Tiling for Dynamic Scheduling

Ravi Teja Mullapudi
Uday Bondhugula

CSA, Indian Institue of Science

January 20, 2014

Table of Contents

Table of Contents

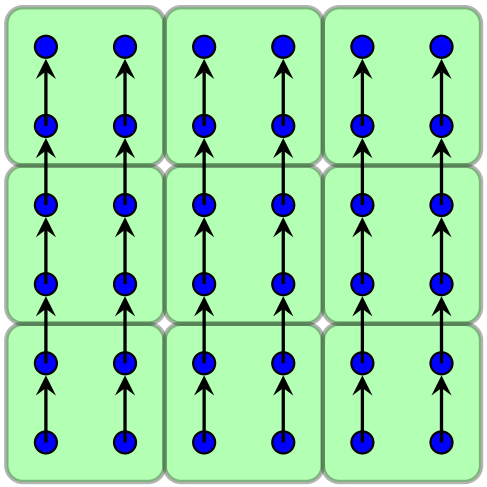
Tiling

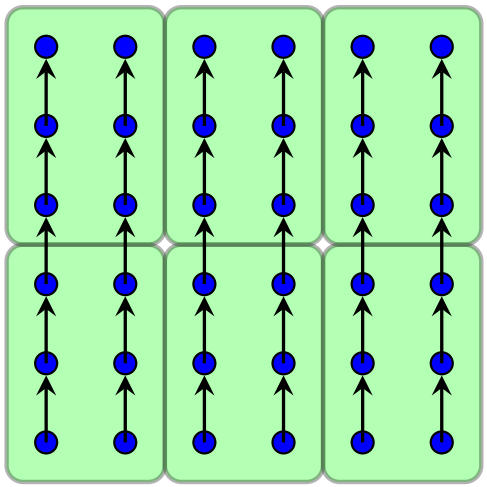
Definition (Computation Graph)

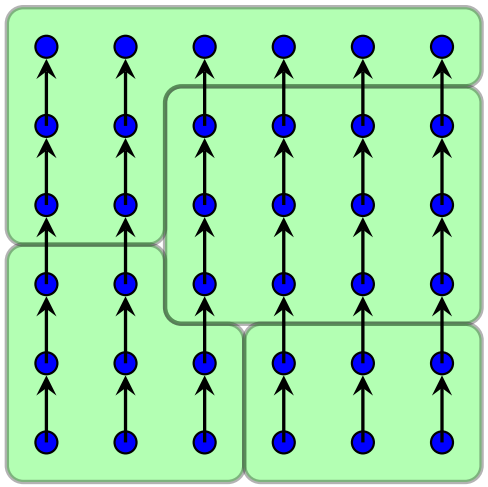
A directed acyclic graph in which the nodes represent operations and edges denote dependences between operations

Definition (Tiling)

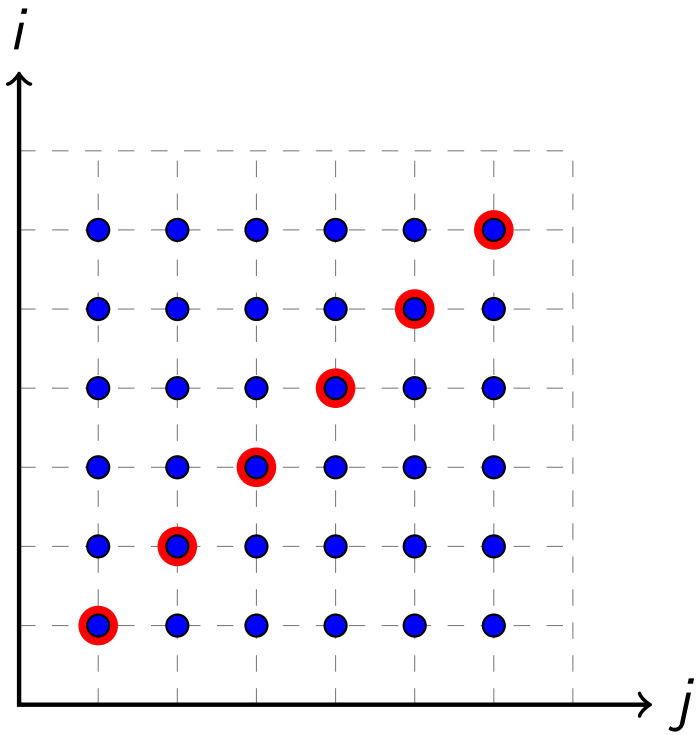
Partitioning a computation graph into *atomic* units of execution







Iteration spaces, Dependences and Hyperplanes

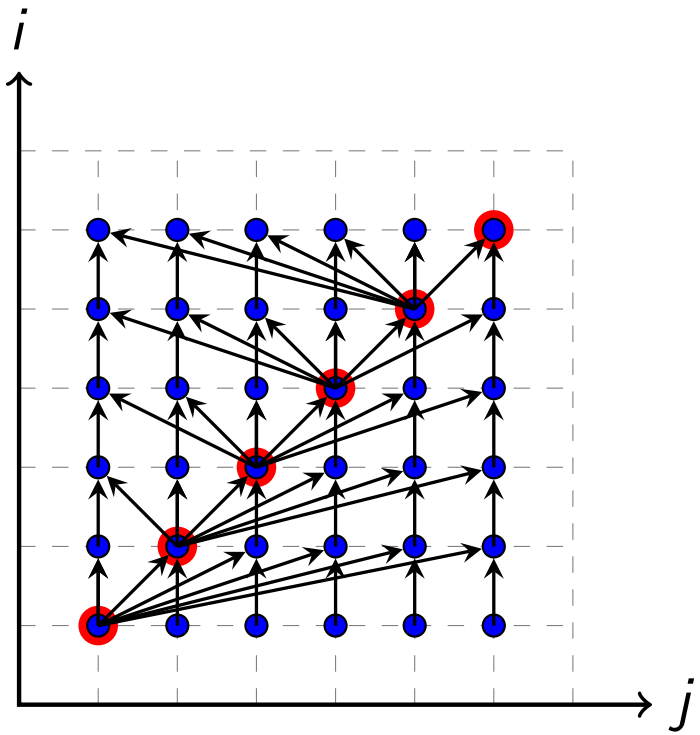



```
for (i = 0; i < N; i++){
    for ( j = 0; j < N; j++) {
        A[j] = A[j] + B[i]; s1
        if (i == j)
            B[i+1] = A[j]; s2
    }
}
```

Iteration spaces

$$D^s = \left\{ \vec{i_s} \;\middle|\; \vec{i_s} \in \mathbb{Z}^n, A\vec{i_s} + b \geq 0 \right\}$$

Iteration spaces, Dependences and Hyperplanes

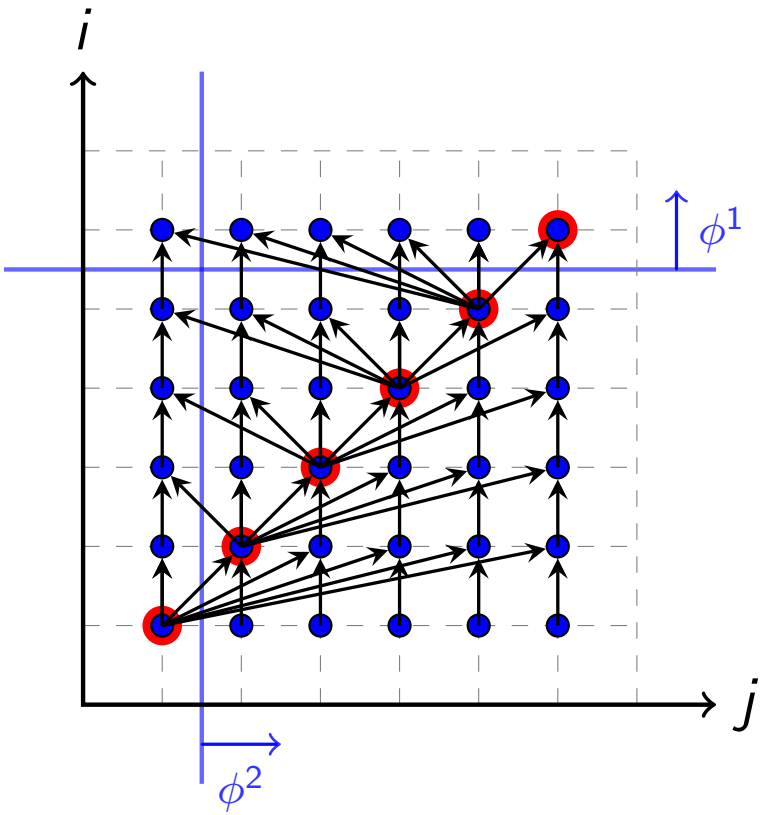



```
for (i = 0; i < N; i++){
    for ( j = 0; j < N; j++) {
        A[j] = A[j] + B[i]; s1
        if (i == j)
            B[i+1] = A[j]; s2
    }
}
```

Dependences

$$P_e = \left\{ \langle \vec{i_s}, \vec{i_t} \rangle \;\middle|\; \vec{i_s} \in D^s, \vec{i_t} \in D^t, \vec{i_s} = h_e(\vec{i_t}), e : s \rightarrow t \right\}$$

Iteration spaces, Dependences and Hyperplanes

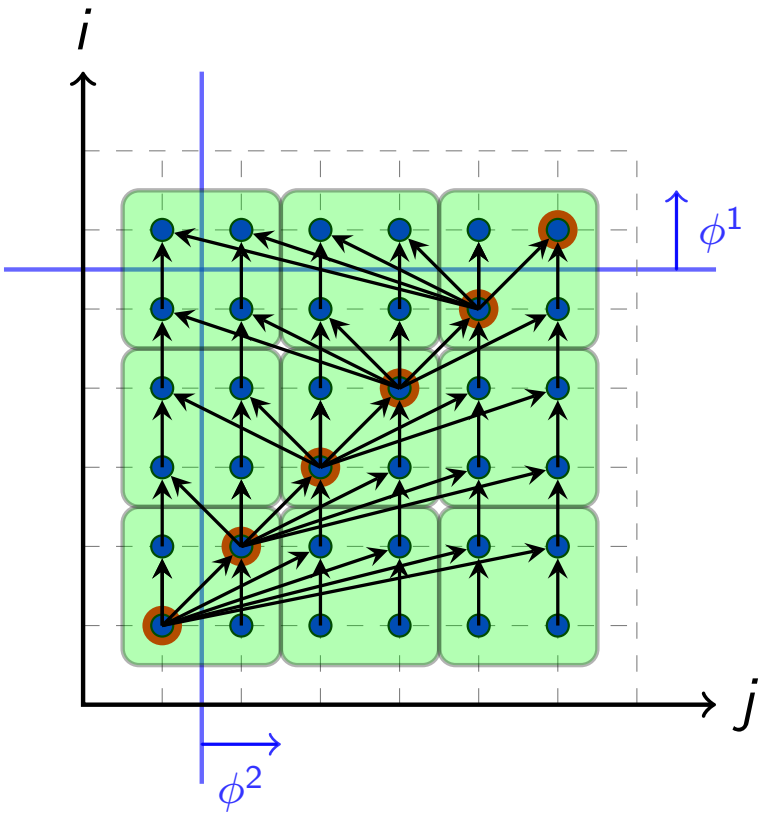



```
for (i = 0; i < N; i++){
    for ( j = 0; j < N; j++) {
        A[j] = A[j] + B[i]; s1
        if (i == j)
            B[i+1] = A[j]; s2
    }
}
```

