 1.48x

RODINIA suite + IBM Power8 (24 cores)



Geometric mean improvement - 1.89x

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]
  - Above work is limited to analysis but we also focus on transformations.
- PENCIL - Platform Neutral Compute Intermediate Language [Baghdadi et.al. PACT'15]
  - Prunes data-dependence relation on parallel loops
  - No support for task parallel constructs as yet
  - Enforces certain coding restrictions related to aliasing, recursion etc.

Related work (contd)

- Polyhedral optimization framework for DFGL [Sbirlea et.al LCPC'15]
    - Dataflow programming model - Implicitly parallel
    - Optimizations via polyhedral & AST-based framework
- 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

PoPP - Conclusions and Future work

- Conclusions: Our approach
  - Reduced spurious dependences from conservative analysis by intersecting with HB relation
  - Broadened the range of legal transformations for parallel programs
  - Integrated HB relation from task-parallel constructs into Polyhedral frameworks
  - Geometric mean performance improvement of 1.62X on Intel westmere and 2.75X on IBM Power8 - Larger improvements !!

- Future work:
  - Parallel constructs that don't satisfy serial-elision property
  - Extend to distributed-memory programming models (Eg: MPI)
  - Happens-Before relation for debugging
  - Beyond polyhedral

Finally,

- *Optimizing explicitly parallel programs is a new direction for Parallel Architectures and Compilation Techniques (PACT)!*

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

- Thank you!