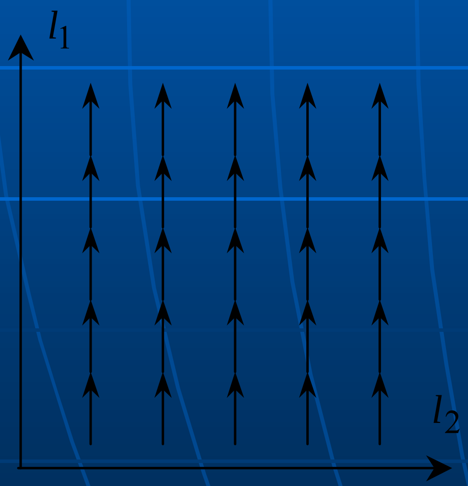
Anna Beletska

anna.beletska@inria.fr

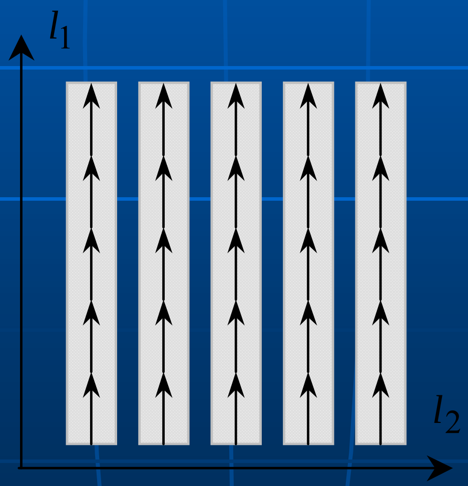
Extracting coarse-grained parallelism in arbitrarily nested loops

Coarse-grained parallelism

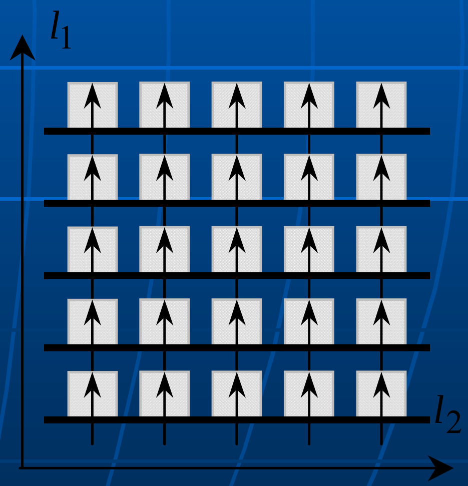
Coarse-grained parallelism is employed by creating a thread on each processor, executing in parallel for a period of time with occasional synchronisation.