Hyperplanes

$$\phi_s(\vec{i_s}) = \vec{h} \cdot \vec{i_s} + h_0$$

Iteration spaces, Dependences and Hyperplanes

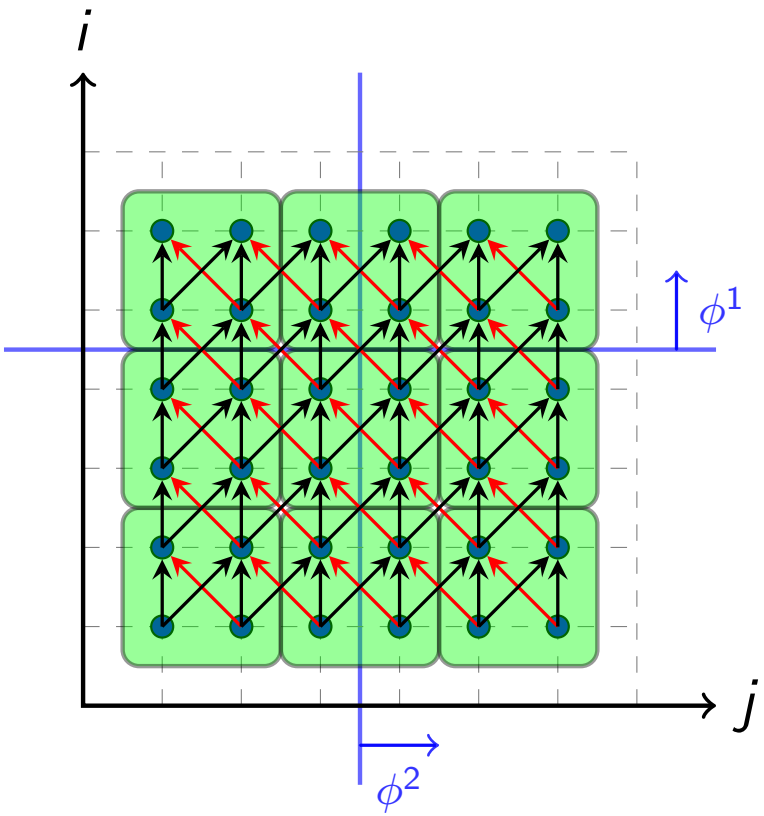


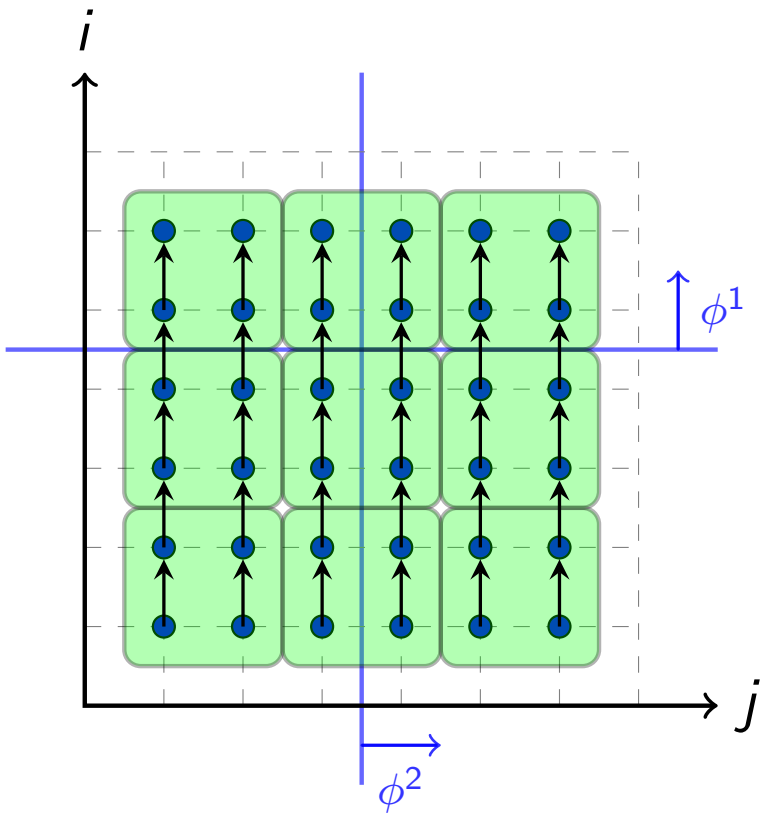

```
for (i = 0; i < N; i++){
    for ( j = 0; j < N; j++) {
        A[j] = A[j] + B[i]; s1
        if (i == j)
            B[i+1] = A[j]; s2
    }
}
```

Hyperplanes

$$\phi_s(\vec{i_s}) = \vec{h} \cdot \vec{i_s} + h_0$$

Validity Conditions



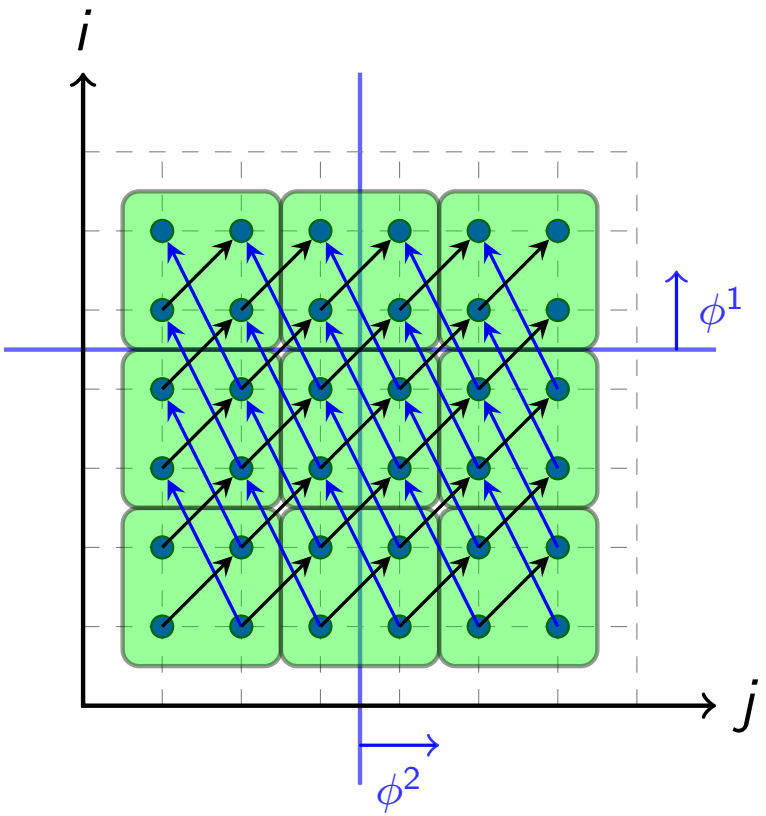


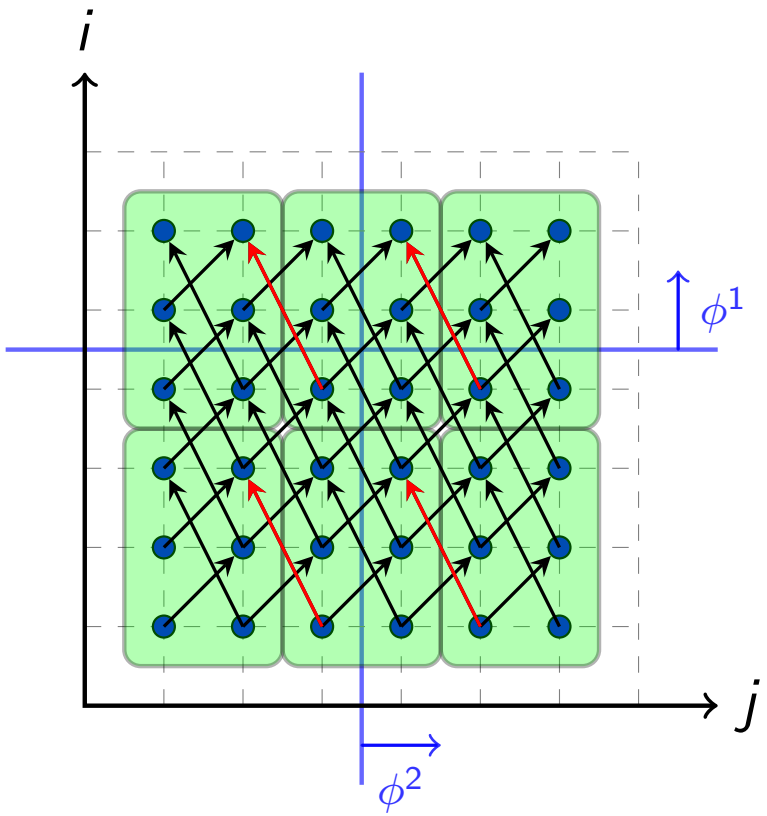
Non-Negative Dependence Components

$$\phi_t(\vec{i_t}) - \phi_s(\vec{i_s}) \geq 0 \, , \langle \vec{i_s}, \vec{i_t} \rangle \in P_e, e : s \rightarrow t \, (HD \geq 0)$$

