Compiler-Assisted Dynamic Scheduling for Effective Parallelization of Loop Nests on Multi-core Processors

Muthu Baskaran[1] Naga Vydyanathan[1]
Uday Bondhugula[1] J. Ramanujam[2]
Atanas Rountev[1] P. Sadayappan[1]

[1]The Ohio State University

[2]Louisiana State University

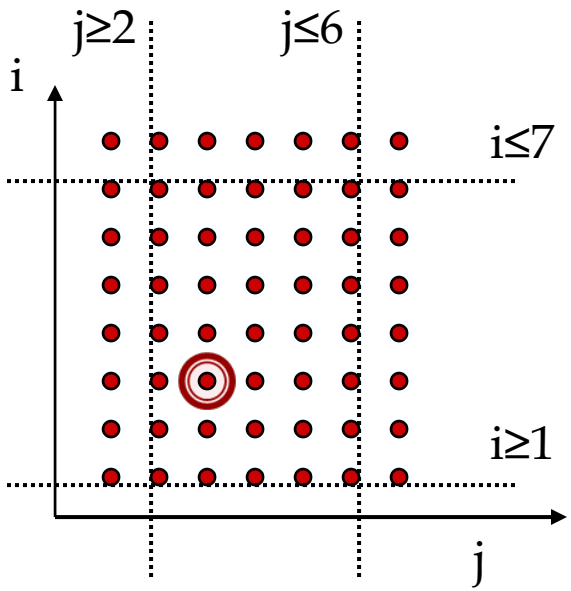
Introduction

- Ubiquity of multi-core processors
- Need to utilize parallel computing power efficiently
- Automatic parallelization of regular scientific programs on multi-core systems
  - Polyhedral Compiler Frameworks
- Support from compile-time and run-time systems required for parallel application development

Polyhedral Model

```
for (i=1; i<=7; i++)
  for (j=2; j<=6; j++)
      S1: a[i][j] = a[j][i] + a[i][j-1];
```

$$F_{3a}(\overrightarrow{x_{S1}}) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ -1 \end{pmatrix}$$

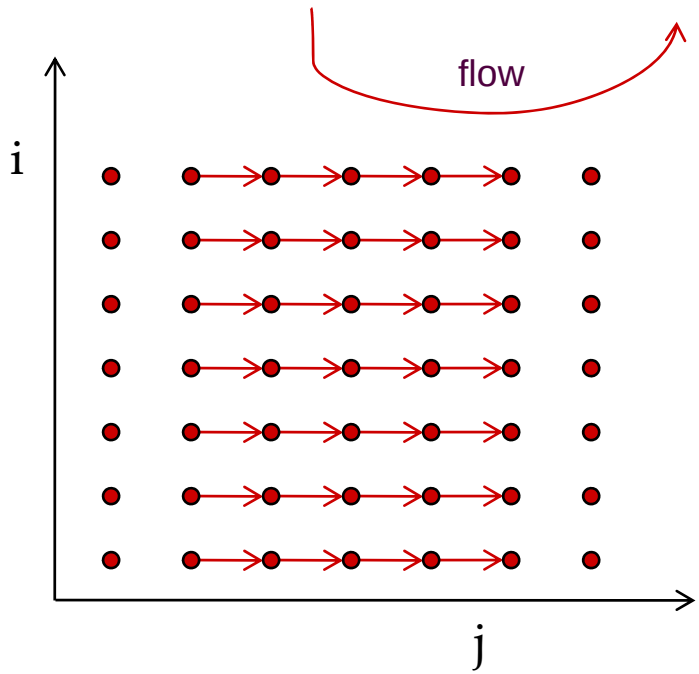


$$\overrightarrow{x_{S1}} = \begin{pmatrix} i \\ j \end{pmatrix}$$

$$\mathcal{D}_{S1}(\overrightarrow{x_{S1}}) = \begin{pmatrix} 1 & 0 & -1 \\ -1 & 0 & 7 \\ 0 & 1 & -2 \\ 0 & -1 & 6 \end{pmatrix} \cdot \begin{pmatrix} i \\ j \\ 1 \end{pmatrix} \geq \vec{0}$$

Dependence Abstraction

```
for (i=1; i<=7; i++)
  for (j=2; j<=6; j++)
      S1: a[i][j] = a[j][i] + a[i][j-1];
```



Dependence Polytope

An instance of statement t (i_t) depends on an instance of statement s (i_s)

- i_s is a valid point in $\mathcal{D}_s$
- i_t is a valid point in $\mathcal{D}_t$
- i_s executed before i_t
- Access same memory location
- h-transform : relates target instance to source instance involved in last conflicting access

$$\begin{pmatrix} D_s & 0 \\ 0 & D_t \\ \hline -I & H \end{pmatrix} \cdot \begin{pmatrix} \vec{i}_s \\ \vec{i}_t \\ \vec{n} \\ 1 \end{pmatrix} \begin{pmatrix} \geq \vec{0} \\ \hline = \vec{0} \end{pmatrix}$$

Compiler-Assisted Dynamic Scheduling for Effective Parallelization of Loop Nests on Multicore Processors, PPoPP 2009

Affine Transformations

- Loop transformations defined using affine mapping functions
  - Original iteration space => Transformed iteration space
  - A one-dimensional affine transform (Φ) for a statement S is given by

$$\Phi_S(\overrightarrow{x_S}) = C_S \cdot \begin{pmatrix} \overrightarrow{x_S} \\ n \\ 1 \end{pmatrix}$$

  - Φ represents a new loop in the transformed space
  - Set of linearly independent affine transforms
    - Define the transformed space
    - Define tiling hyperplanes for tiled code generation