Iteration space and data dependences



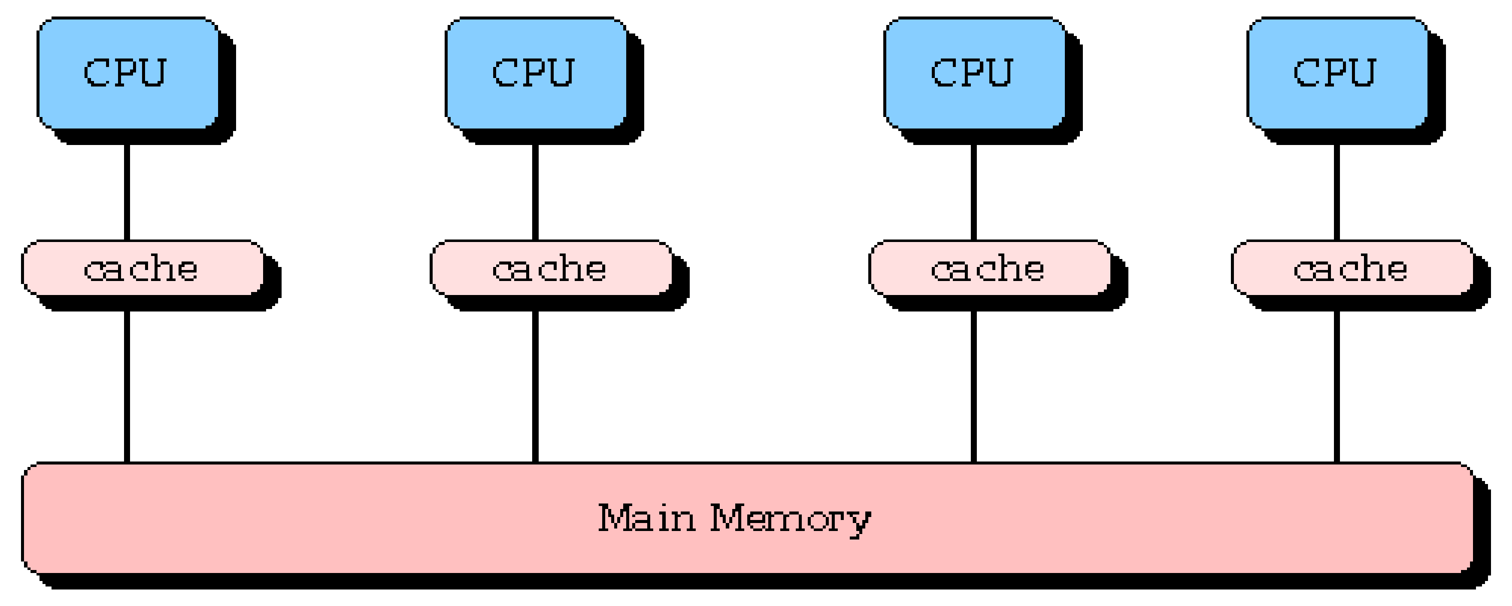
Coarse-grained scheme



Fine-grained scheme

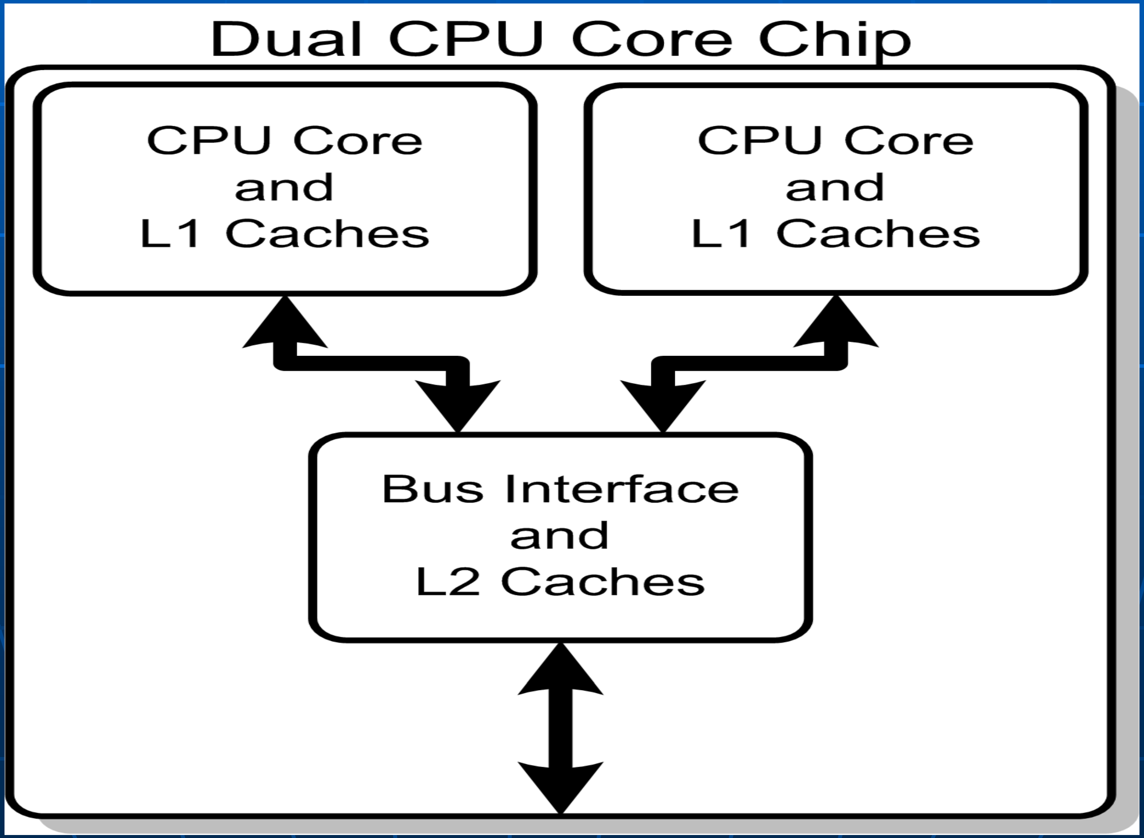
Coarse-grained parallelism

- Provides high performance on multiprocessors



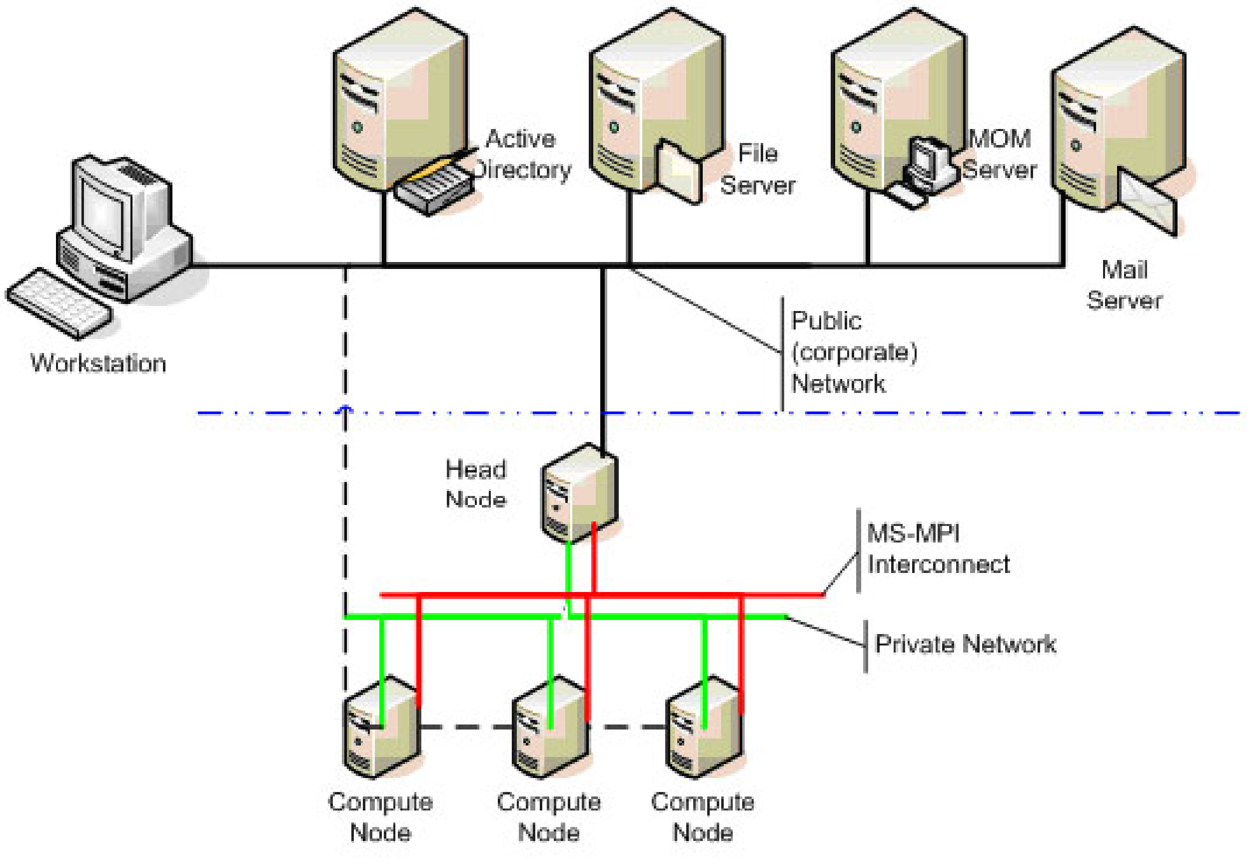
Coarse-grained parallelism

- Increases performance on computers with dual CPU core chips



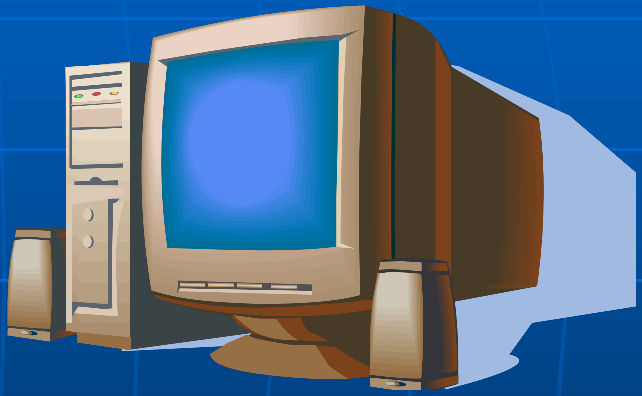
Coarse-grained parallelism

- Increases performance of distributed systems



Coarse-grained parallelism

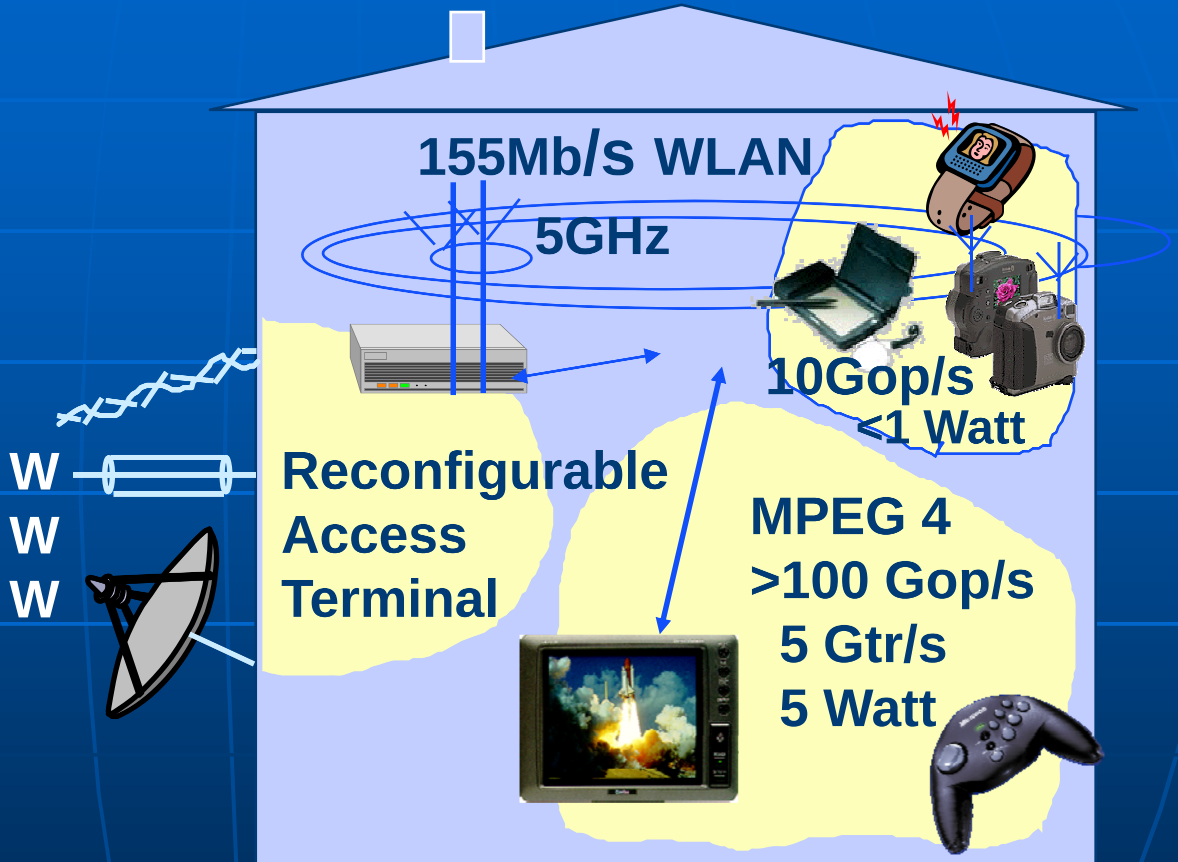
- Enhances performance of uniprocessors



  - Improves code locality
  - Decreases memory requirements

Coarse-grained parallelism

Intelligent home



It can be used in embedded systems decreasing cost and power consumption!

Approaches to extract CGP

- Unimodular transforms[1]
  - Can be applied only to perfectly-nested uniform loops

---

[1] Banerjee U. Unimodular transformations of double loops. In *Proceedings of the Third Workshop on Languages and Compilers for Parallel Computing*. (1990) pp. 192-219

[1] Wolf M.E. *Improving locality and parallelism in nested loops*. Ph.D. Dissertation CSL-TR-92-538, Stanford University, Dept. Computer Science. (1992)

Approaches to extract CGP

- Approach based on the Hamiltonian recurrences [2]
  - Is applicable only to uniform non-parameterized loops

[2] Gavaldà R.,Ayguade E., Torres J. *Obtaining Synchronization-Free Code with Maximum Parallelism*. Technical Report LSI-96-23-R, Universitat Politècnica de Catalunya. (1996)

Approaches to extract CGP

- Procedures of heuristic searches[3]
  - do not guarantee extracting the entire coarse-grained parallelism available in non-uniform loops

---

[3] W. Kelly, W. Pugh, Minimizing communication while preserving parallelism, in: Proceedings of the 1996 ACM International Conference on Supercomputing. (1996) 52-60

Approaches to extract CGP

- Affine transformation framework[4]

[4] Feautrier P. Some efficient solutions to the affine scheduling problem, part i, one dimensional time. *International Journal of Parallel Programming 21*. (1992), pp. 313-348

[4] Lim W., Cheong G.I., Lam M.S. An affine partitioning algorithm to maximize parallelism and minimize communication. In *Proceedings of the 13th ACM SIGARCH International Conference on Supercomputing*. (1999)

[4] Darte A., Robert Y., Vivien F. *Scheduling and Automatic Parallelization*. Birkhäuser Boston. (2000)

[4] Bastoul C., Cohen A., Girbal S., Sharma S., and Temam O. Putting polyhedral loop transformations to work. In *Languages and Compilers for Parallel Computing (LCPC'03)*. LNCS, pp 23--30, College Station, Texas, Springer-Verlag (2003).

Approaches to extract CGP

- Slicing framework[5]

---

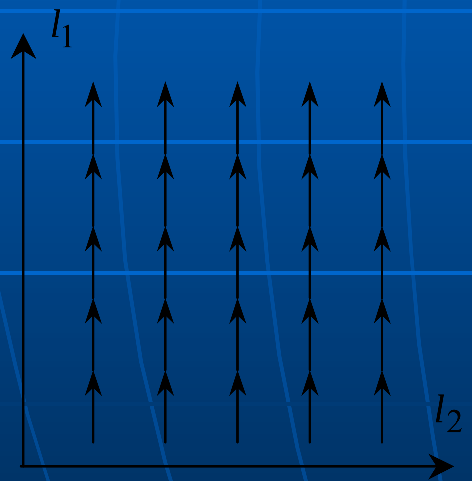
[5] Weiser M.. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, Ann Arbor, MI. (1979)

[5] Weiser M. Program Slicing. *IEEE Transactions on Software Engineering, v. SE-10, no. 7*. (1984), pp 352-357.

[5] Pugh W. , Rosser E. Iteration Space Slicing and Its Application to Communication Optimization In *Proceedings of the International Conference on Supercomputing*. (1997), pp 221-228

Data dependences

Definition 1. ***A dependence relation*** is a mapping from one iteration space to another, and is represented by a set of linear constraints on variables that stand for the values of the loop indices at the source and destination of the dependence and the values of the symbolic constants[6].



Iteration space and data dependences

$$\overbrace{\{\ \underbrace{[i,j]}_{\text{iteration space}} : \underbrace{1 \le i,j \le 5}_{\text{Presburger formula}}\ \}}^{\text{set}}$$

Iteration space

$$\overbrace{\{\ \underbrace{[i,j]}_{\text{source}} \text{ -> } \underbrace{[i,j+1]}_{\text{destination}} : \underbrace{1 \le i < j \le 5}_{\text{Presburger formula}}\ \}}^{\text{relation}}$$

Data dependences

---

[6] Pugh, W., Wonnacott D.: An Exact Method for Analysis of Value-based Array Data Dependences. Workshop on Languages and Compilers for Parallel Computing, 1993

Dependence analysis

Our approaches require an ***exact dependence analysis*** which detects a dependence if and only if it exists.

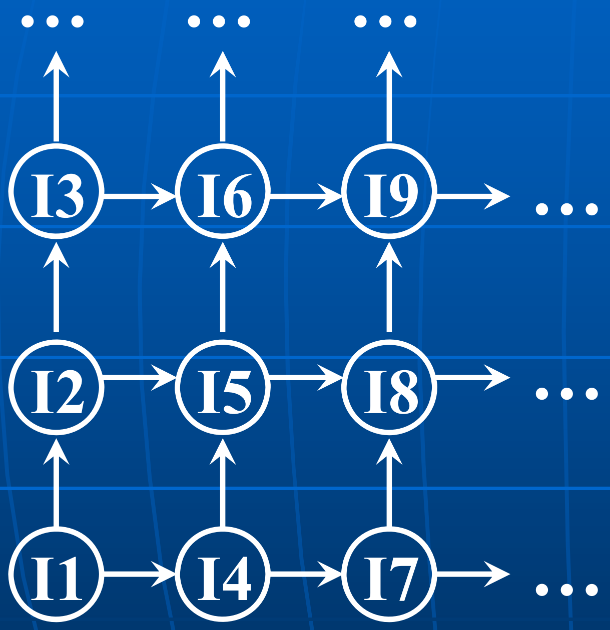
The dependence analysis by ***Pugh and Wonnacott*** was chosen where dependences are found in the form of tuple relations[7] .