- Irigoin and Triolet, Lim and Lam, Griebl, Pluto

Validity Conditions





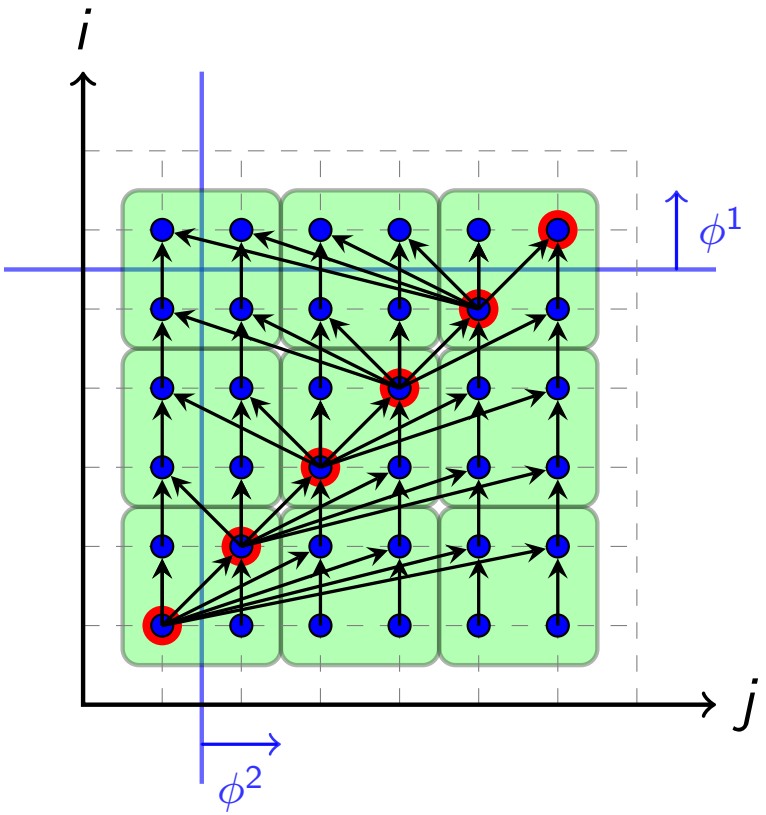
Lexicographically Non-Negative Tile Dependences

$$\lfloor HD \rfloor \succcurlyeq \vec{0}$$

- Xue

Table of Contents

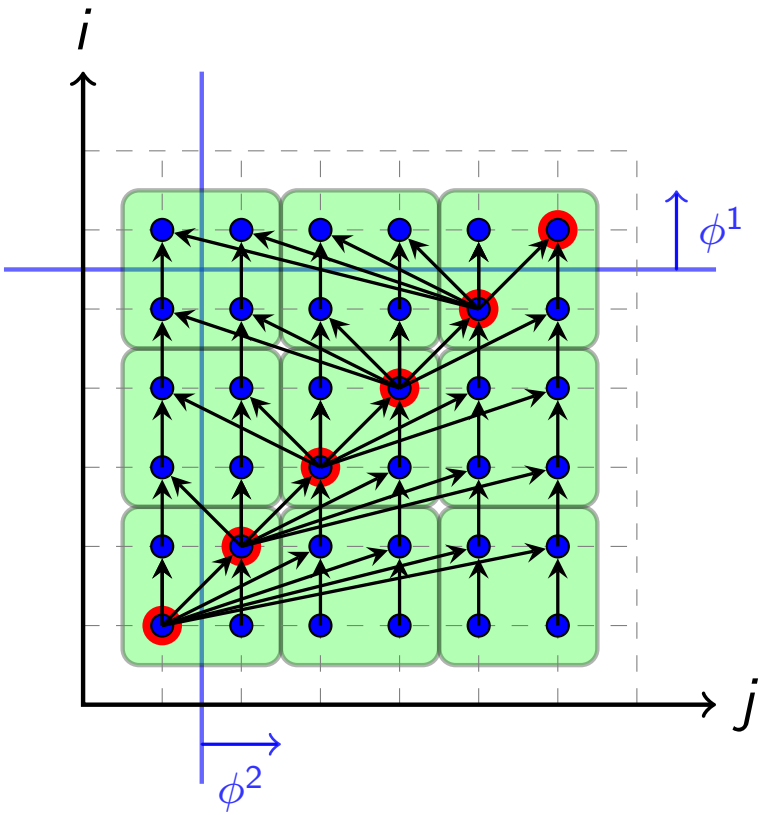
Conservative Validity Conditions

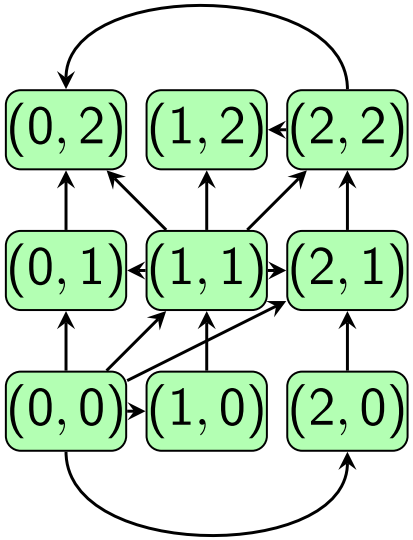



```
for (i = 0; i < N; i++){
    for ( j = 0; j < N; j++) {
        A[j] = A[j] + B[i]; s1
        if (i == j)
            B[i+1] = A[j]; s2
    }
}
```

- Negative dependence components along ϕ^2 but tiling is valid
- Inter-tile dependences lexicographically negative
- Tile sizes effect validity

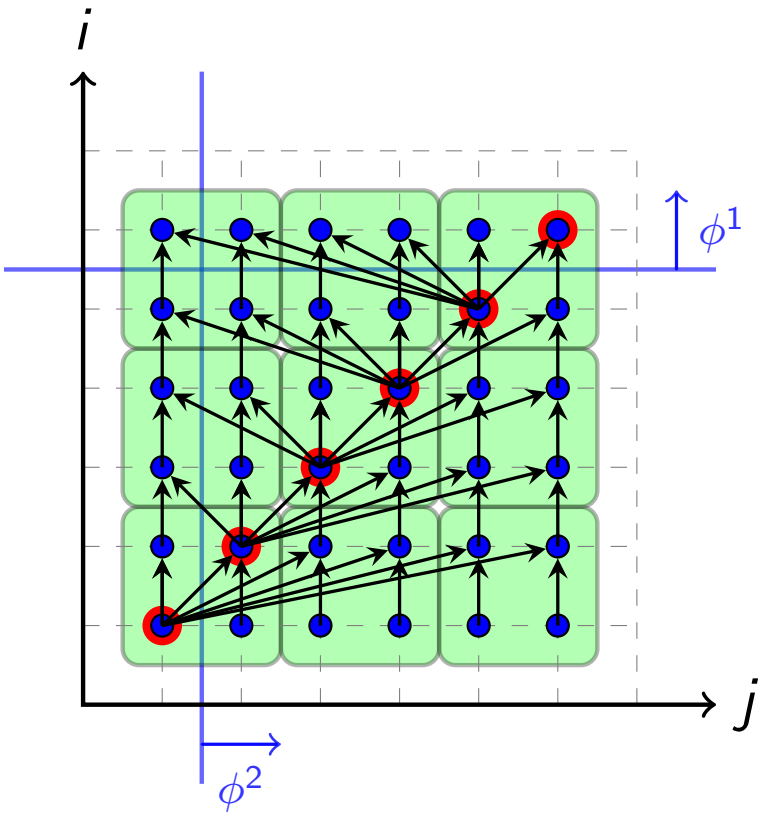
Conservative Validity Conditions

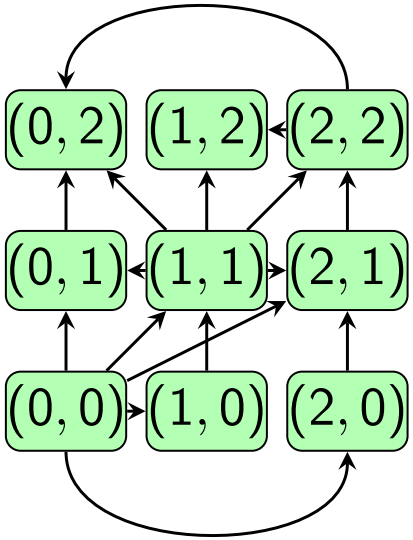




- Negative dependence components along ϕ^2 but tiling is valid
- Inter-tile dependences lexicographically negative
- Tile sizes effect validity

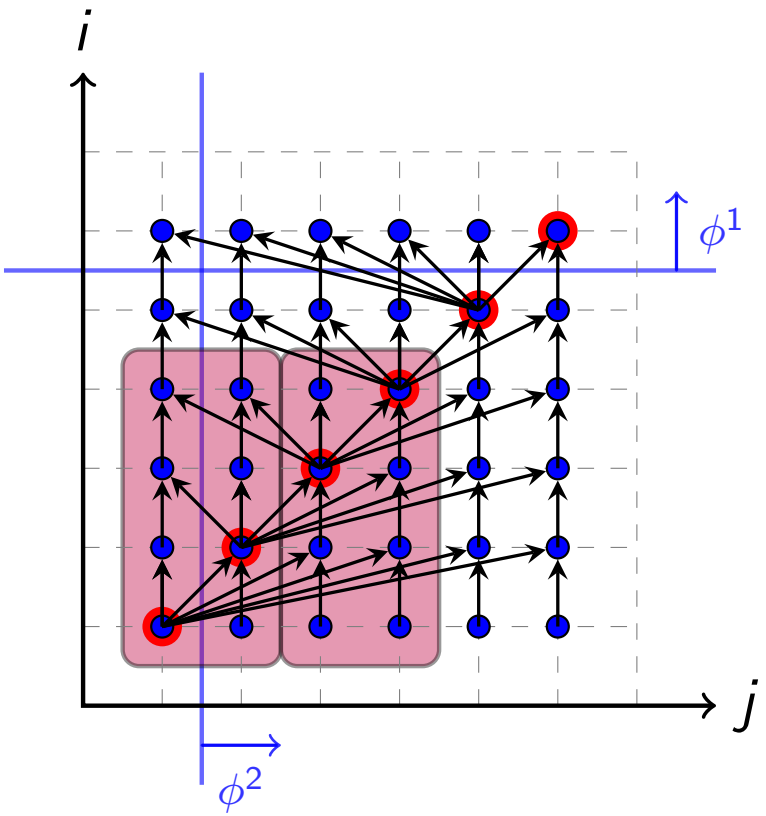
Conservative Validity Conditions

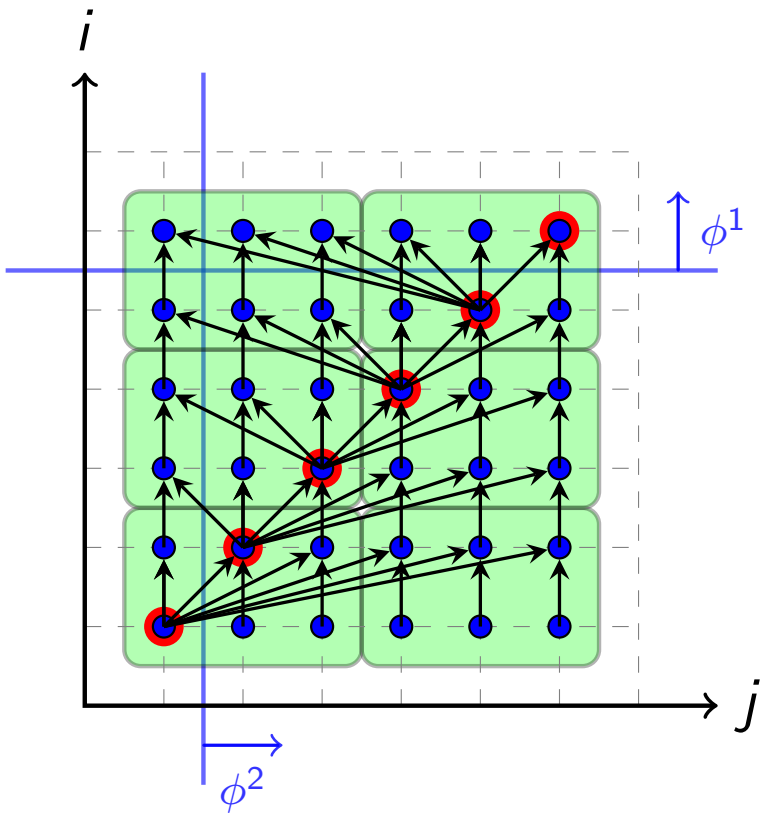




- Negative dependence components along ϕ^2 but tiling is valid
- Inter-tile dependences lexicographically negative
- Tile sizes effect validity