Tiling

```
for (i=0; i<N; i++)
  x[i]=0;
  for (j=0; j<N; j++)
    S: x[i] += a[j][i] * y[j];
```

$$D_s \cdot \begin{pmatrix} i \\ j \\ N \\ 1 \end{pmatrix} \geq \vec{0}$$

```
for ( it =0; it<=floor(N-1,32);it++)
  for ( jt =0; jt<=floord(N-1,32);jt++)
  …
    for ( i=max(32it ,0);
          i<=min(32it+31,N-1);  i++)
      for ( j=max(32jt ,1);
            j<=min(32jt+31,N-1);j++)
            S: x[i] += a[j][i] * y[j];
```

$$DT_s \cdot \begin{pmatrix} it \\ jt \\ i \\ j \\ N \\ 1 \end{pmatrix} \geq \vec{0}$$

$$\begin{pmatrix} T_s & 0 \\ 0 & D_s \end{pmatrix} \cdot \begin{pmatrix} \vec{iT_s} \\ \vec{i_s} \\ \vec{n} \\ 1 \end{pmatrix} \geq \vec{0}$$

- Tiled iteration space
  - Higher-dimensional polytope
  - Supernode iterators
  - Intra-tile iterators

PLUTO

- State-of-the-art polyhedral model based automatic parallelization system
- First approach to explicitly model tiling in a polyhedral transformation framework
- Finds a set of "good" affine transforms or tiling hyperplanes to address two key issues
  - effective extraction of coarse-grained parallelism
  - data locality optimization
- Handles sequences of imperfectly nested loops
- Uses state-of-the-art code generator CLooG
  - Takes original statement domains and affine transforms to generate transformed code

Affine Compiler Frameworks

- Pros
  - Powerful algebraic framework for abstracting dependences and transformations
  - Enables the feasibility of automatic parallelization
    - Eases the burden of programmers
- Cons
  - Generated parallel code may have excessive barrier synchronization due to affine schedules
    - Loss of efficiency on multi-core systems due to load imbalance and poor scalability!

Compiler-Assisted Dynamic Scheduling for Effective Parallelization of Loop Nests on Multicore Processors, PPoPP 2009

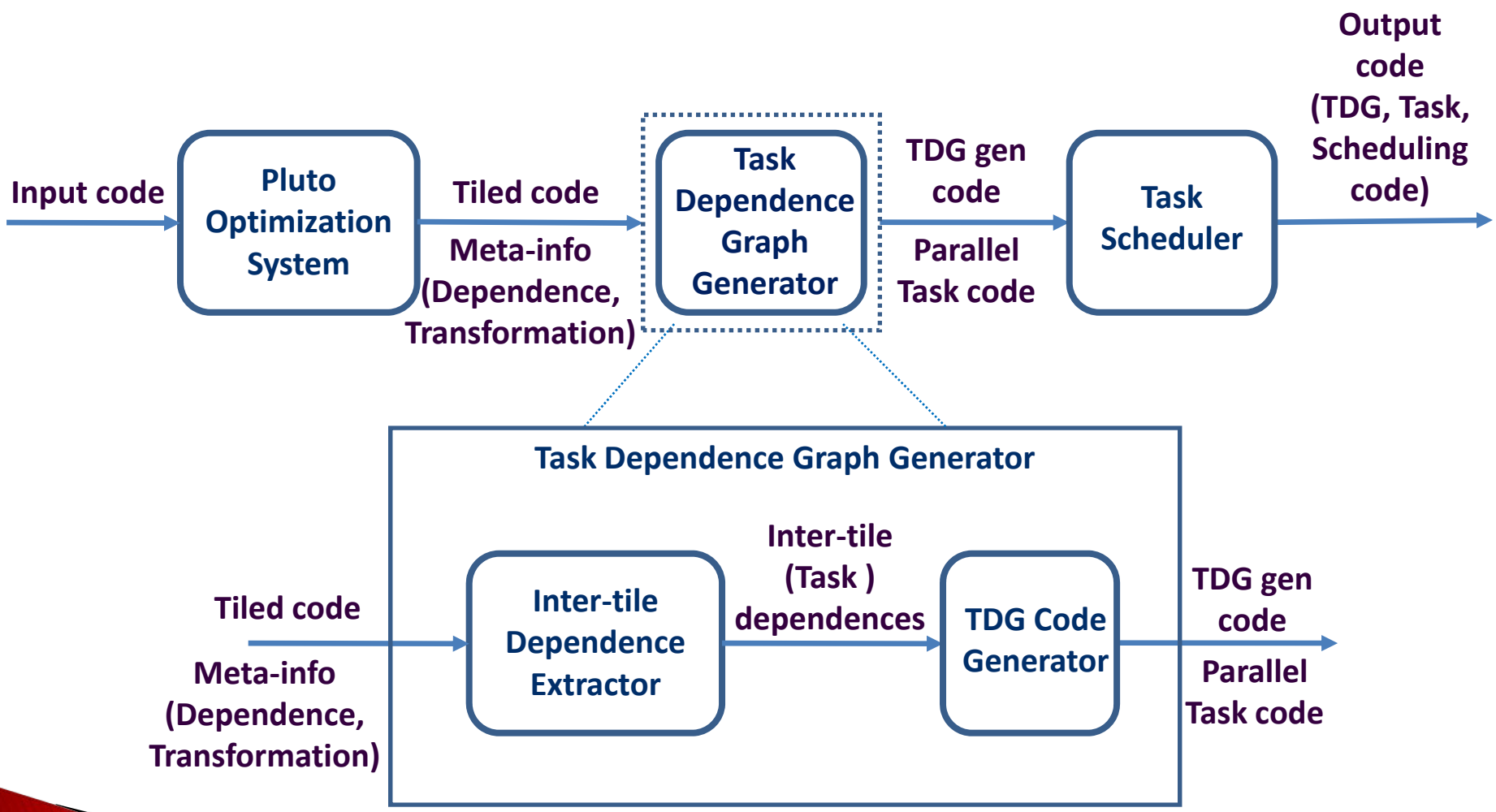
Aim and Approach

- Can we develop an automatic parallelization approach for asynchronous, load-balanced parallel execution?
- Utilize the powerful polyhedral model
  - To generate tiling hyperplanes (generate tiled code)
  - To derive inter-tile dependences
- Effectively schedule the tiles for parallel execution on the processor cores of a multi-core system
  - Dynamic (run-time) scheduling