---

[7] Pugh W., Wonnacott D. Constraint-based array dependence analysis. In *ACM Trans. on Programming Languages and Systems*. (1998)

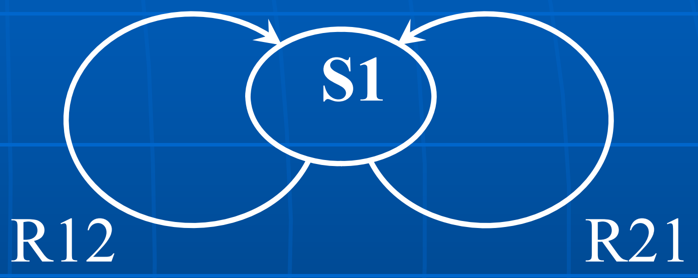
Dependence graphs

Dependence Graph



represents all the dependences among iterations available in a loop

Reduced Dependence Graph



is composed of vertices for each statement of the loop and edges joining vertices according to dependence relations

Strongly Connected Components

- ***Strongly connected component*** is a maximal subset of vertices and edges of a reduced dependence graph where for every pair of vertices there exists a direct path.

This graph has two strongly connected components given by {S1, S2} and {S3}, respectively.

Affine transformation framework

The Affine Transformation Framework[4] is considered in many works and unifies a large number of previously proposed loop transformations.

Today, it is one of the most powerful frameworks for loop transformations allowing us to extract coarse-grained parallelism presented in arbitrarily nested uniform loops and in some cases of non-uniform loops.

[4] Feautrier P. Some efficient solutions to the affine scheduling problem, part i, one dimensional time. *International Journal of Parallel Programming 21*. (1992), pp. 313-348

[4] Lim W., Cheong G.I., Lam M.S. An affine partitioning algorithm to maximize parallelism and minimize communication. In *Proceedings of the 13th ACM SIGARCH International Conference on Supercomputing*. (1999)

[4] Darte A., Robert Y., Vivien F. *Scheduling and Automatic Parallelization*. Birkhäuser Boston. (2000)

[4] Bastoul C., Cohen A., Girbal S., Sharma S., and Temam O. Putting polyhedral loop transformations to work. In *Languages and Compilers for Parallel Computing (LCPC'03)*. LNCS, pp 23--30, College Station, Texas, Springer-Verlag (2003).

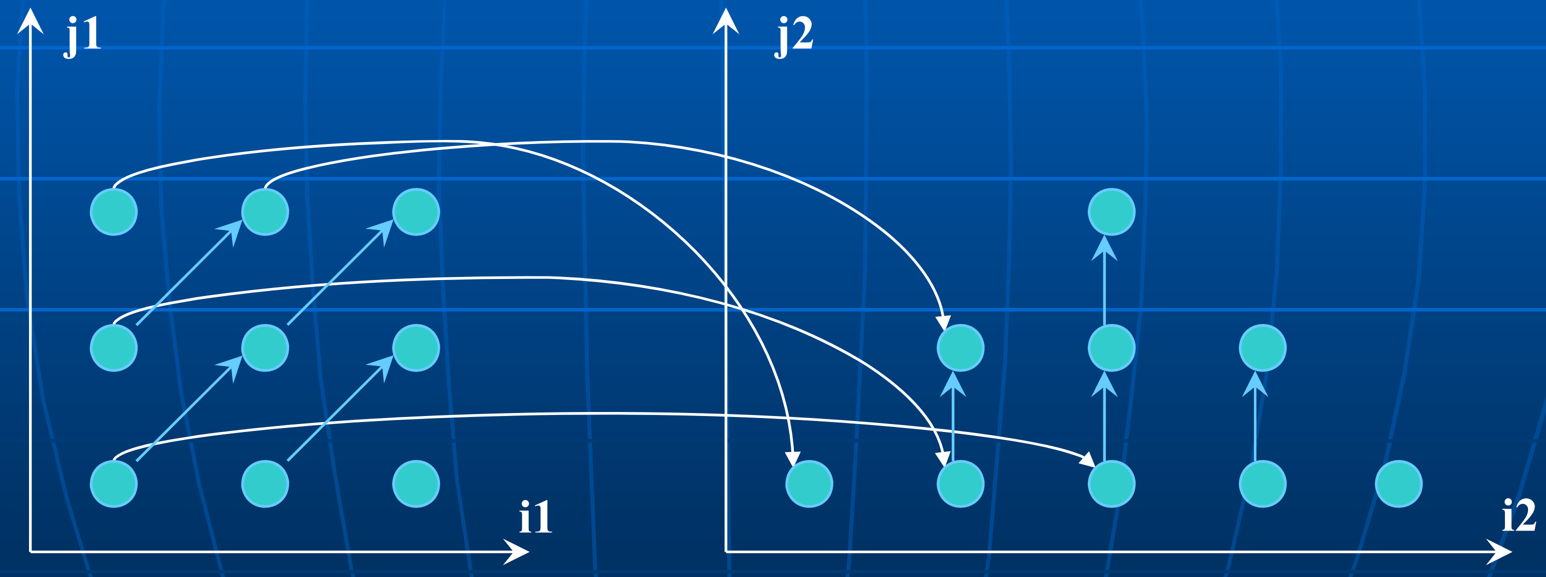
Affine transformation framework

Instances of each instruction are identified by the loop index values of their surrounding loops, and affine expressions are used to map these loops index values to a partition number:

- ***Space partition (Affine mapping):*** operations belonging to the same space partition are mapped to the same processor.
- ***Time partition (Affine scheduling):*** operations belonging to time partition i are executed before those in partition i+1.

Affine transformation framework

The operations of a loop are divided into partitions such that dependent operations are placed in the same partition.



A partitioning is described by an affine mapping for each loop statement.

ATF Algorithm

BEGIN

Find all dependences

Form the reduced dependence graph

Find all strongly connected components (SCC)

Find affine transforms for each SCC

Generate parallel code taking into account the order of the SCCs execution

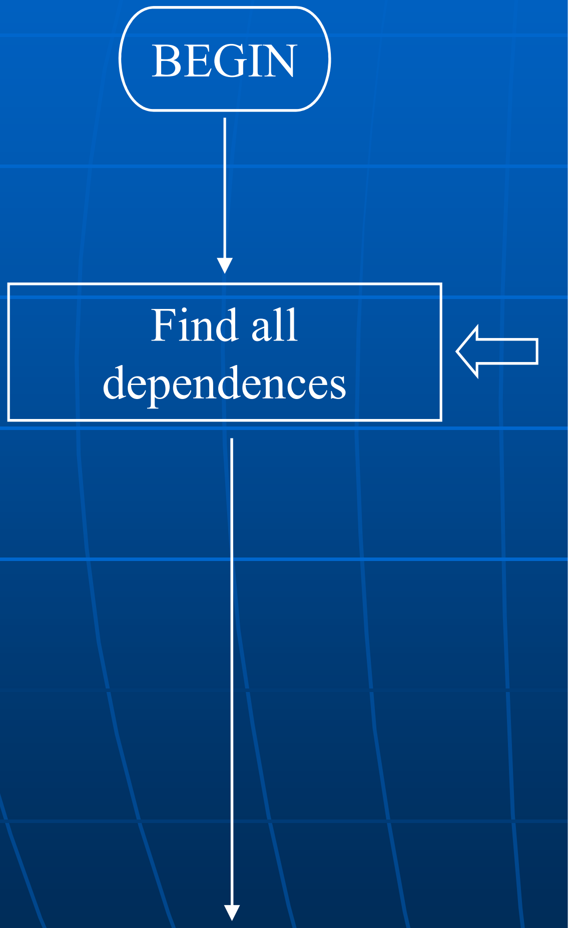
END

Tools

- Petit[9] : a research tool for performing dependence analysis and program transformations.
- Omega Calculator[9]: a research tool for Presburger arithmetics, including solving linear systems of equalities and code generation.

[9] http://www.cs.umd.edu/projects/omega/

Example of parallelization by ATF



BEGIN

Find all dependences

For the following loop:

```
for i=1 to m do
  for j=1 to m do
  3:  a(i,j)=a(i,j-1)
  4:  b(i,j)=b(i-1,j)
  5:  c(i,j)=c(i,j)+a(i,j-1)*b(i-1,j)
  endfor
endfor
```

We get the information about dependences:

```
flow    3: a(i,j)         -->  3: a(i,j-1)
{[i,j] -> [i,j+1] : 1 <= i <= m && 1 <= j < m}
flow    3: a(i,j)         -->  5: a(i,j-1)
{[i,j] -> [i,j+1] : 1 <= i <= m && 1 <= j < m}
flow    4: b(i,j)         -->  4: b(i-1,j)
{[i,j] -> [i+1,j] : 1 <= i < m && 1 <= j <= m}
flow    4: b(i,j)         -->  5: b(i-1,j)
{[i,j] -> [i+1,j] : 1 <= i < m && 1 <= j <= m}
```

Example of parallelization by ATF

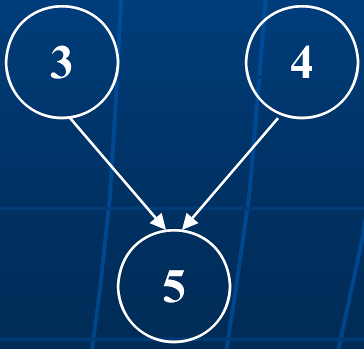
Form the reduced dependence graph

Find all SCCs

According to the information

flow 3: a(i,j) --> 3: a(i,j-1)
{[i,j] -> [i,j+1] : 1 <= i <= m && 1 <= j < m}
flow 3: a(i,j) --> 5: a(i,j-1)
{[i,j] -> [i,j+1] : 1 <= i <= m && 1 <= j < m}
flow 4: b(i,j) --> 4: b(i-1,j)
{[i,j] -> [i+1,j] : 1 <= i < m && 1 <= j <= m}
flow 4: b(i,j) --> 5: b(i-1,j)
{[i,j] -> [i+1,j] : 1 <= i < m && 1 <= j <= m}

we construct the following reduced dependence graph



The graph contains three SCCs, given by instruction 3, 4 i 5.

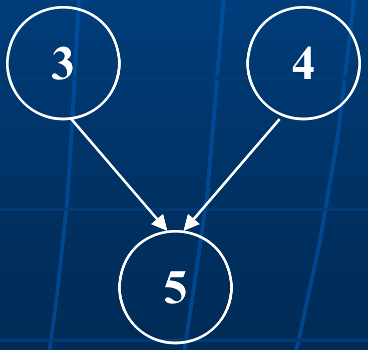
Example of parallelization by ATF

Find the affine transforms for each SCC

1. For each SCC, form a set of the dependence relations and construct the system of linear equations.

2. Find the solution of the system.

Generate parallel code taking into account the order of the SCCs execution



In this graph SCCs 3 and 4 can be executed in parallel, while 5 can be executed only after executing SCCs 3 and 4.