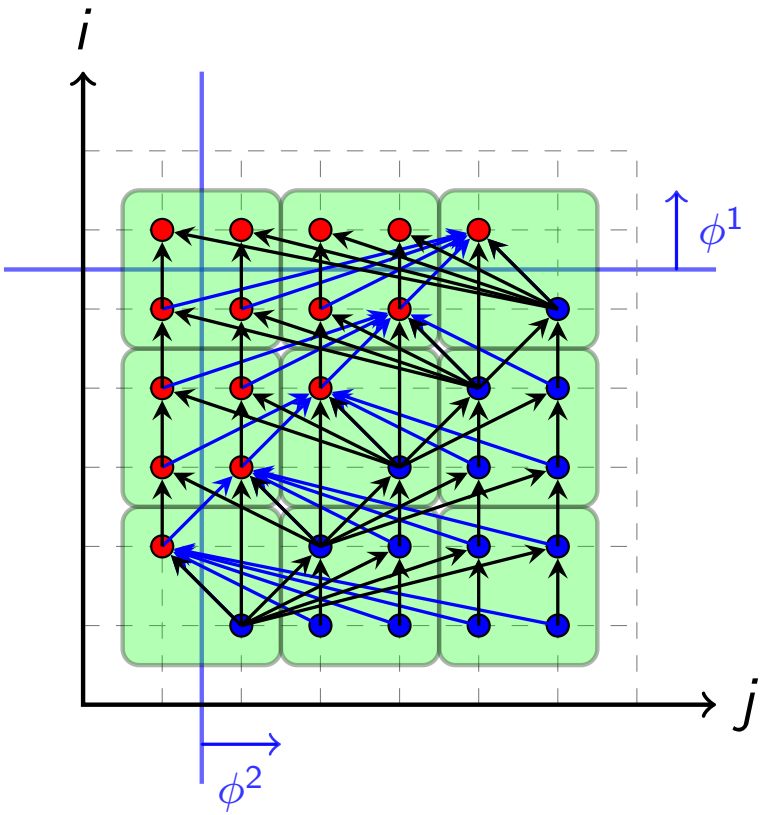
Conservative Validity Conditions





- Negative dependence components along ϕ^2 but tiling is valid
- Inter-tile dependences lexicographically negative
- Tile sizes effect validity

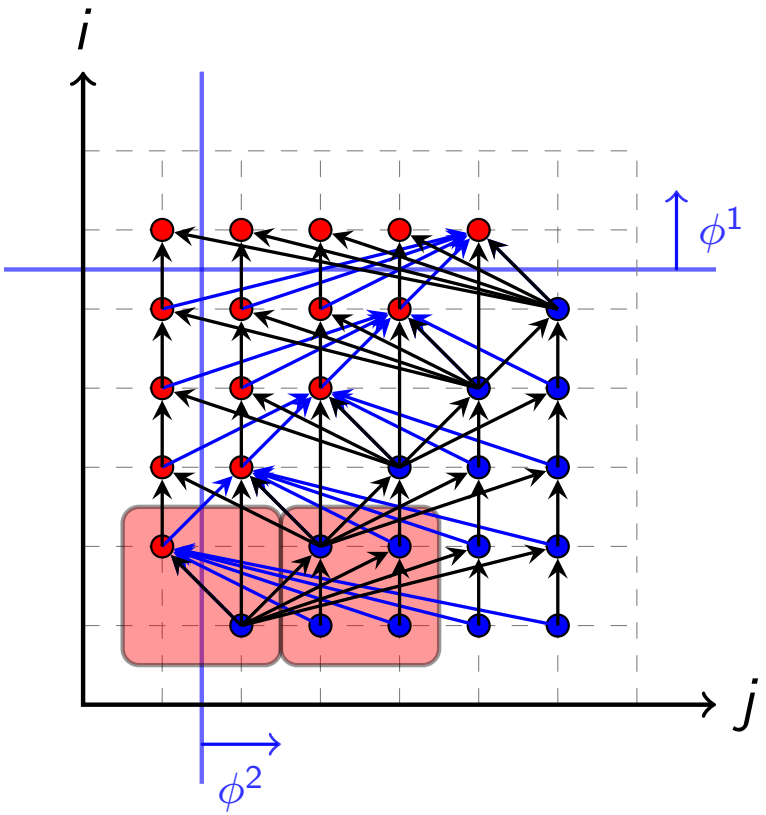
Uniformly Tiling Iteration Spaces




```
for (i = 0; i < N; i++){
    for ( j = 0; j < N; j++) {
        if (j > i)
            A[j] = A[j] + A[i]; s1
        if (j < i)
            A[j] = A[j] + A[i]; s2
    }
}
```

- Tile dependence graph has cycles
- Splitting breaks cycles

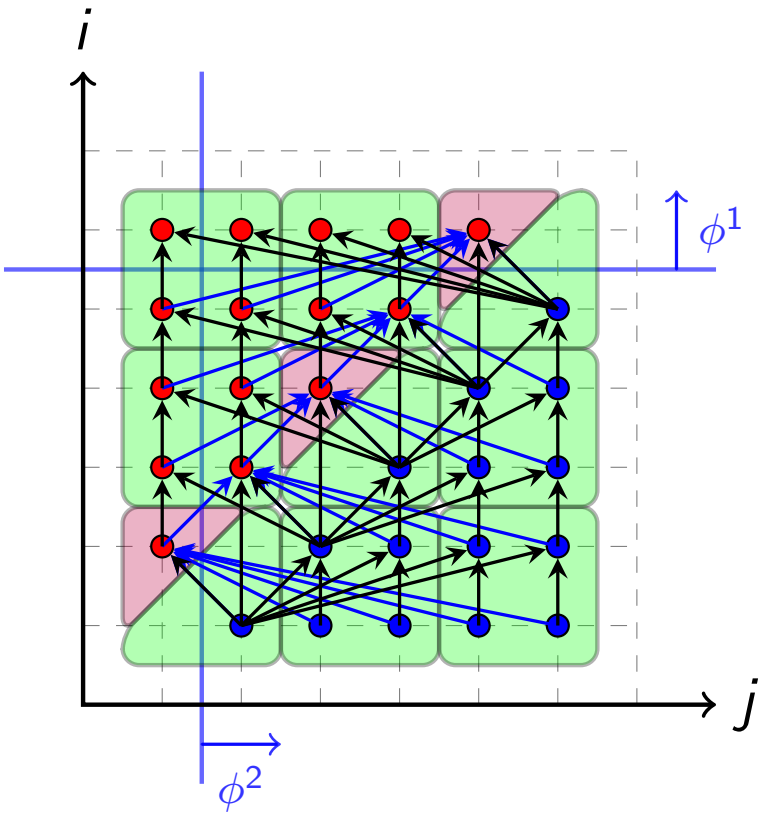
Uniformly Tiling Iteration Spaces

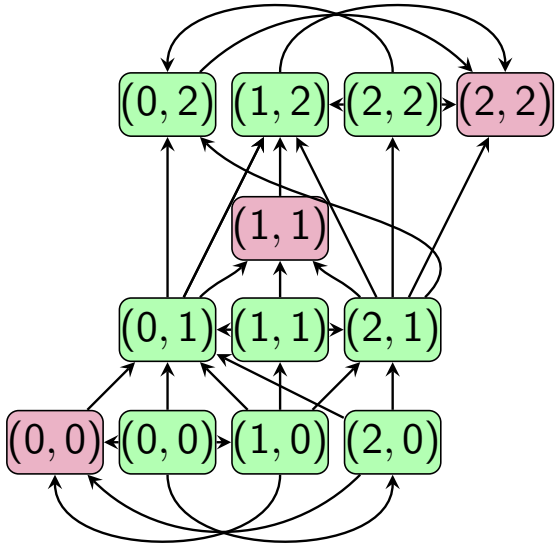



```
for (i = 0; i < N; i++){
    for ( j = 0; j < N; j++) {
        if (j > i)
            A[j] = A[j] + A[i]; s1
        if (j < i)
            A[j] = A[j] + A[i]; s2
    }
}
```

- Tile dependence graph has cycles
- Splitting breaks cycles

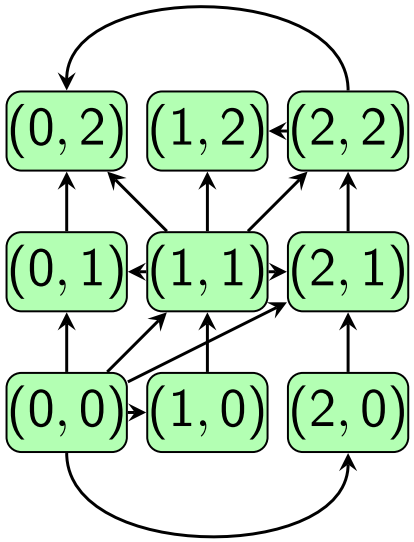
Uniformly Tiling Iteration Spaces

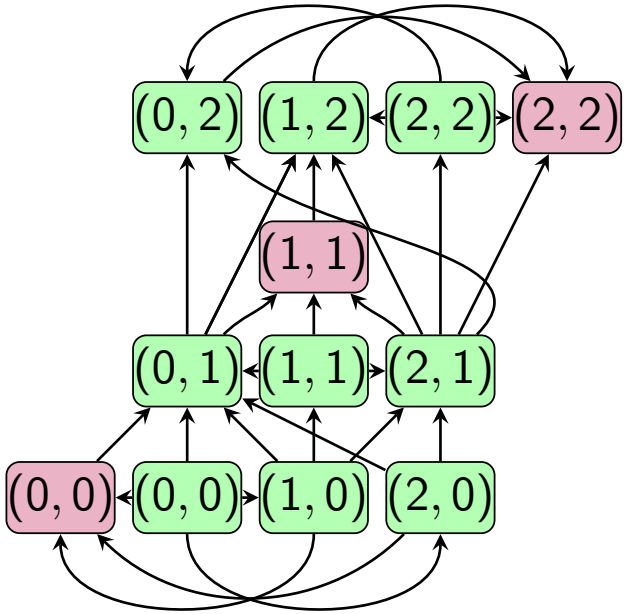




- Tile dependence graph has cycles
- Splitting breaks cycles

Dynamic Scheduling





- Not easy to come up with a static schedule for the tiles
- Dynamic task scheduler

Table of Contents

General Validity Condition

Definition (Valid Tiling)

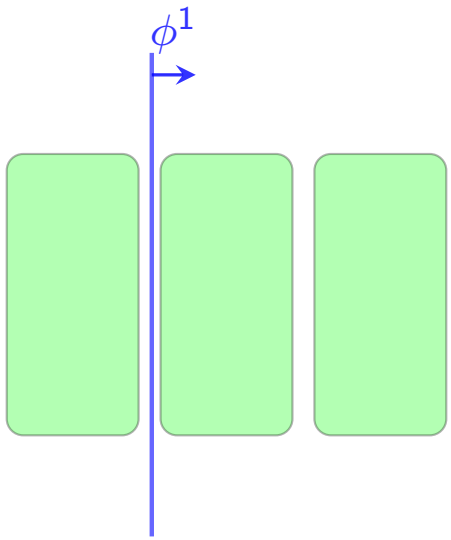
A set of hyperplanes ϕ^1, ϕ^2, ..., ϕ^k with tile size τ_i for ϕ^i, is a valid tiling of an iteration space if the dependence graph of k-dimensional tiles formed by the hyperplanes with their respective tile sizes is cycle-free.