Aim and Approach

- Each tile identified by affine framework is a "task" that is scheduled for execution
- Compile-time generation of following code segments
  - Code executed within a tile or task
  - Code to extract inter-tile (inter-task) dependences in the form of *task dependence graph* (TDG)
  - Code to dynamically schedule the tasks using critical path analysis on TDG to prioritize tasks
- Run-time execution of the compile-time generated code for efficient asynchronous parallelism

System Overview



Compiler-Assisted Dynamic Scheduling for Effective Parallelization of Loop Nests on Multicore Processors, PPoPP 2009

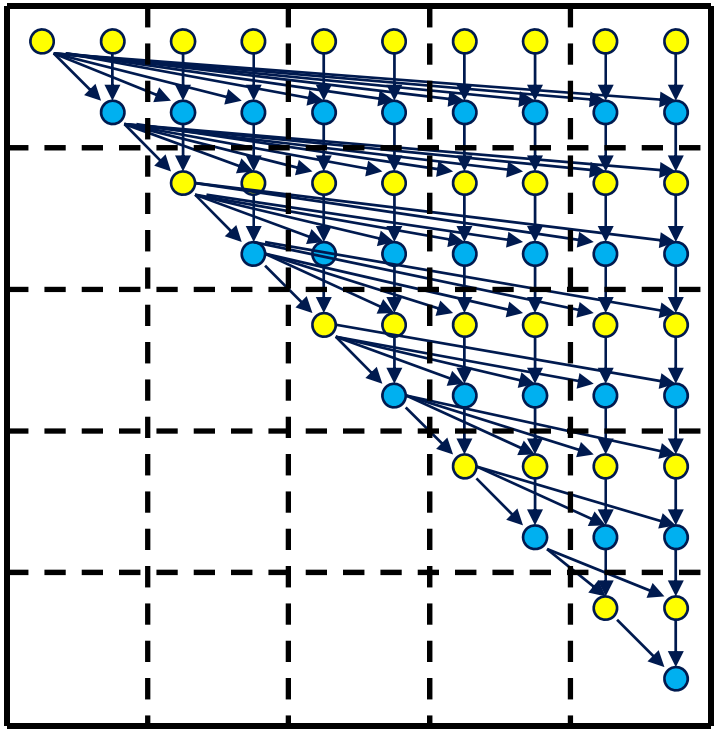
Task Dependence Graph Generation

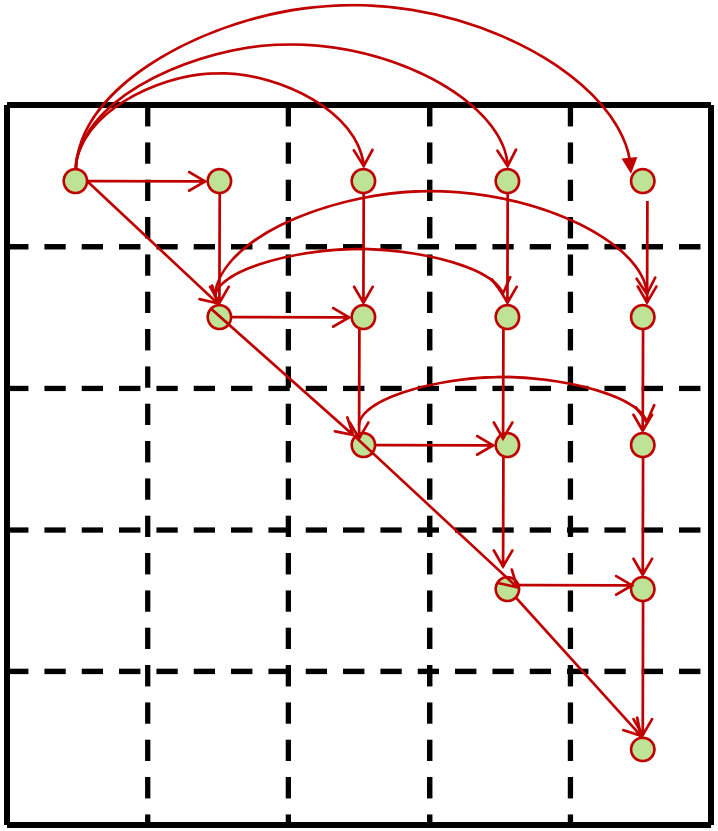
- Task Dependence Graph
  - DAG
    - Vertices – Tiles or tasks
    - Edges – Dependence between the corresponding tasks
  - Vertices and edges may be assigned weights
    - Vertex weight – based on task execution
    - Edge weight – based on communication between tasks
    - Current implementation
      - Unit weights for vertices and zero weights for edges

Task Dependence Graph Generation

- Compile-time generation of TDG generation code
  - Code to enumerate the vertices
    - Scan the iteration space polytopes of all statements in the tiled domain, projected to contain only the supernode iterators
    - CLooG loop generator is used for scanning the polytopes
  - Code to create the edges
    - Requires extraction of inter-tile dependences

Inter-tile Dependence Abstraction





Compiler-Assisted Dynamic Scheduling for Effective Parallelization of Loop Nests on Multicore Processors, PPoPP 2009

Inter-tile Dependence Abstraction

$$\begin{pmatrix} D_s & 0 \\ 0 & D_t \\ \hline -I & H \end{pmatrix} \cdot \begin{pmatrix} \vec{i}_s \\ \vec{i}_t \\ \vec{n} \\ 1 \end{pmatrix} \begin{pmatrix} \geq \vec{0} \\ \hline = \vec{0} \end{pmatrix}$$