Example of parallelization by ATF

The generated parallel code:

```
#parallel
{
  #independent
     parfor (i  =  1 ;  i <=  m ;  i++  )
         for (j  =  1 ;  j <=  m ;  j++  )
             a (i,j) = a (i,j-1);
  #independent
     parfor (i  =  1 ;  i <=  m ;  i++  )
         for (j  = 1 ; j <=  m ;  j++  )
             b (j,i ) = b (j-1, i );
}

parfor (i=1; i<=m; i+=1)
    parfor (j=1; j<=m; j+=1)
         c(i,j)=c(i,j)+a(i,j-1)*b(i-1,j)
```

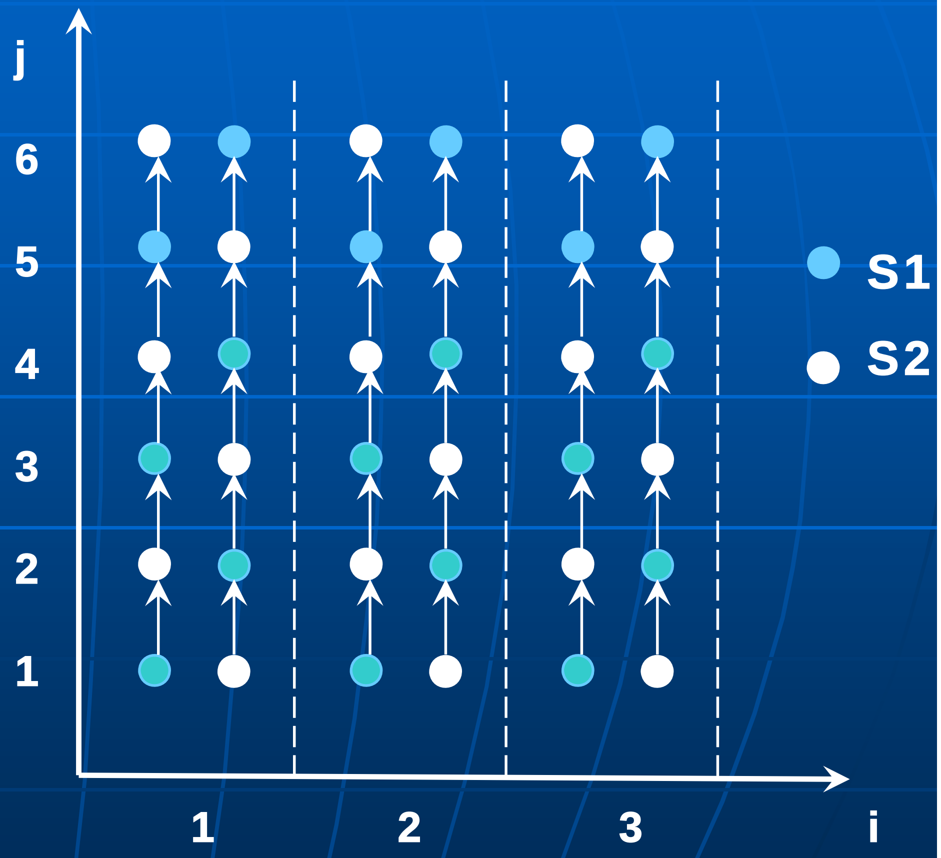
Pragma #parallel contains SCCs which are within pragmas #independent and which can be executed in parallel

The keyword „*parfor*” defines loops whose iterations can be executed in parallel.

Limitations of ATF

- It fails to extract *all synchronization-free slices* available in a loop

```
for i=1 to n do
  for j=1 to m do
s1:  a(i,j)=b(i,j)+c(i,j)
s2:  c(i,j-1)=a(i,j+1)
```



$R1=\{[i,j] \rightarrow [i,j+1] : 1 \le i \le n \ \&\& \ 1 \le j < m\}$

$R2=\{[i,j] \rightarrow [i,j+1] : 1 \le i \le n \ \&\& \ 1 \le j < m\}$

Limitations of ATF

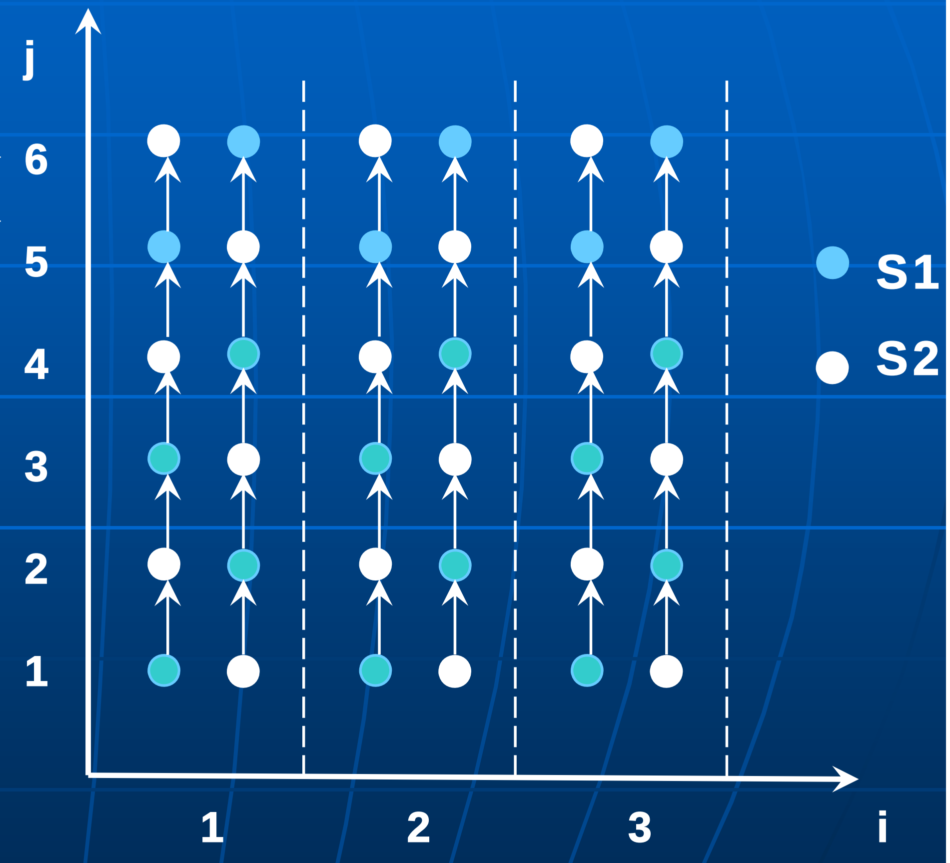
- It fails to extract *all synchronization-free slices* available in a loop

$$R1=\{[i,j] \rightarrow [i,j+1] : 1 \leq i \leq n \;\&\&\; 1 \leq j < m\}$$
$$R2=\{[i,j] \rightarrow [i,j+1] : 1 \leq i \leq n \;\&\&\; 1 \leq j < m\}$$

$$\begin{cases} C11*i+C12*j+C1=C21*i+C22*j+C22+C2 \\ C21*i+C22*j+C2=C11*i+C12*j+C12+C1 \end{cases}$$

$$\Downarrow$$

$$\begin{cases} C11 = C21 = \text{arbitrary value, let it be } n1,\ n1 \geq 0. \\ C12 = C22 = 0 \end{cases}$$



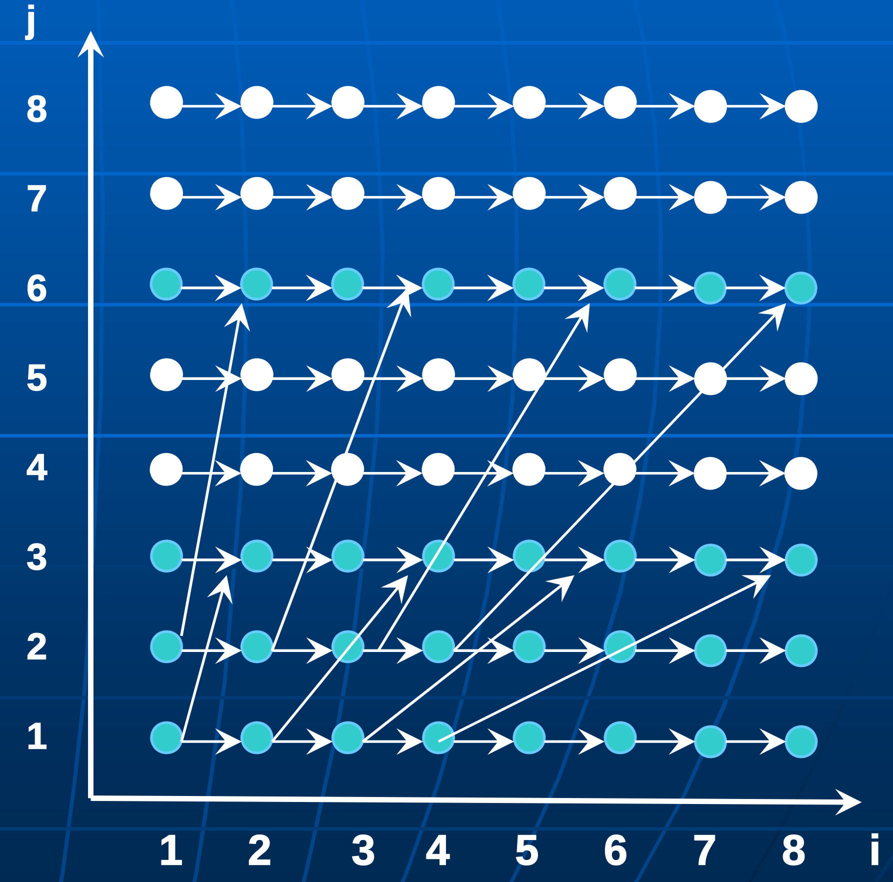
Limitations of ATF

- It fails to extract coarse-grained parallelism available in a *subspace of the loop domain*

```
for i = 1 to n do
    for j = 1 to n do
        a(2*i, 3*j) = b(i,j)
        b(i+1, j) = a(i, j)
```

$R1 = \{[i,j] \rightarrow [2i,3j]: 1 \le j \ \&\ 2i \le n \ \&$
$1 \le i \ \&\ 3j \le n \ \}$

$R2 = \{[i,j] \rightarrow [i+1,j]: 1 \le j < n \ \&\ 1 \le j \le n \ \}$



Limitations of ATF

- It fails to extract coarse-grained parallelism available in a *subspace of the loop domain*