- Transitive closure of dependence relations
  - Precise computation might be infeasible
  - Expensive operation even for an approximation
- Conservative cycle detection

Localizing Dependences

Theorem

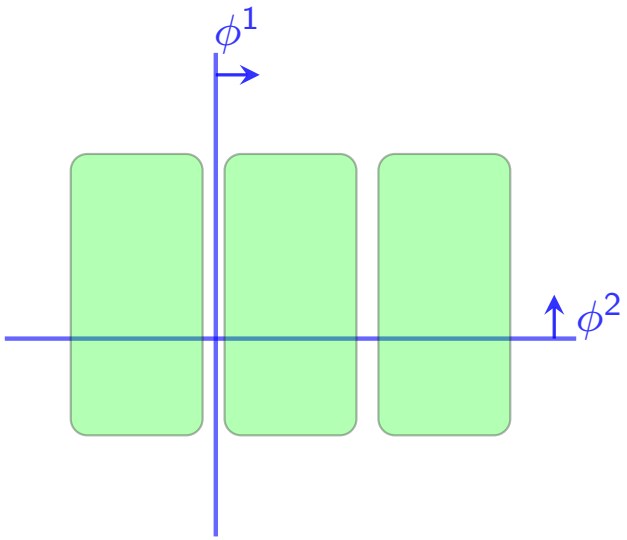
$\{\phi^1, \ldots, \phi^k\}$ with tile size τ_i for ϕ^i is a valid tiling of an iteration space, if $\{\phi^1, \ldots, \phi^{k-1}\}$ is a valid tiling and ϕ^k is a valid one-dimensional tiling of each $k-1$ dimensional tile formed by $\{\phi^1, \ldots, \phi^{k-1}\}$.



Localizing Dependences

Theorem

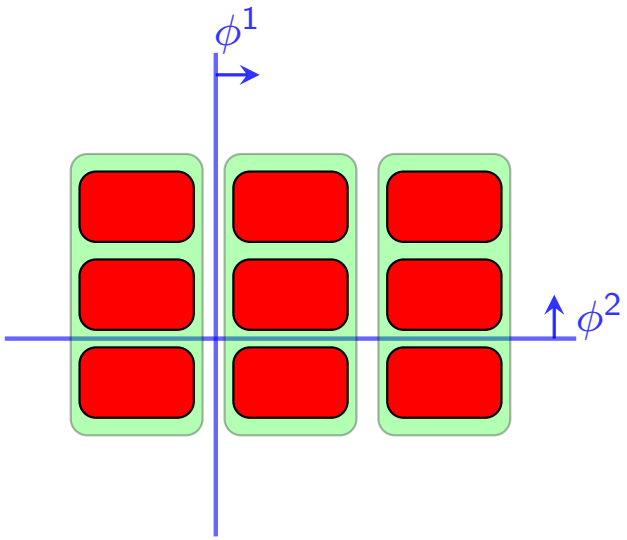
$\{\phi^1, \ldots, \phi^k\}$ with tile size τ_i for ϕ^i is a valid tiling of an iteration space, if $\{\phi^1, \ldots, \phi^{k-1}\}$ is a valid tiling and ϕ^k is a valid one-dimensional tiling of each $k-1$ dimensional tile formed by $\{\phi^1, \ldots, \phi^{k-1}\}$.



Localizing Dependences

Theorem

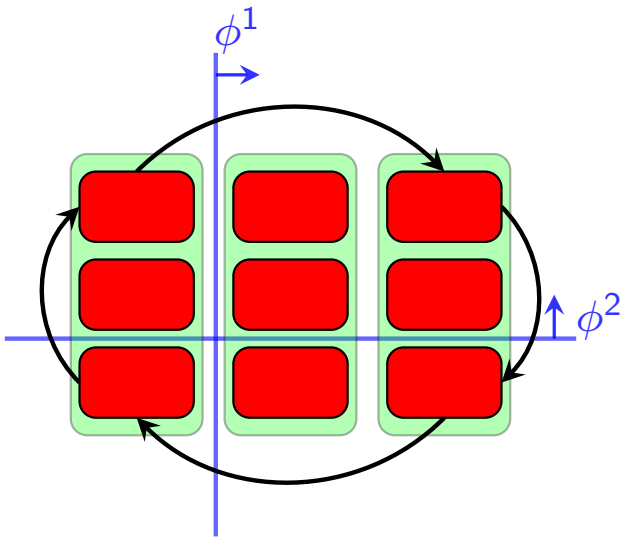
$\{\phi^1, \ldots, \phi^k\}$ with tile size τ_i for ϕ^i is a valid tiling of an iteration space, if $\{\phi^1, \ldots, \phi^{k-1}\}$ is a valid tiling and ϕ^k is a valid one-dimensional tiling of each $k-1$ dimensional tile formed by $\{\phi^1, \ldots, \phi^{k-1}\}$.



Localizing Dependences

Theorem

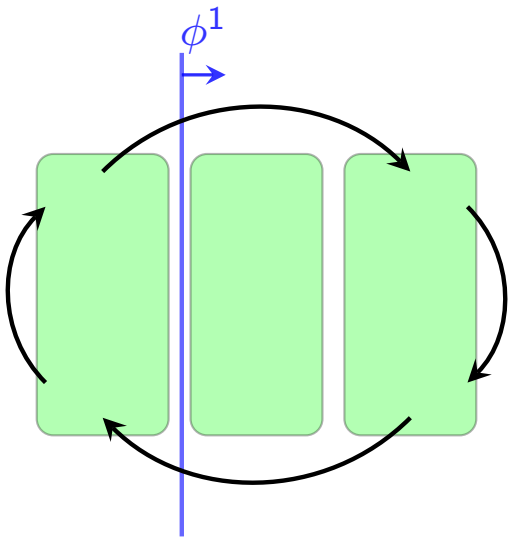
$\{\phi^1, \ldots, \phi^k\}$ with tile size τ_i for ϕ^i is a valid tiling of an iteration space, if $\{\phi^1, \ldots, \phi^{k-1}\}$ is a valid tiling and ϕ^k is a valid one-dimensional tiling of each $k-1$ dimensional tile formed by $\{\phi^1, \ldots, \phi^{k-1}\}$.



Localizing Dependences

Theorem

$\{\phi^1, \ldots, \phi^k\}$ with tile size τ_i for ϕ^i is a valid tiling of an iteration space, if $\{\phi^1, \ldots, \phi^{k-1}\}$ is a valid tiling and ϕ^k is a valid one-dimensional tiling of each $k-1$ dimensional tile formed by $\{\phi^1, \ldots, \phi^{k-1}\}$.



Localizing Dependences

Dependences in the transformed space

$$\Big\{\langle\langle T_s^1, \ldots, T_s^k, \vec{i_s}\rangle, \langle T_t^1, \ldots, T_t^k, \vec{i_t}\rangle\rangle \mid \langle \vec{i_s}, \vec{i_t}\rangle \in P_e, e : s \rightarrow t,$$
$$1 \leq l \leq k, \tau_l * T_s^l \leq \phi_s^l(\vec{i_s}) \leq \tau_l * (T_s^l + 1) - 1,$$
$$\tau_l * T_t^l \leq \phi_t^l(\vec{i_t}) \leq \tau_i * (T_t^l + 1) - 1\Big\}$$

Inter-Tile Dependences

P_e denotes the inter-tile dependence polyhedron between k-dimensional tiles formed by $\langle \phi^1, \ldots, \phi^k \rangle$ due to the depedence edge e.

Localizing Dependences

Dependences in the transformed space

$$\Big\{\langle\langle T_s^1, \ldots, T_s^k, \vec{i_s}\rangle, \langle T_t^1, \ldots, T_t^k, \vec{i_t}\rangle\rangle \mid \langle \vec{i_s}, \vec{i_t}\rangle \in P_e, e : s \rightarrow t,$$
$$1 \leq l \leq k, \tau_l * T_s^l \leq \phi_s^l(\vec{i_s}) \leq \tau_l * (T_s^l + 1) - 1,$$
$$\tau_l * T_t^l \leq \phi_t^l(\vec{i_t}) \leq \tau_i * (T_t^l + 1) - 1\Big\}$$

Inter-Tile Dependences

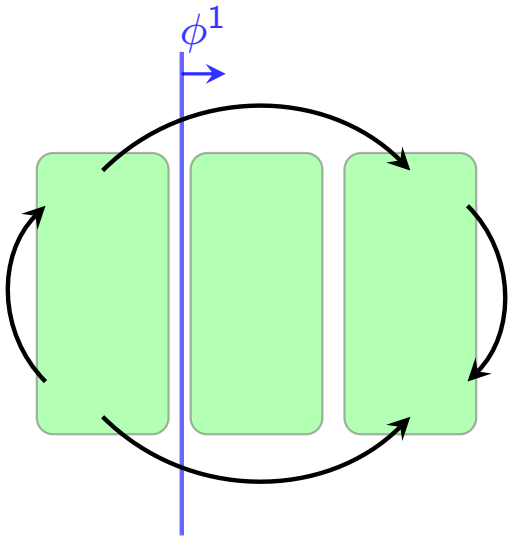
P_e denotes the inter-tile dependence polyhedron between k-dimensional tiles formed by $\langle \phi^1, \ldots, \phi^k \rangle$ due to the depedence edge e.

- Computed by projecting out dimensions inner to the tiling dimension ϕ^k (for both source and target statements)

Localizing Dependences

Restricted Tile Dependence

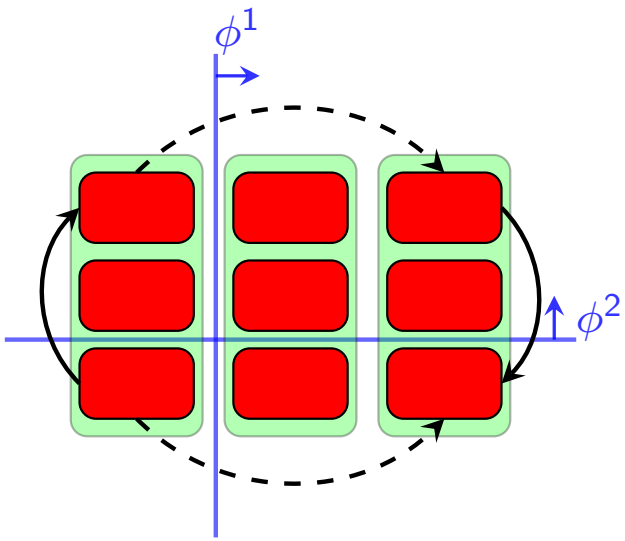
Q_e^k is a subset of P_e^k restricted to the same $k-1$ dimensional tile defined by the $k-1$ tiling hyperplanes outer to ϕ^k.