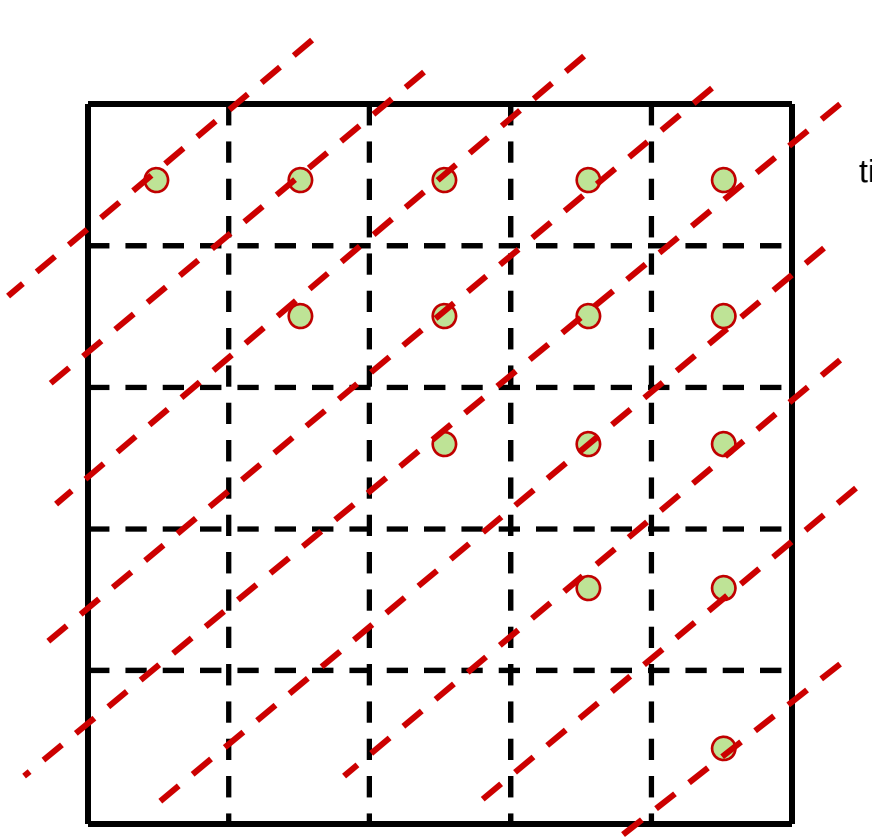
$$\begin{pmatrix} T_s & 0 & 0 & 0 \\ 0 & D_s & 0 & 0 \\ 0 & 0 & T_t & 0 \\ 0 & 0 & 0 & D_t \\ \hline 0 & -I & 0 & H \end{pmatrix} \cdot \begin{pmatrix} \vec{iT}_s \\ \vec{i}_s \\ \vec{iT}_t \\ \vec{i}_t \\ \vec{n} \\ 1 \end{pmatrix} \begin{pmatrix} \geq \vec{0} \\ \hline = \vec{0} \end{pmatrix}$$

$$\begin{pmatrix} D'_s & 0 \\ 0 & D'_t \end{pmatrix} \cdot \begin{pmatrix} \vec{iT'}_s \\ \vec{iT'}_t \\ \vec{n} \\ 1 \end{pmatrix} \geq \vec{0}$$

- Dependences expressed in tiled domain using a higher-dimensional dependence polytope
- Project out intra-tile iterators from the system
- Resulting system characterizes inter-tile dependences
- Scan the system using CLooG to generate loop structure that has
  - Source tile iterators as outer loops
  - Target tile iterators as inner loops
- Loop structure gives code to generate edges in TDG

Compiler-Assisted Dynamic Scheduling for Effective Parallelization of Loop Nests on Multicore Processors, PPoPP 2009