R1 = {[i,j] → [2i,3j]: 1≤j & 2i ≤ n &
1≤i & 3j≤n }

R2 = {[i,j] → [i+1,j]: 1≤j<n & 1≤j≤n }

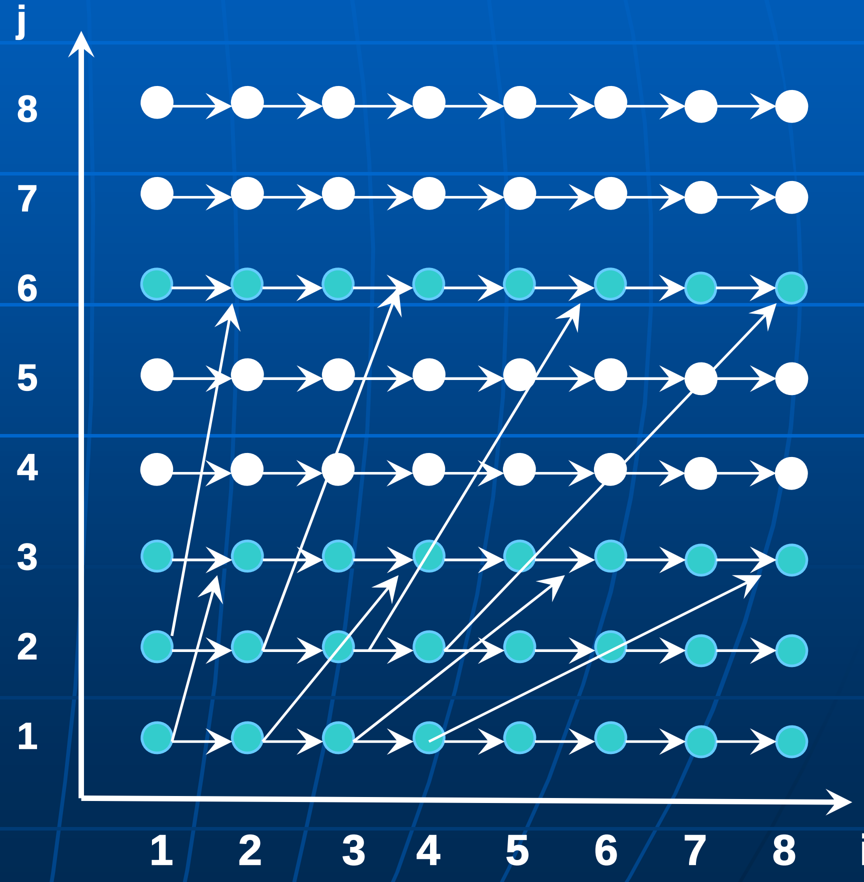
$$\begin{cases} C2*i+C1*j+C0=C2*2i+C1*3j+C0 \\ C2*i+C1*j+C0=C1*(i+1)+C1*j+C0 \end{cases}$$

$$\Downarrow$$

$$\begin{cases} 0=C2*i+C1*2j \\ 0=C1 \end{cases}$$

$$\Downarrow$$

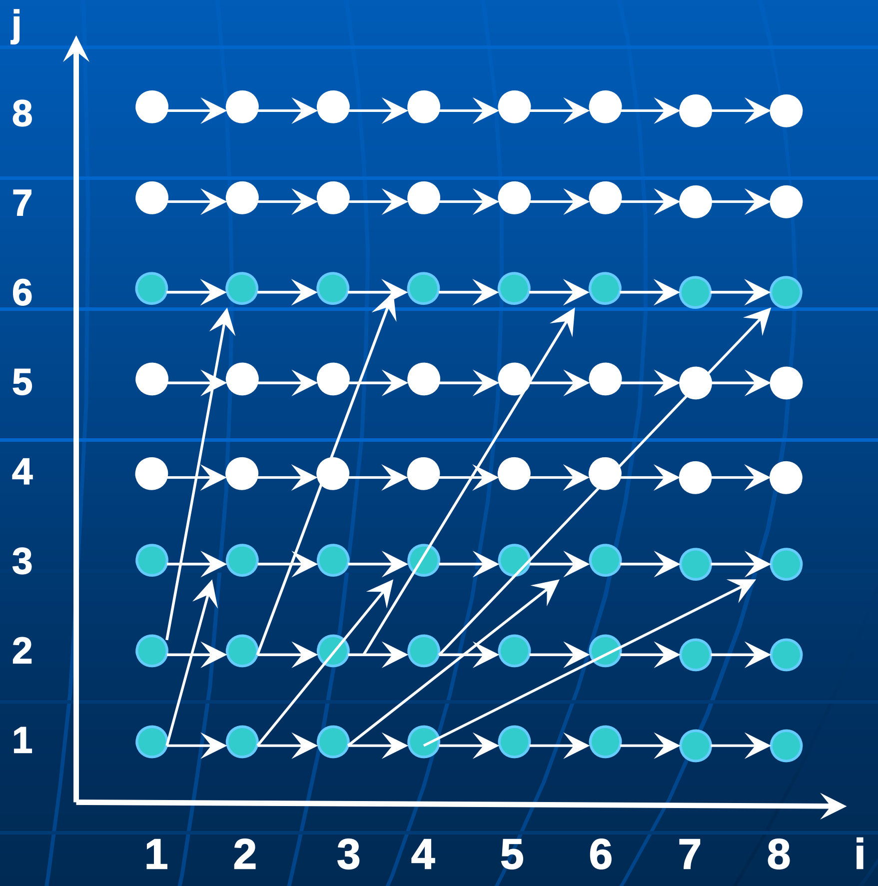
$$\begin{cases} C1=0 \\ C2=0 \end{cases}$$



Limitations of ATF

- It fails to extract coarse-grained parallelism in the general case of *non-uniform loops*

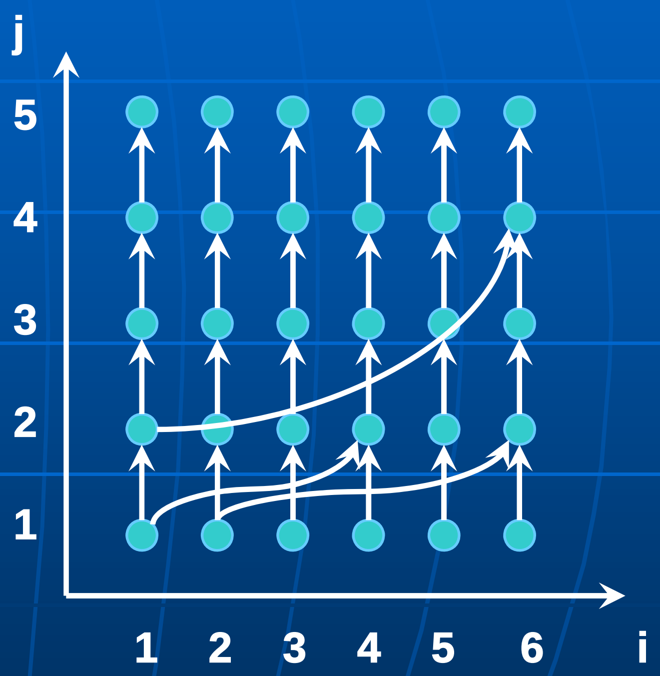
```
for i = 1 to n do
    for j = 1 to n do
        a(2*i, 3*j) = b(i,j)
        b(i+1, j) = a(i, j)
```



Limitations of ATF

- It fails to extract threads when *synchronization* is required among them

```
for i=1 to n do
  for j=1 to m do
    a(i,j)=a(2*i+2*j,2*j)+a(i,j-1)
```



$R1=\{[i,j] \rightarrow [2i+2j,2j] : 1 \leq j \ \&\ 2j \leq m \ \&\ 1 \leq i \ \&\ 2i+2j \leq n\}$

$R2=\{[i,j] \rightarrow [i,j+1] : 1 \leq i \leq n \ \&\ 1 \leq j < m\}$.

Limitations of ATF

- It fails to extract threads when *synchronization* is required among them

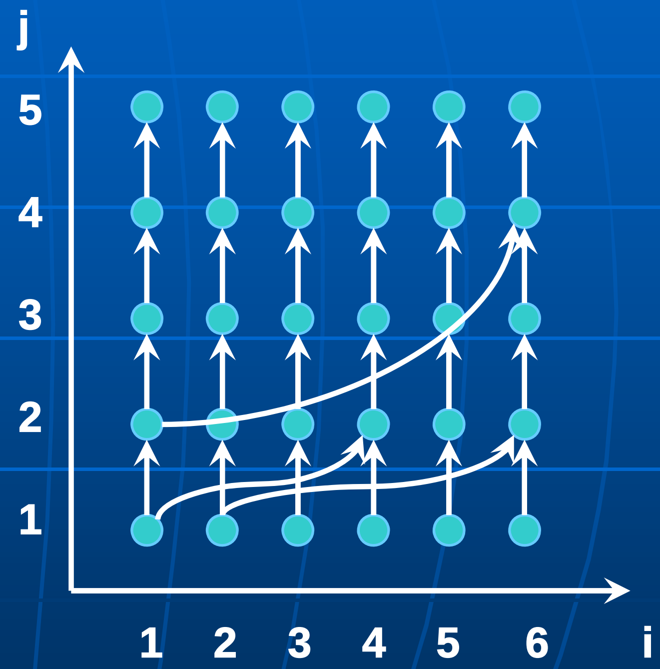
$$\begin{cases} C11*i + C12*j + C1 = C11*(2i+2j) + C12*(2j) + C1 \\ C11*i + C12*j + C1 = C11*i + C12*(j+1) + C1 \end{cases}$$

$$\Downarrow$$

$$\begin{cases} (-C11)*i + (-C12-2C11)*j = 0 \\ C12 = 0 \end{cases}$$

$$\Downarrow$$

$$\begin{cases} C12 = 0 \\ C11 = 0 \end{cases}$$



Limitations of the ATF motivate further research aimed at developing more advanced techniques for extracting parallelism

Slicing Framework

Program slicing (introduced by Mark Weiser in 1979) is a viable method to restrict the focus of a task to specific sub-components of a program.