Localizing Dependences

Restricted Tile Dependence

Q_e^k is a subset of P_e^k restricted to the same $k-1$ dimensional tile defined by the $k-1$ tiling hyperplanes outer to ϕ^k.



Localizing Dependences

Restricted Tile Dependence

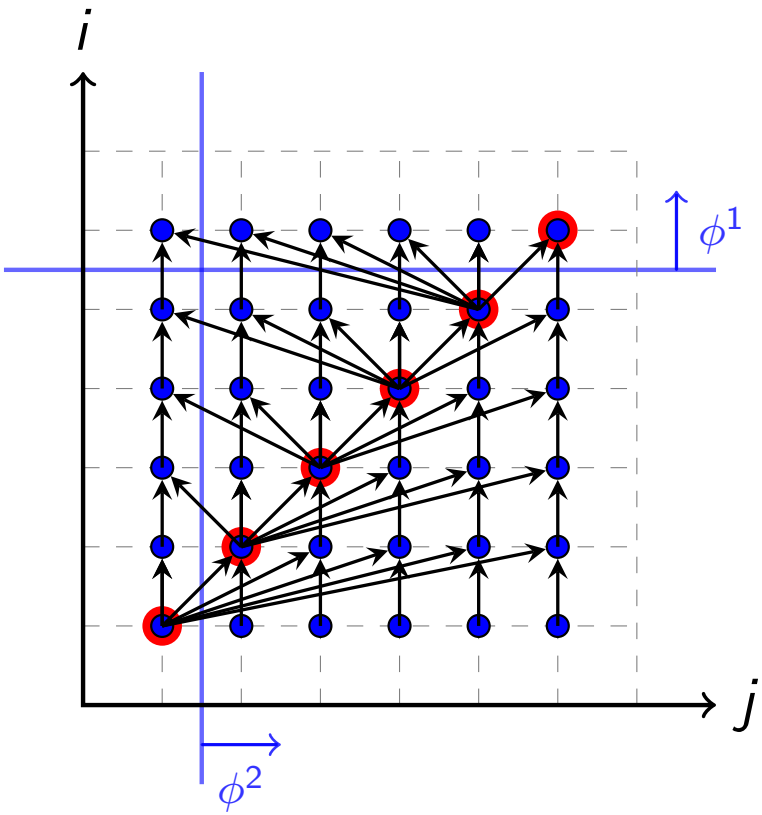
Q_e^k is a subset of P_e^k restricted to the same $k-1$ dimensional tile defined by the $k-1$ tiling hyperplanes outer to ϕ^k.

- Q_e^k thus captures dependences between only those k-dimensional tiles which are in the same $k-1$ dimensional tile formed by $\langle \phi^1, \ldots, \phi^{k-1} \rangle$

If e is from statement s to statement t, then:

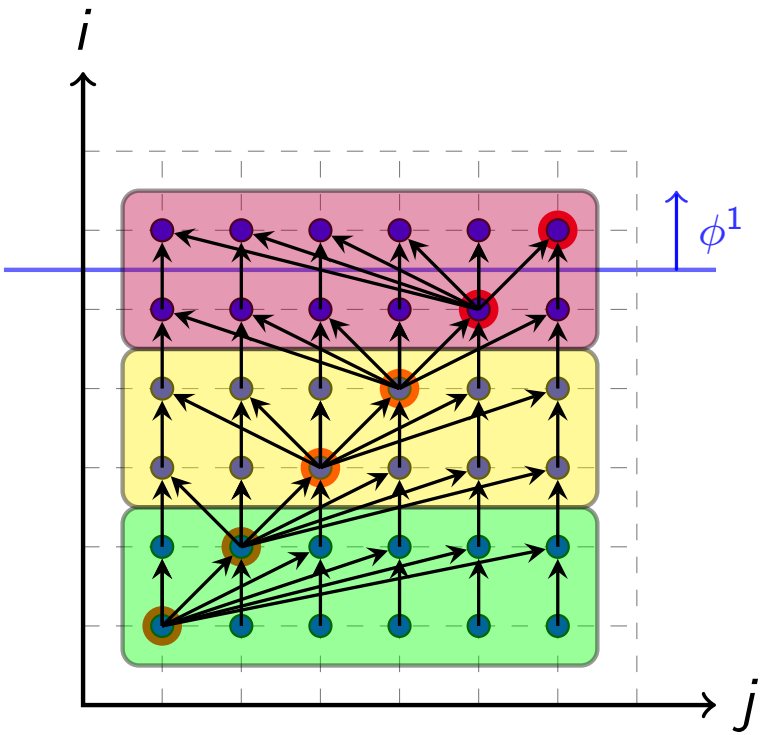
$$Q_e^k = P_e^k \wedge \left(\bigwedge_{1 \leq l \leq k-1} T_s^l = T_t^l \right)$$

Improved Iterative Tiling



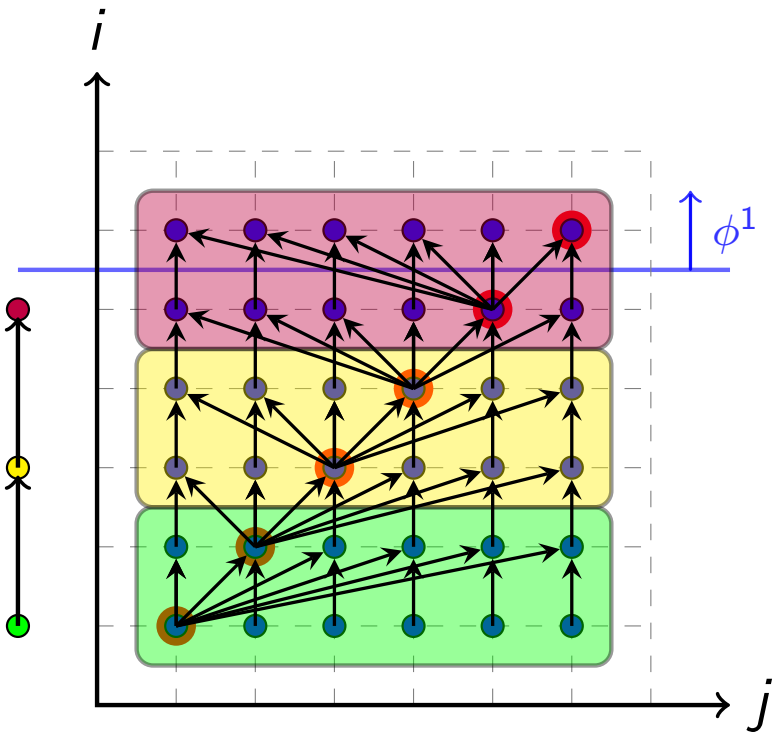

```
1 while τ_k > 1 do
    // 1. Check validity of tiling at level k.
    for each e ∈ E do
      Compute Q_e^k for dependence e: s → t.
    // C is the set of tiles that might be in a
       cycle
    C = CycleCheck(Q_e^k for each edge e ∈ E)
    if C = ∅ then
      // Tiling is valid, move to next level
      break
    // 2. Attempt to correct tiling
    τ_k = ⌊τ_k/2⌋
```

Improved Iterative Tiling



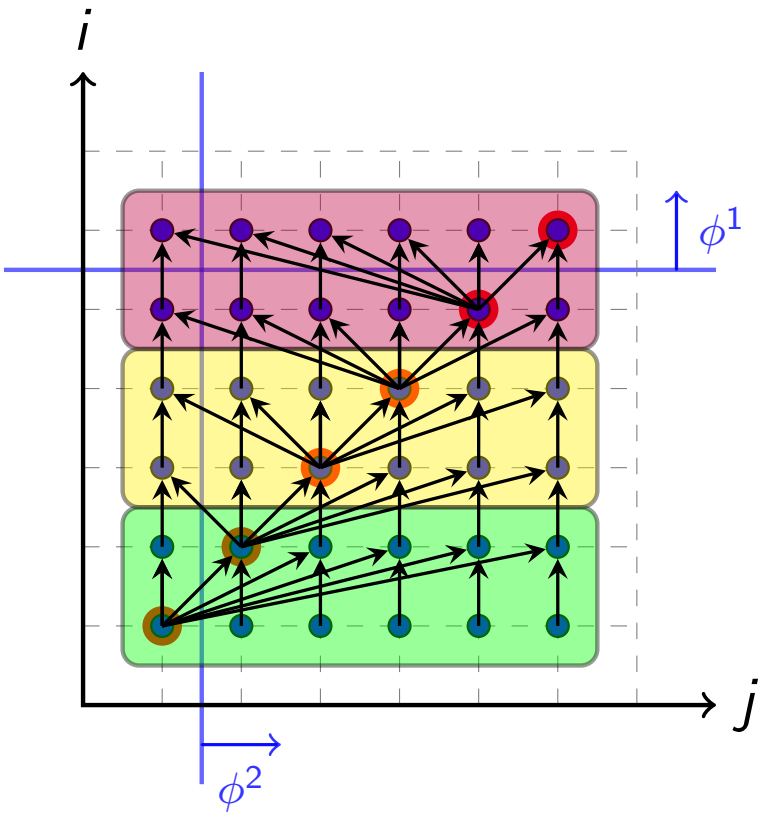

```
1 while τ_k > 1 do
    // 1. Check validity of tiling at level k.
    for each e ∈ E do
      Compute Q_e^k for dependence e: s → t.
    // C is the set of tiles that might be in a
       cycle
    C = CycleCheck(Q_e^k for each edge e ∈ E)
    if C = ∅ then
      // Tiling is valid, move to next level
      break
    // 2. Attempt to correct tiling
    τ_k = ⌊τ_k/2⌋
```

Improved Iterative Tiling



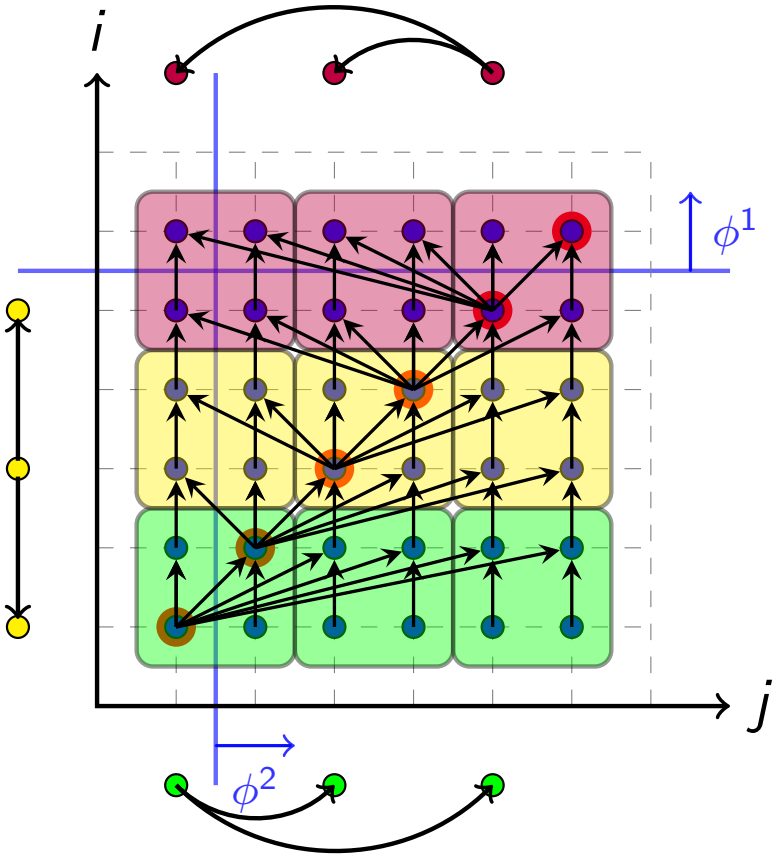

```
1 while τ_k > 1 do
    // 1. Check validity of tiling at level k.
    for each e ∈ E do
      Compute Q_e^k for dependence e: s → t.
    // C is the set of tiles that might be in a
       cycle
    C = CycleCheck(Q_e^k for each edge e ∈ E)
    if C = ∅ then
      // Tiling is valid, move to next level
      break
    // 2. Attempt to correct tiling
    τ_k = ⌊τ_k/2⌋
```