Static Affine Scheduling



time

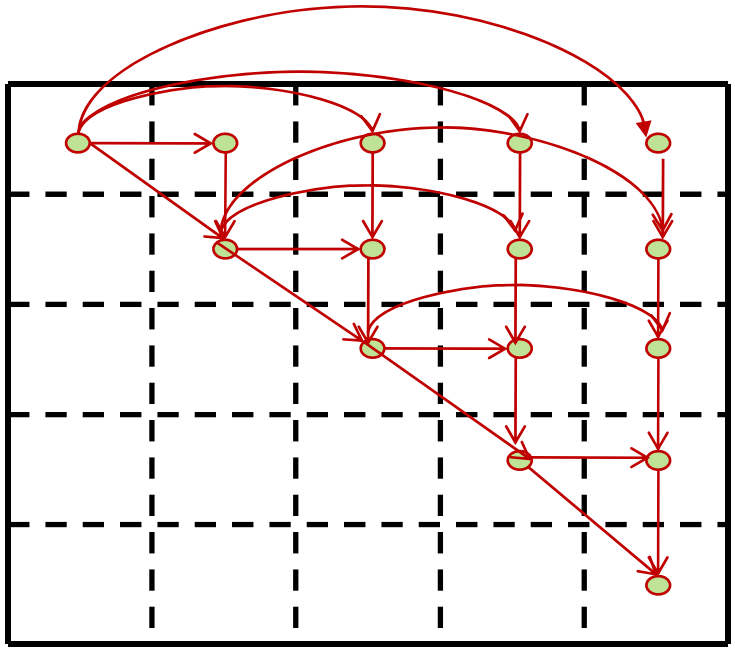
Affine Schedule

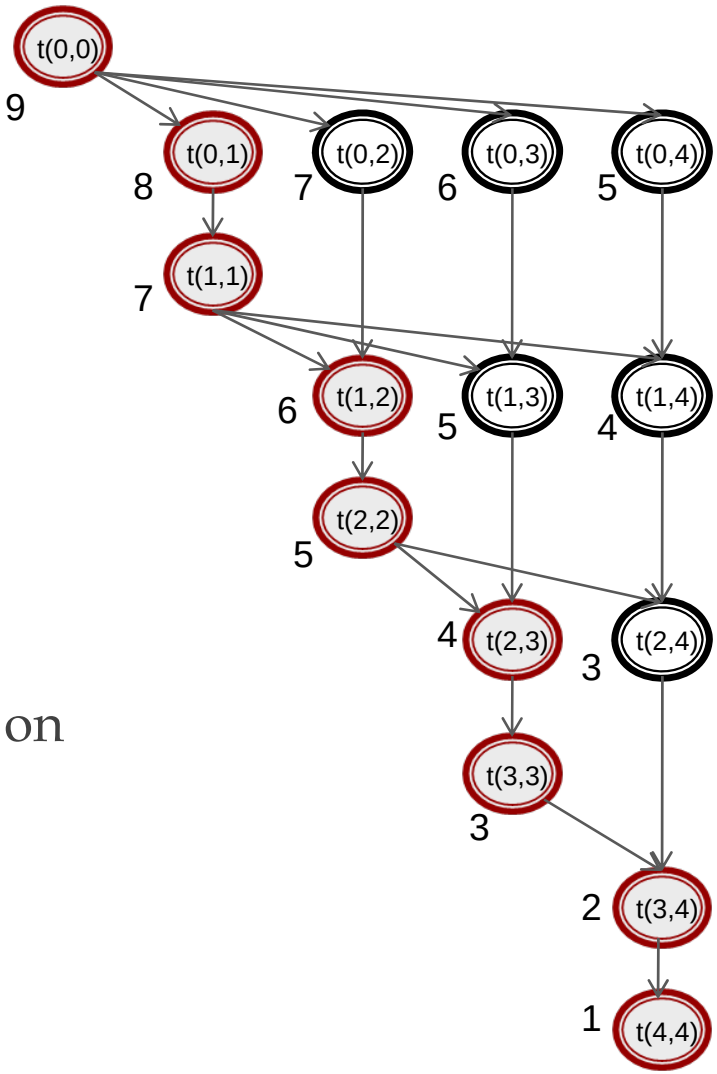
C1	C2
t(4,4)	
t(3,4)	
t(3,3)	t(2,4)
t(2,3)	t(1,4)
t(0,4)	
t(2,2)	t(1,3)
t(1,2)	t(0,3)
t(1,1)	t(0,2)
t(0,1)	
t(0,0)	

Dynamic Scheduling

- Scheduling strategy: *critical path analysis* for prioritizing tasks in TDG
- Priority metrics associated with vertices
  - *topL(v)* - length of the longest path from the source vertex (i.e., the vertex with no predecessors) in G to v, excluding the vertex weight of v
  - *bottomL(v)* - length of the longest path from v to the sink (vertex with no children), including the vertex weight of v
- Tasks are prioritized based on
  - sum of their top and bottom levels or
  - just the bottom level