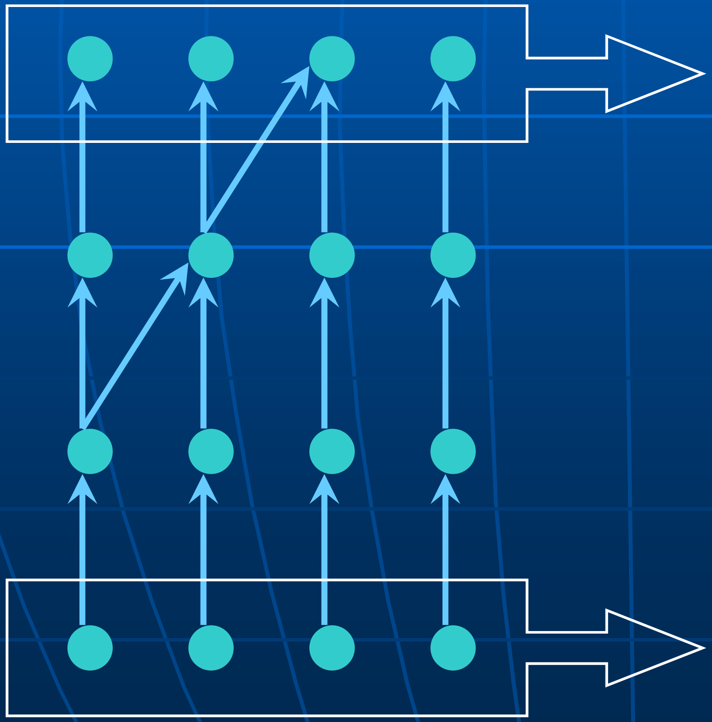
Iteration space slicing (introduced by Pugh in 1997) takes dependence information as input to find all operations which must be executed to produce the correct values for the specified array elements.

Slicing Framework

Definition 4. Operations I and J are called the ***source*** and ***destination*** of a dependence, respectively, provided that I is lexicographically smaller than J (I is executed before J).

Slicing framework

Definition 2. The source/destination of a dependence is the ***ultimate dependence source / destination*** if it is not the destination/source of any other dependence.

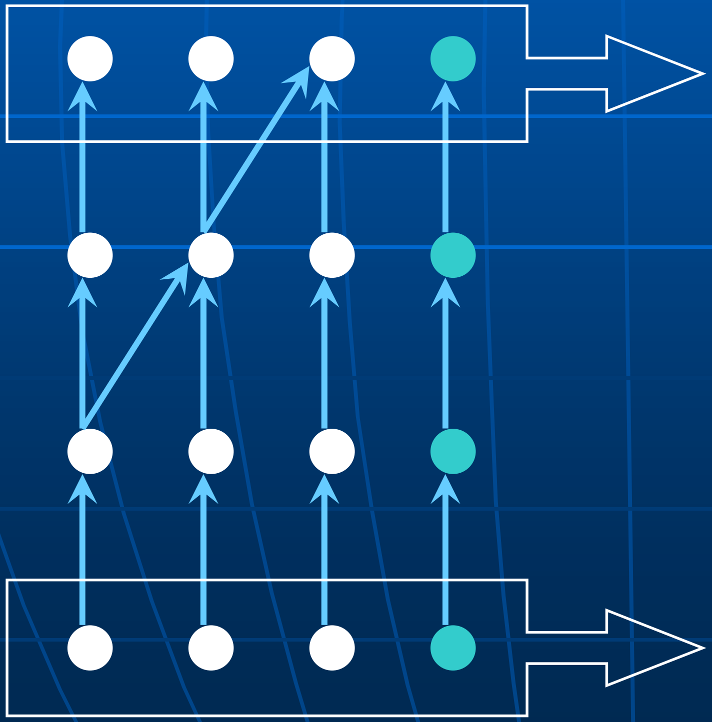


Ultimate dependence destinations

Ultimate dependence sources

Slicing framework

Definition 3. For a given set of dependence relations D, ***the slice*** of D is a maximal subset S of iterations such that there exists a (possibly indirect) path between any pair of iterations in S.

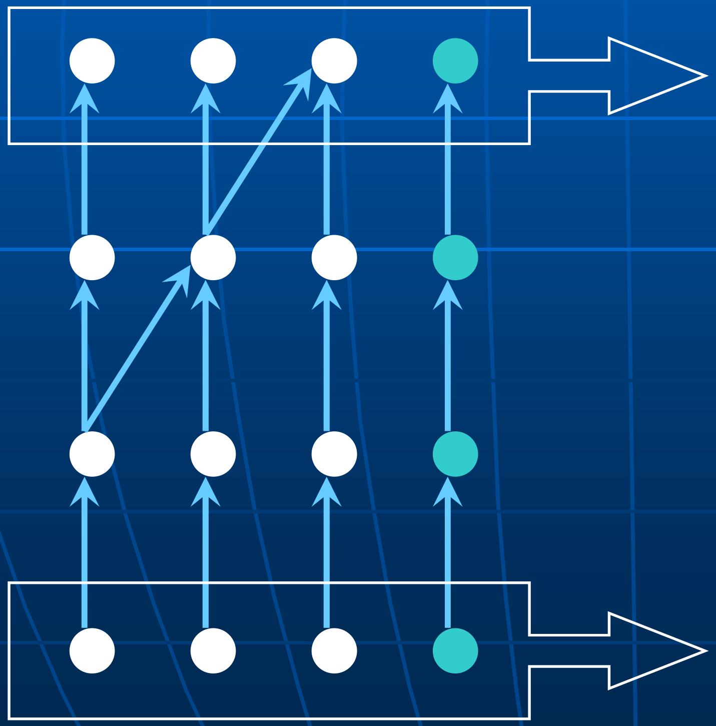


Ultimate dependence destinations

Ultimate dependence sources

Slicing framework

Definition 4. A slice is independent or ***synchronization-free*** if there is no dependence between the iterations in slice and the remaining iterations in the iteration space

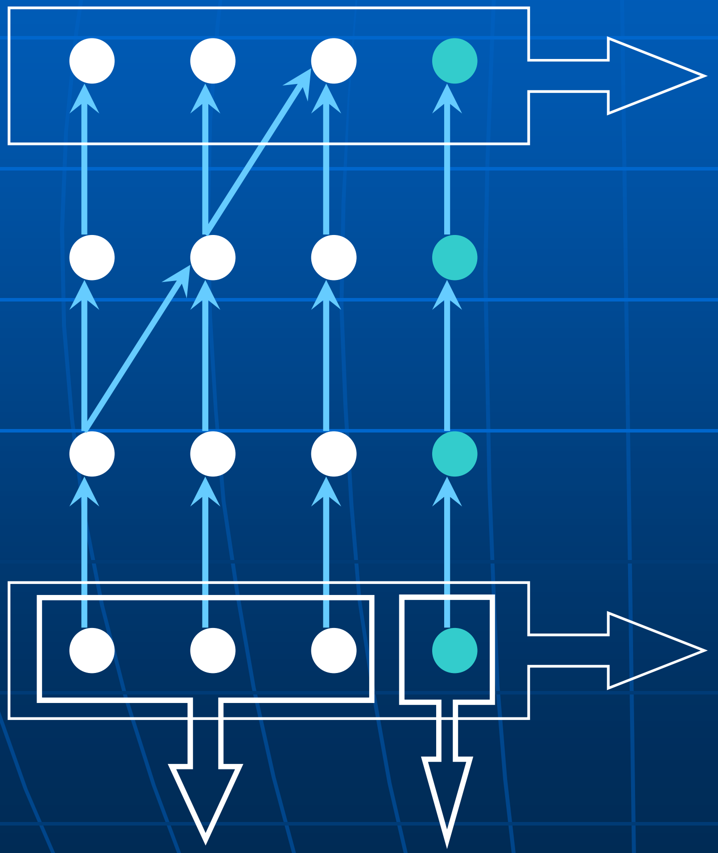


Ultimate dependence destinations

Ultimate dependence sources

Slicing framework

Definition 5. The ***source(s) of a slice*** is the ultimate dependence source(s) that this slice comprises.



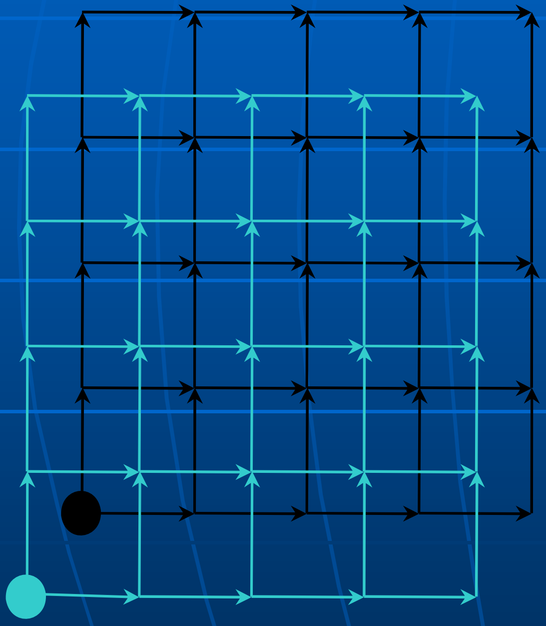
Ultimate dependence destinations

Ultimate dependence sources

Sources of Slice A **Source of Slice B**

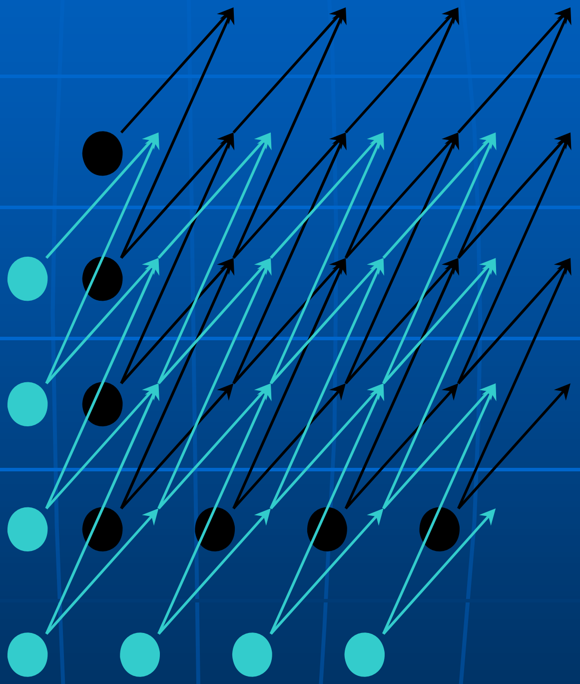
Examples of slices

Dependences in loop A:



Two slices with a single ultimate source each

Dependences in loop B:



Two slices with multiple ultimate sources each

Notations for each of loops A and B:

→ Dependences of Slice One

→ Dependences of Slice Two

● Ultimate sources of Slice One

● Ultimate sources of Slice Two

Modified Floyd-Warshal algorithm

Input: a set of A set of dependence relations $\{R_{i,j}\}$ describing direct dependences between each pair of statements i,j in an SCC

/* for some i,j, $R_{i,j}$ can be empty if a dependence analysis does not extract direct dependences between statements i and j */

```
foreach statement r
    foreach statement p
        foreach statement q
            $R_{p,q} = R_{p,q} \cup R_{r,q} \circ (R_{r,r})^* \circ R_{p,r}$
```

Output: At the end, each Ri,j describes all transitive dependences between statements i and j in the SCC.