Improved Iterative Tiling




```
1 while τ_k > 1 do
    // 1. Check validity of tiling at level k.
    for each e ∈ E do
      Compute Q_e^k for dependence e: s → t.
    // C is the set of tiles that might be in a
       cycle
    C = CycleCheck(Q_e^k for each edge e ∈ E)
    if C = ∅ then
      // Tiling is valid, move to next level
      break
    // 2. Attempt to correct tiling
    τ_k = ⌊τ_k/2⌋
```

Improved Iterative Tiling

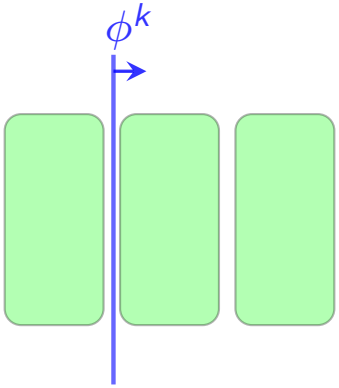


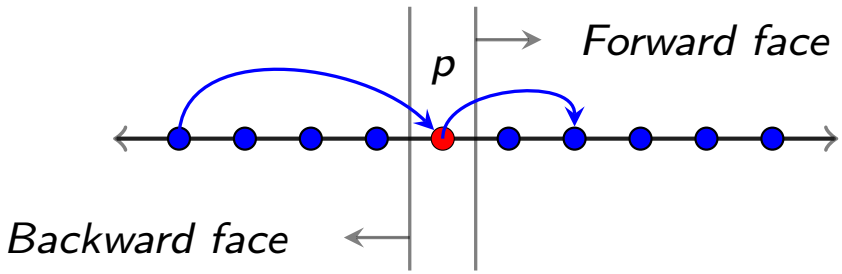

```
1 while τ_k > 1 do
    // 1. Check validity of tiling at level k.
    for each e ∈ E do
      Compute Q_e^k for dependence e: s → t.
    // C is the set of tiles that might be in a
       cycle
    C = CycleCheck(Q_e^k for each edge e ∈ E)
    if C = ∅ then
      // Tiling is valid, move to next level
      break
    // 2. Attempt to correct tiling
    τ_k = ⌊τ_k/2⌋
```

Approximate Cycle Check

Dependences On a Line

Each hyperplane ϕ^k gives a one-dimensional coordinate for a tile which can be used to map a tile to a point on a line.



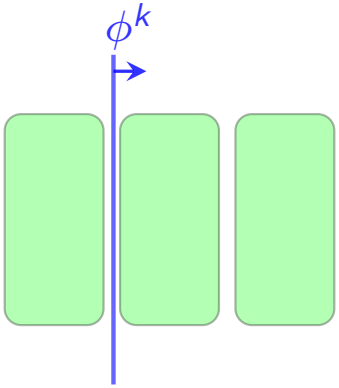


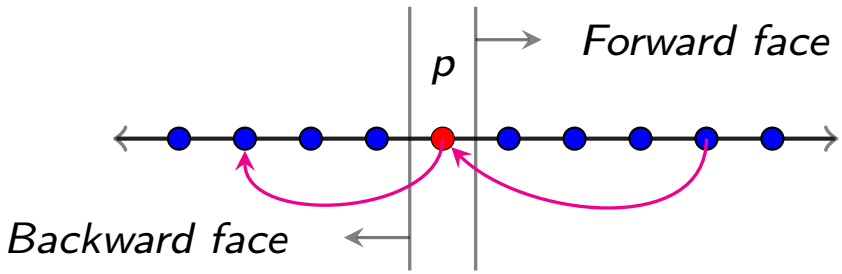
- Forward Dependences
- Backward Dependences
- When are there no cycles?

Approximate Cycle Check

Dependences On a Line

Each hyperplane ϕ^k gives a one-dimensional coordinate for a tile which can be used to map a tile to a point on a line.



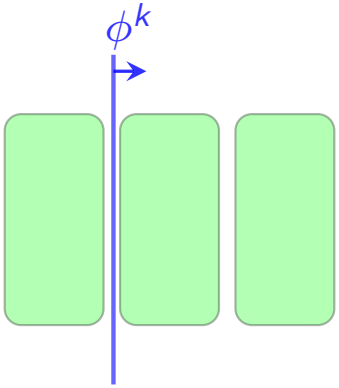


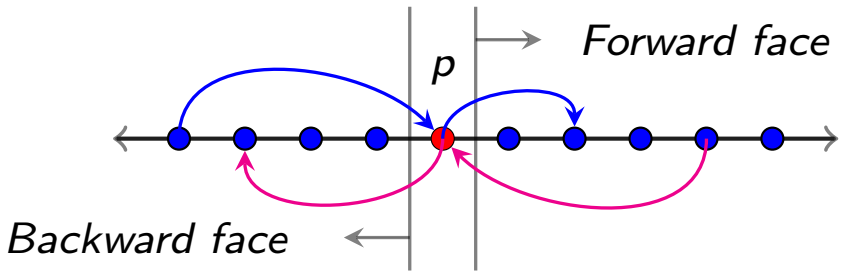
- Forward Dependences
- Backward Dependences
- When are there no cycles?

Approximate Cycle Check

Dependences On a Line

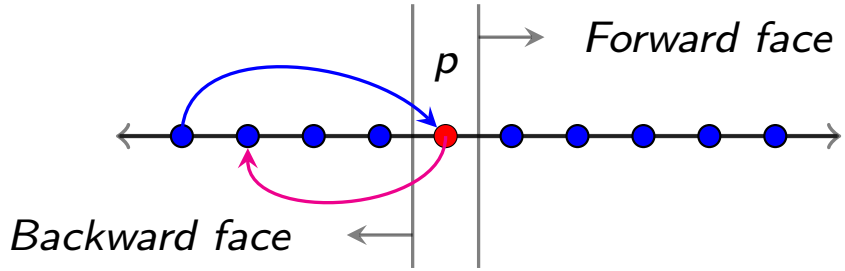
Each hyperplane ϕ^k gives a one-dimensional coordinate for a tile which can be used to map a tile to a point on a line.





- Forward Dependences
- Backward Dependences
- When are there no cycles?

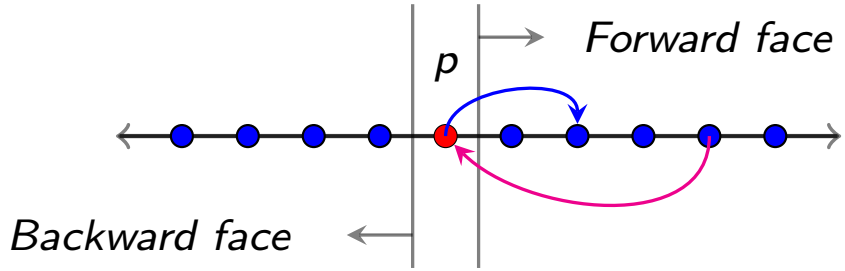
Approximate Cycle Check



Backward Violation

$$B^k_{e,e'} = \Big\{ \langle T^1_s, \ldots, T^k_s \rangle \mid \exists T^l_t, \exists T^l_{s'}, \exists T^l_{t'}, (T^k_s \geq T^k_t + 1) \wedge$$
$$(T^k_{s'} \leq T^k_{t'} - 1) \wedge_{1 \leq l \leq k} (T^l_{t'} = T^l_s) \wedge$$
$$\langle \langle T^1_s, \ldots, T^k_s \rangle, \langle T^1_t, \ldots, T^k_t \rangle \rangle \in Q^k_e, e : s \rightarrow t$$
$$\langle \langle T^1_{s'}, \ldots, T^k_{s'} \rangle, \langle T^1_{t'}, \ldots, T^k_{t'} \rangle \rangle \in Q^k_{e'}, e' : s' \rightarrow t' \Big\}$$

Approximate Cycle Check



Forward Violation

$$F^k_{e,e'} = \Big\{ \langle T^1_s, \ldots, T^k_s \rangle \mid \exists T^l_t, \exists T^l_{s'}, \exists T^l_{t'}, (T^k_s \leq T^k_t - 1) \wedge$$
$$(T^k_{s'} \geq T^k_{t'} + 1) \wedge_{1 \leq l \leq k} (T^l_{t'} = T^l_s) \wedge$$
$$\langle \langle T^1_s, \ldots, T^k_s \rangle, \langle T^1_t, \ldots, T^k_t \rangle \rangle \in Q^k_e, e : s \rightarrow t$$
$$\langle \langle T^1_{s'}, \ldots, T^k_{s'} \rangle, \langle T^1_{t'}, \ldots, T^k_{t'} \rangle \rangle \in Q^k_{e'}, e' : s' \rightarrow t' \Big\}$$

Approximate Cycle Check

Input : Tile dependence polyhedra at level k Q_e^k for all $e \in E$ the set of edges in the GDG.

Output: Set of tiles that might be part of a cycle

// B^k set of tiles that satisfy B at level k.

// F^k set of tiles that satisfy F at level k.