Dynamic Scheduling

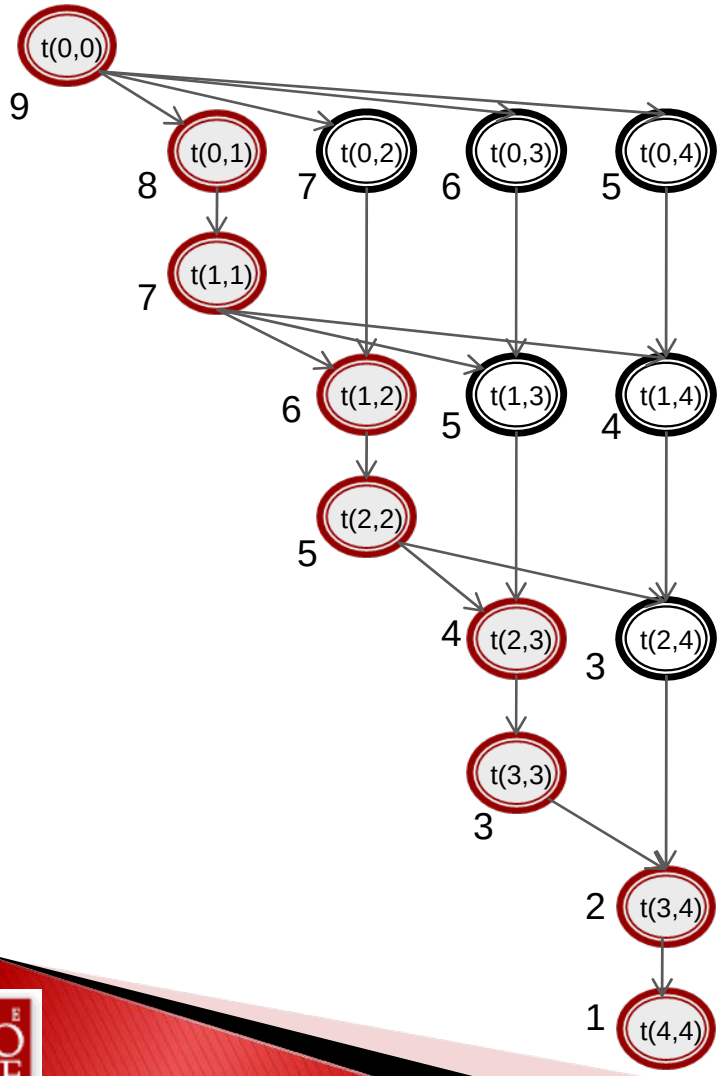


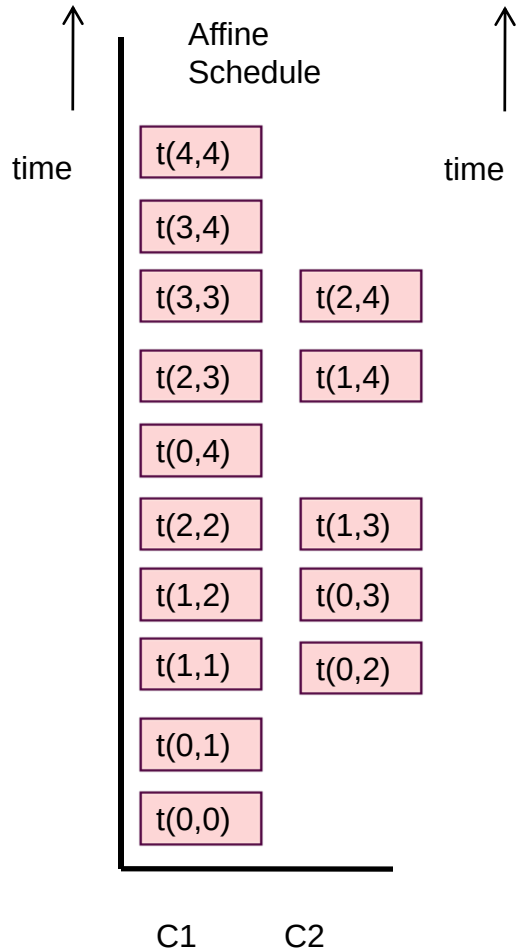


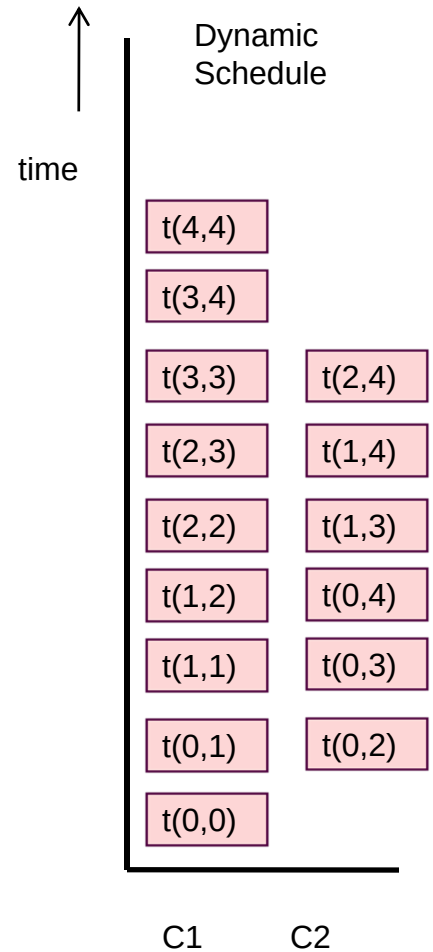
- Tasks are scheduled for execution based on
  - completion of predecessor tasks
  - *bottomL(v)* priority
  - availability of processor core

Compiler-Assisted Dynamic Scheduling for Effective Parallelization of Loop Nests on Multicore Processors, PPoPP 2009

Affine vs. Dynamic Scheduling







Compiler-Assisted Dynamic Scheduling for Effective Parallelization of Loop Nests on Multicore Processors, PPoPP 2009