Slicing algorithm[8]

INPUT:
Dependence relations representing an SCC

BEGIN

Find all ultimate dependence sources

Find sources of slices

Find operations of each slice

Generate code scanning slices and iterations of each slice in lexicographical order

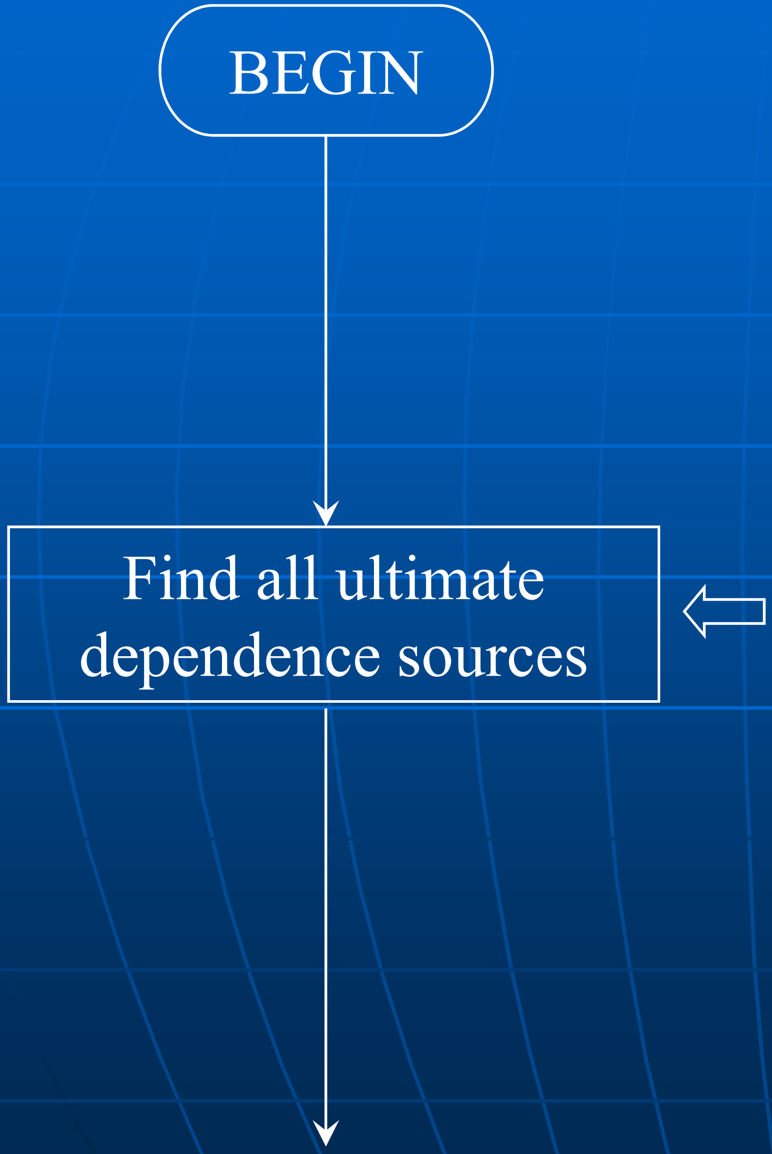
END

OUTPUT:
Parallel code

---

[8] Beletska A., Bielecki W., San Pietro P.: Finding synchronization-free slices of operations in arbitrarily nested loops. ICCSA 2008.

Slicing algorithm



BEGIN

Find all ultimate dependence sources

INPUT: n - dimension of loop

Set S={$\mathbf{R_{ij}}$ | **i,j** ∊ **[1,q]**}

Foreach relation $R_{i,j}$ ∊ S do

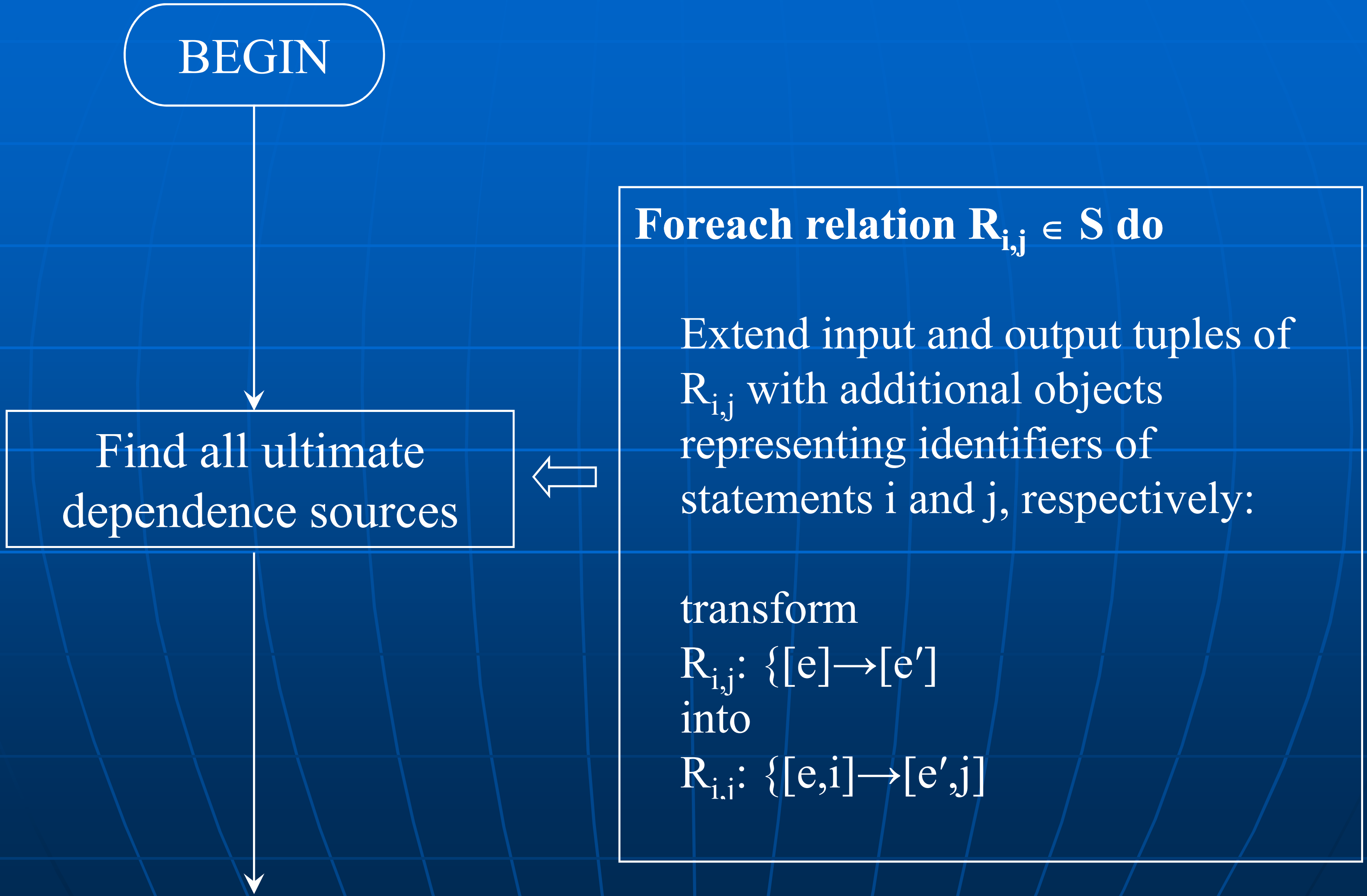
Normalize relation $R_{i,j}$ so that each input and output tuple has exactly n elements, by inserting value “-1” at the rightmost positions of tuples:

$[e]=[e_1\ e_2\ \ldots\ e_{n-k}]$,

where k is some integer, replace by a tuple

$[e_1\ e_2\ ..\ e_{n-k}\ -1\ -1\ \ldots\ -1]$.

Slicing algorithm



BEGIN

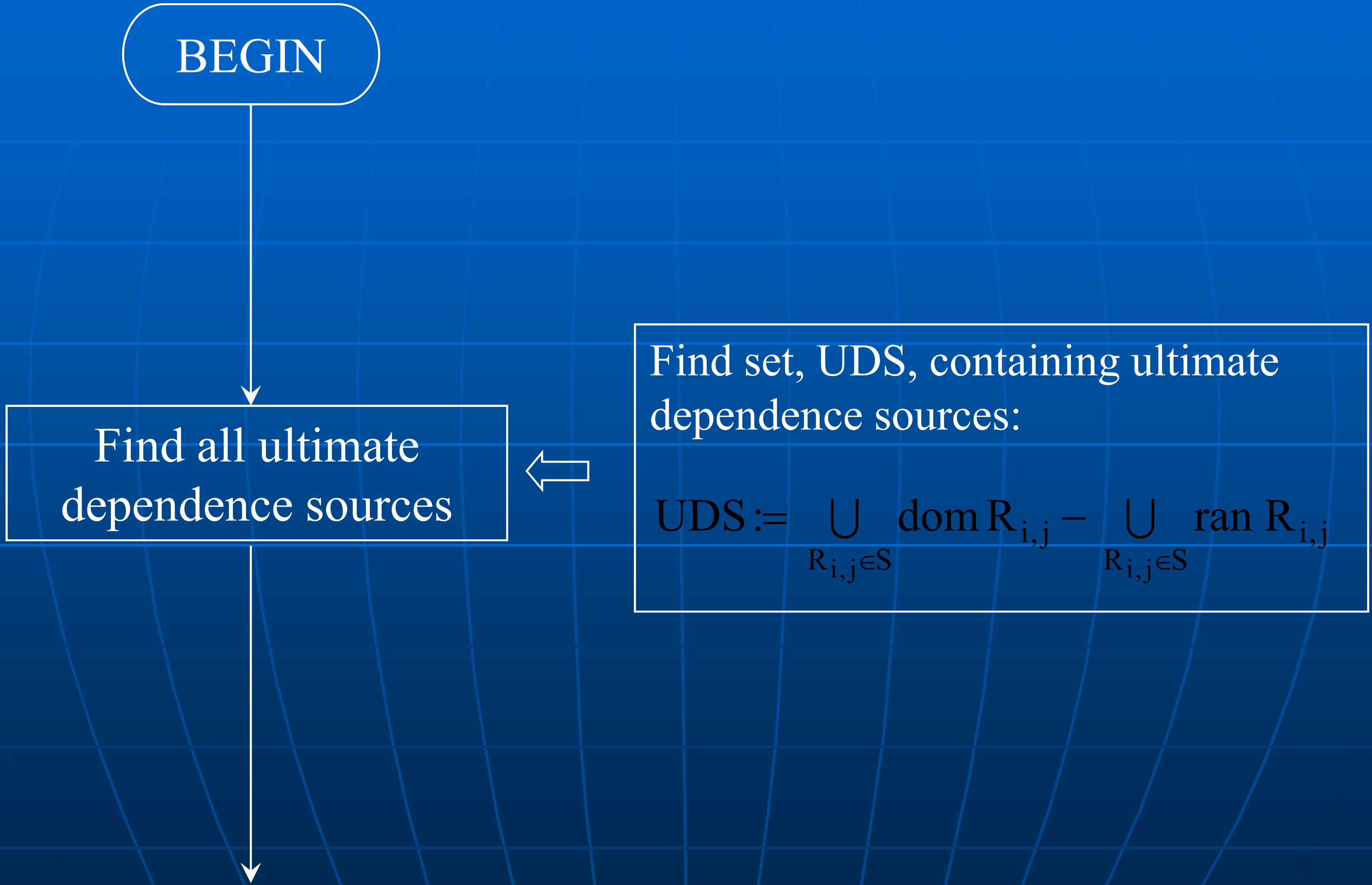
Find all ultimate dependence sources

Foreach relation $R_{i,j} \in S$ do

Extend input and output tuples of $R_{i,j}$ with additional objects representing identifiers of statements i and j, respectively:

transform
$R_{i,j}$: $\{[e] \rightarrow [e']$
into
$R_{i,j}$: $\{[e,i] \rightarrow [e',j]$