$B^k = \emptyset, F^k = \emptyset$

for each *pair* $\langle Q_e^k, Q_{e'}^k \rangle$ $e, e' \in E$ // e can be equal to e'

do

  Compute $B_{e,e'}^k$, $F_{e,e'}^k$ using Q_e^k and $Q_{e'}^k$

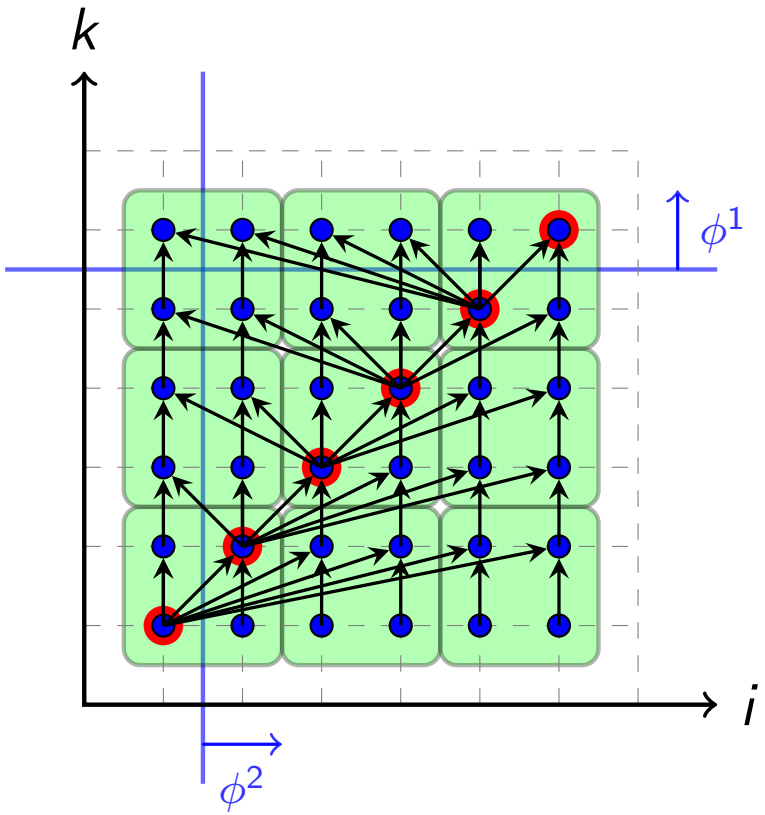
  $B^k = B_{e,e'}^k \cup B^k$

  $F^k = F_{e,e'}^k \cup F^k$

return $B^k \cup F^k$

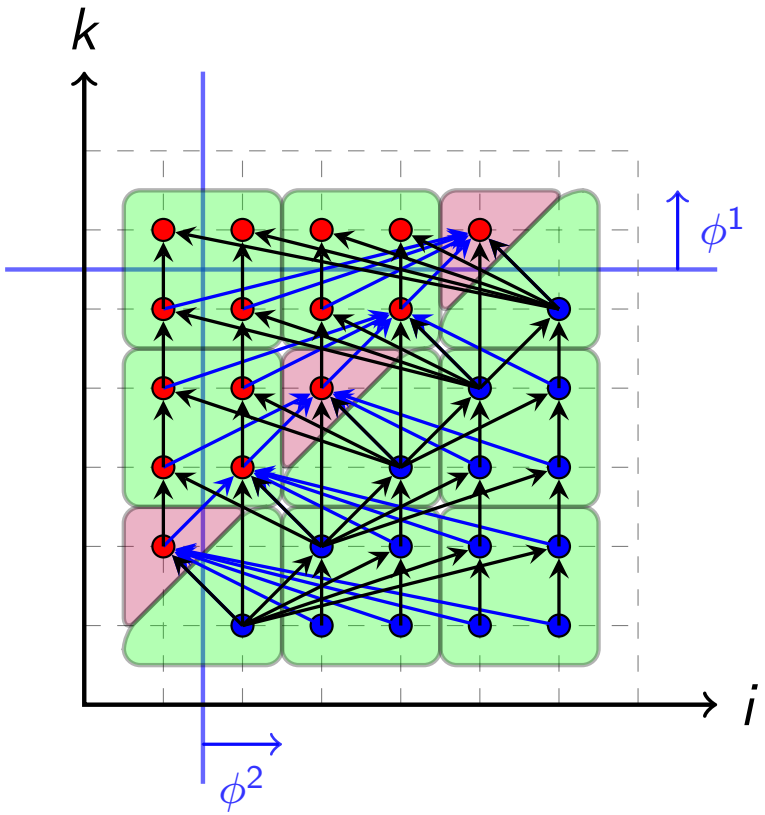
Table of Contents

Floyd-Warshall All Pairs Shortest Paths



- Using additional arrays R and C to remove false dependences in the All-Pairs Shortest-Paths kernel

Floyd-Warshall All Pairs Shortest Paths



- All-Pairs Shortest-Paths kernel after removing spurious writes

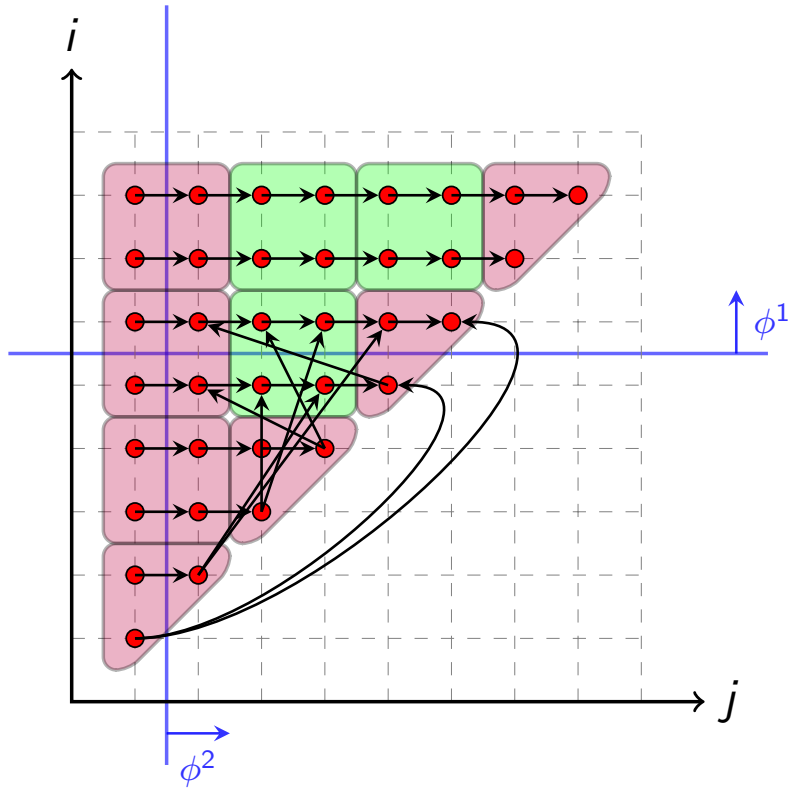
Zuker's RNA Secondary Structure Prediction

```
for (i = N-1; i >= 0; i--) {
    for (j = i+1; j < N; j++) {
        for (k = 0; k < j-i; k++) {
            S[i][j] = MAX(S[i][k+i] + S[k+i+1][j], S[i][j]);
        }
        S[i][j] = MAX(S[i][j], S[i+1][j-1] +
                              can_pair(RNA[i],RNA[j]));
    }
}
```

- Complex dynamic programming recurrence
- 3-d tiling the $\mathcal{O}(n^3)$ loop nest

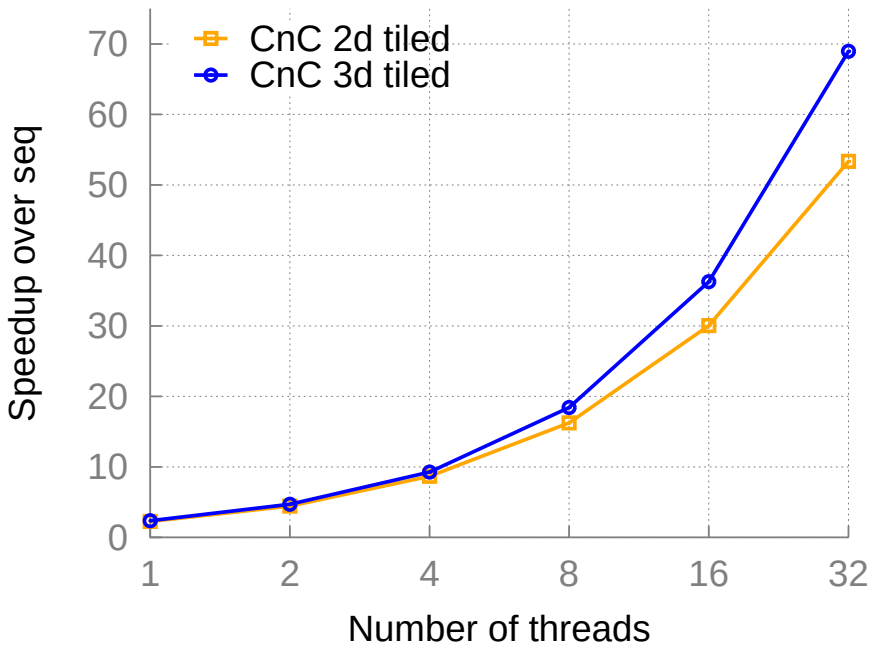
Zuker's RNA Secondary Structure Prediction

```
for (i = 0; i < N; i++){
    for ( j = 0; j < i+1; j++) {
        A[i] = A[i] + A[i-j]; s1
    }
}
```

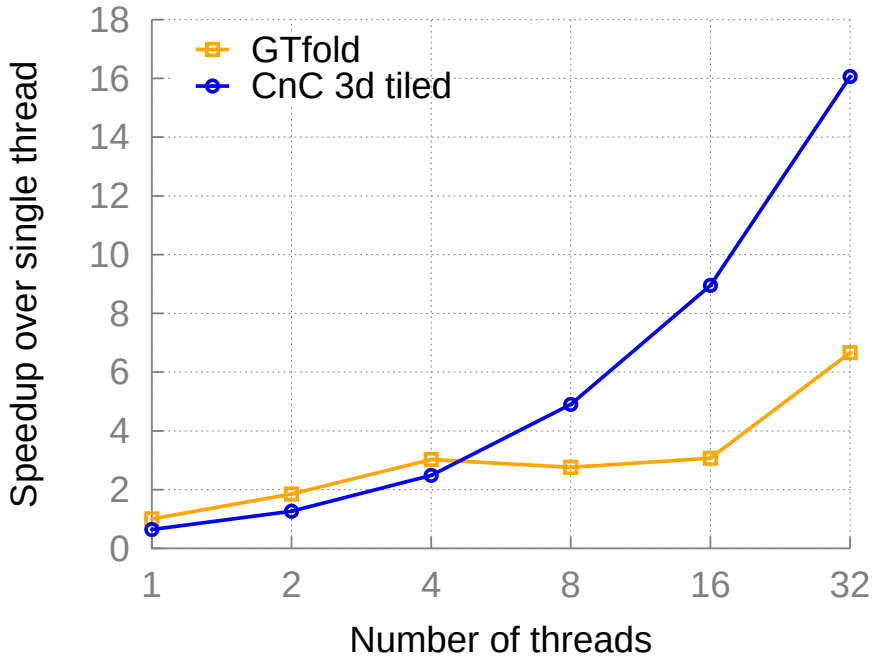


- Merging tiles
- Use commutative properties

Experimental Evalutation



(a) `floyd` – *seq* time is 231s



(b) `zuker` – *single thread* time is 253s

- Experimental setup is a four socket machine with an AMD Opteron 6136 (2.4 GHz, 128 KB L1, 512 KB L2, 6 MB L3 cache) in each socket.

Conclusions and Future Work

- Using improved validity constraints for hyperplane search
- Integrating splitting techniques
- More accurate cycle detection

Questions?

Thank You!