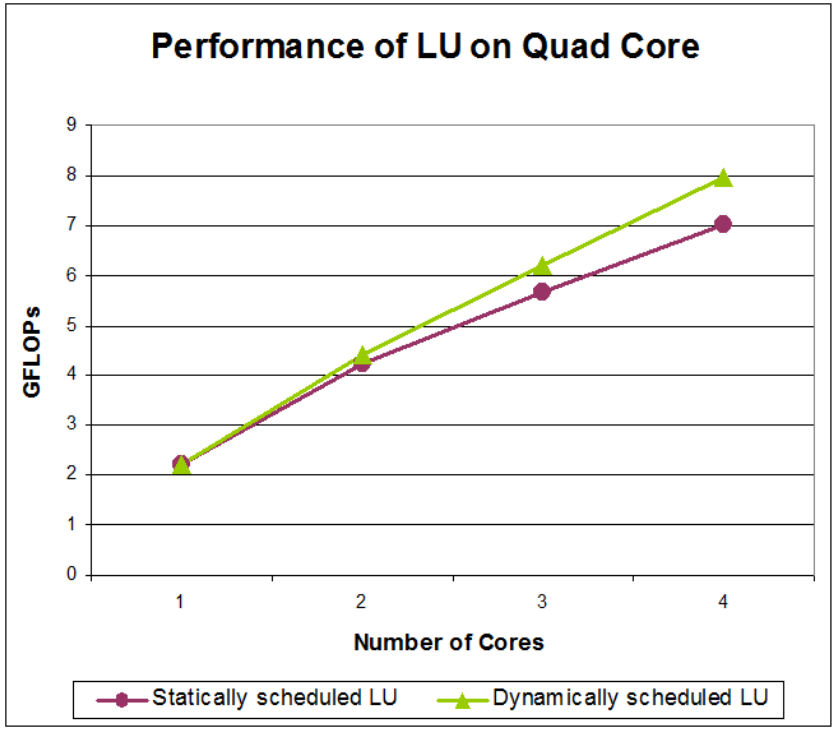
Run-time Execution

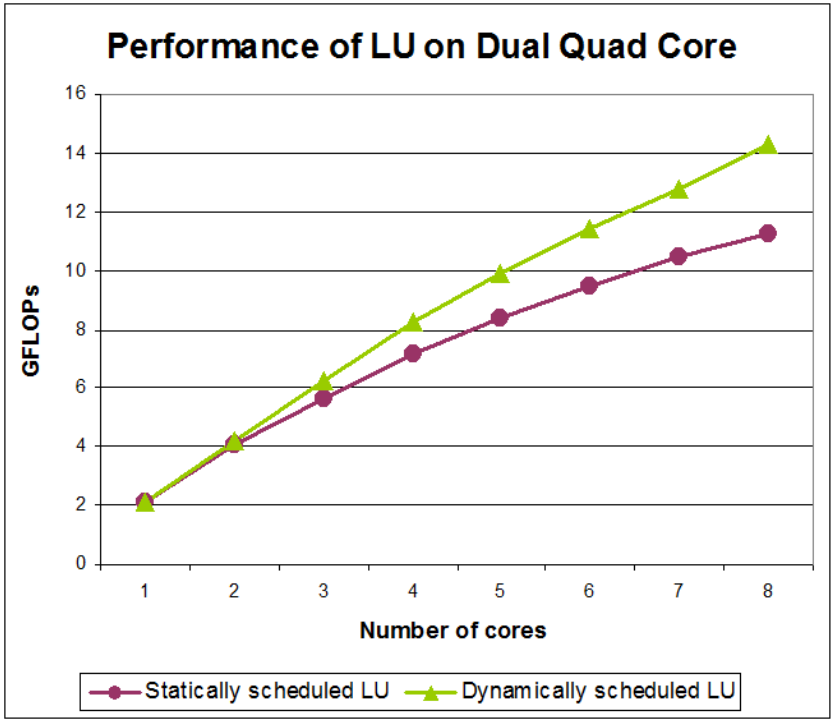
```
Execute TDG generation code to create a DAG G
Calculate topL(v) and bottomL(v) for each vertex v in G,
    to prioritize the vertices
Create a Priority Queue PQ
PQ.insert ( vertices with no parents in G)
while not all vertices in G are processed do
    taskid = PQ.extract( )
    Execute taskid // Compute code
    Remove all outgoing edges of taskid from G
    PQ.insert ( vertices with no parents in G)
end while
```

Experiments

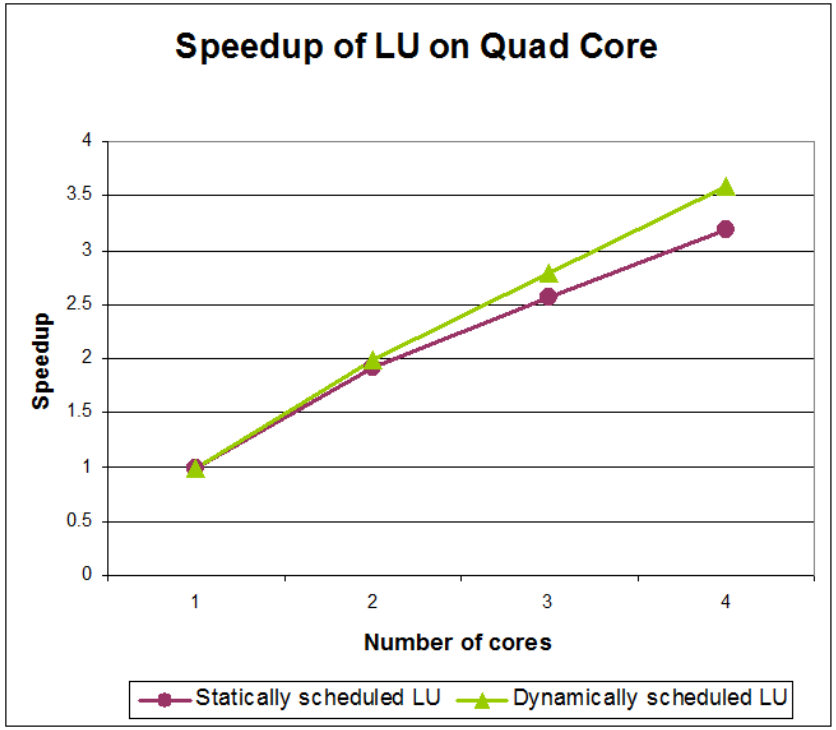
- Experimental Setup
  - a quad-core Intel Core 2 Quad Q6600 CPU
    - clocked at 2.4 GHz (1066 MHz FSB)
    - 8MB L2 cache (4MB shared per core pair)
    - Linux kernel version 2.6.22 (x86-64)
  - a dual quad core Intel Xeon(R) E5345 CPU
    - clocked at 2.33 GHz
    - each chip having a 8MB L2 cache (4MB shared per core pair)
    - Linux kernel version 2.6.18
  - ICC 10.x compiler
    - Options: -fast –funroll-loops (-openmp for parallelized code)

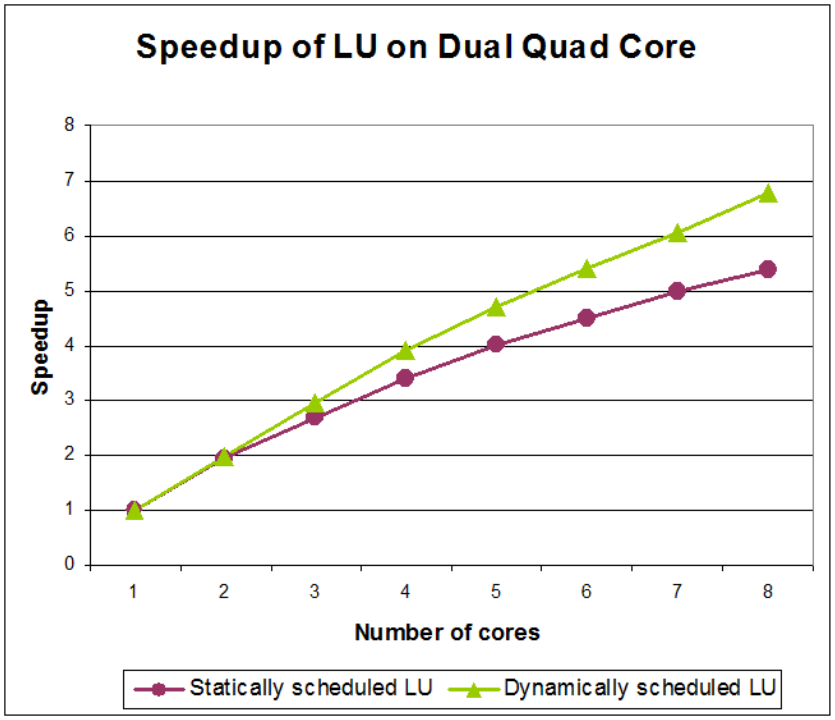
Experiments



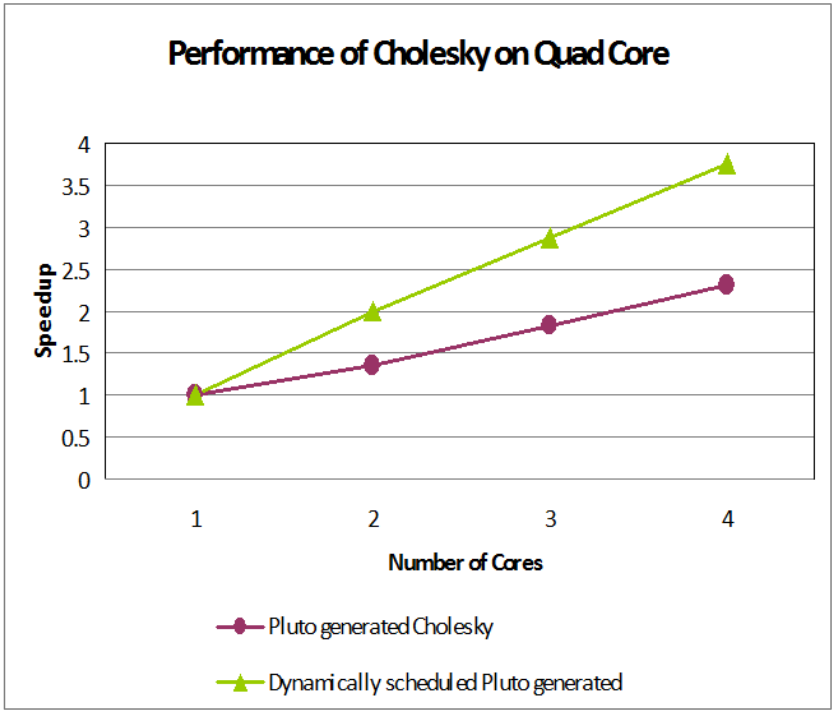


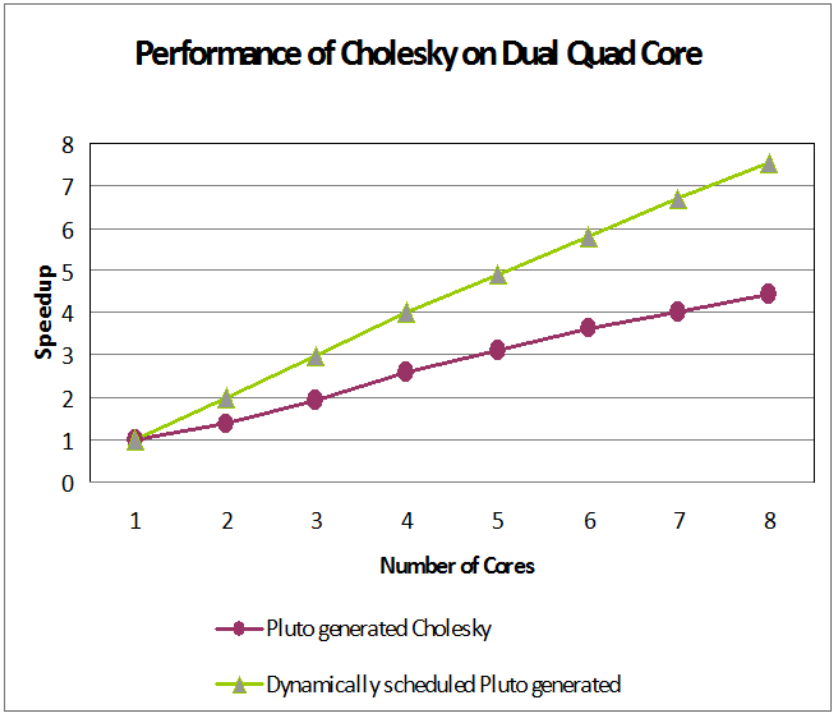
Experiments





Experiments





Discussion

- Absolute achieved GFLOPs performance is currently lower than the machine peak by over a factor of 2
- Single-node performance sub-optimal
  - No pre-optimized tuned kernels
    - E.g. BLAS kernels like DGEMM in LU code
- Work in progress to provide
  - Identification of tiles where pre-optimized kernels can be substituted

Related Work

- Dongarra et al. - PLASMA (Parallel Linear Algebra for Scalable Multi-core Architectures)
  - LAPACK codes optimization
    - Manual rewriting of LAPACK routines
    - Run-time scheduling framework
- Robert van de Geijn et al. - FLAME
- Dynamic run-time parallelization [LRPD, Mitosis, etc.]
  - Basic difference: Dynamic scheduling of loop computations amenable to compile-time characterization of dependences
- Plethora of work on DAG scheduling

Summary

- Developed a fully-automatic approach for asynchronous load balanced parallel execution on multi-core systems
- Basic idea
  - To automatically generate tiled code along with additional helper code at compile time
  - Role of helper code at run time
    - to dynamically extract inter-tile data dependences
    - to dynamically schedule the tiles on the processor cores
- Achieved significant improvement over programs automatically parallelized using affine compiler frameworks

Thank You