Slicing algorithm



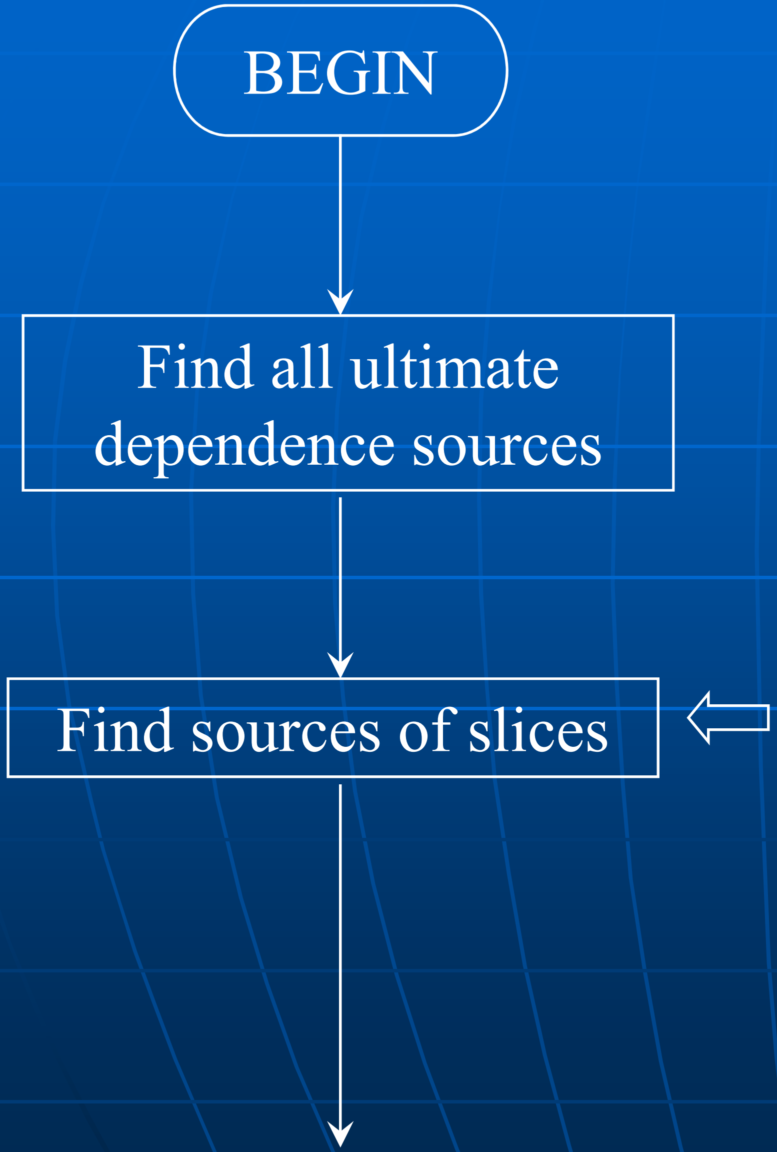
BEGIN

Find all ultimate dependence sources

Find set, UDS, containing ultimate dependence sources:

$$UDS := \bigcup_{R_{i,j} \in S} \operatorname{dom} R_{i,j} - \bigcup_{R_{i,j} \in S} \operatorname{ran} R_{i,j}$$

Slicing algorithm



BEGIN

Find all ultimate dependence sources

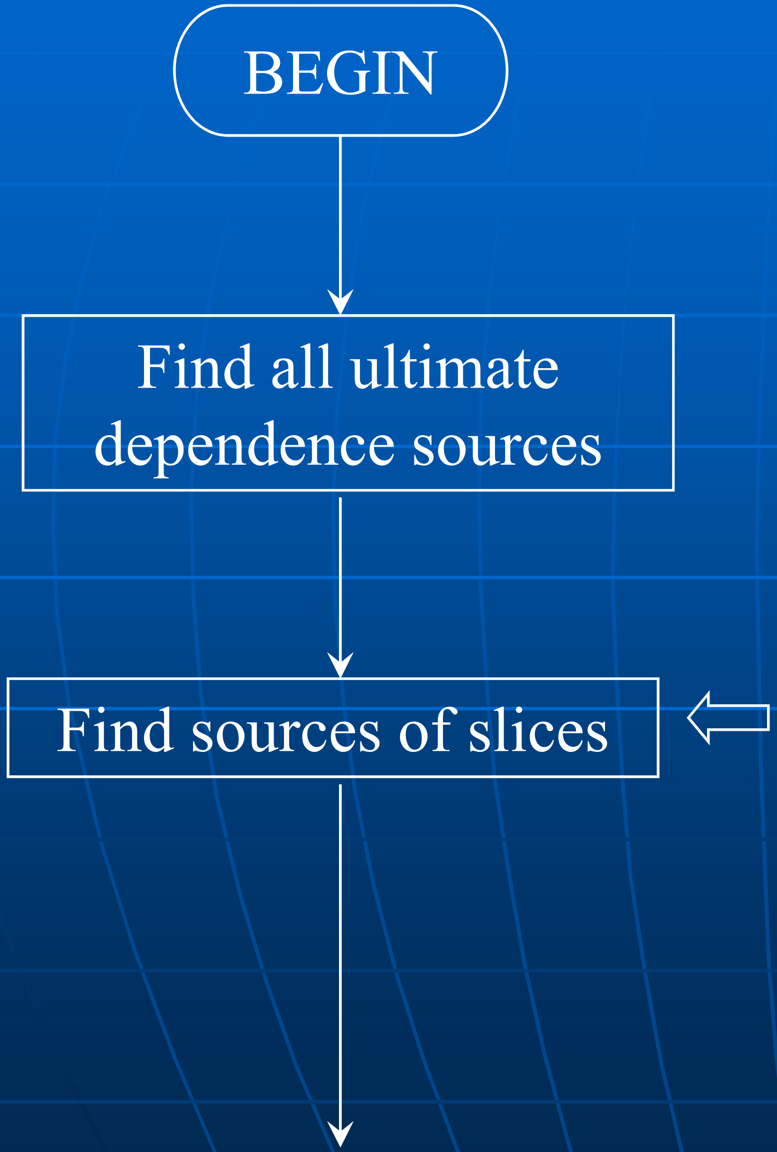
Find sources of slices

Calculate exact transitive closure, R*, representing all the transitive dependenes in SCC, by applying the modified Floyd-Warshal algorithm to calculate relations $\overline{R}^{+}_{i,j}$ representing all transitive dependences between each pair of statements i, j in SCC:

$$R^* = \bigcup_{1 \le i, j \le q} (\overline{R}^{+}_{i,j}) \cup I$$

where I is the identity relation.

Slicing algorithm


BEGIN

Find all ultimate dependence sources

Find sources of slices

Form relation R_UCS representing all pairs of ultimate dependence sources that are connected (by an indirect path) in the dependence graph formed by R:

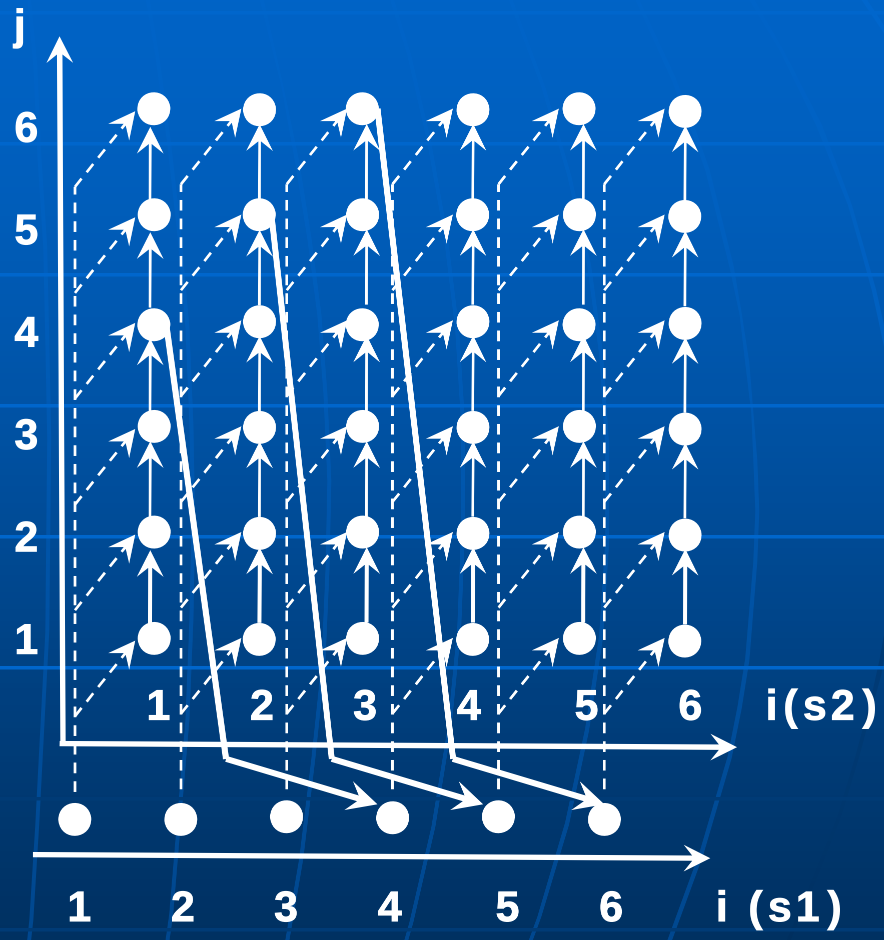
$$\mathbf{R\_UCS := \{[e] \rightarrow [e']:\ e, e' \in UDS,\ e' \prec e,\ range\ (R^*(e')) \cap range\ (R^*(e)) \neq \emptyset\ \}.}$$

Form set, Sources, comprising the (lexicographically minimal) sources of slices:

$$\mathbf{Sources := UDS - range\ R\_UCS}$$

Example of parallelization

```
for i = 1 to n do
s1:     b(i,i) = a(i-3,i)
        for j = 1 to n do
s2:         a(i,j)=a(i,j-1)+b(i,j);
```



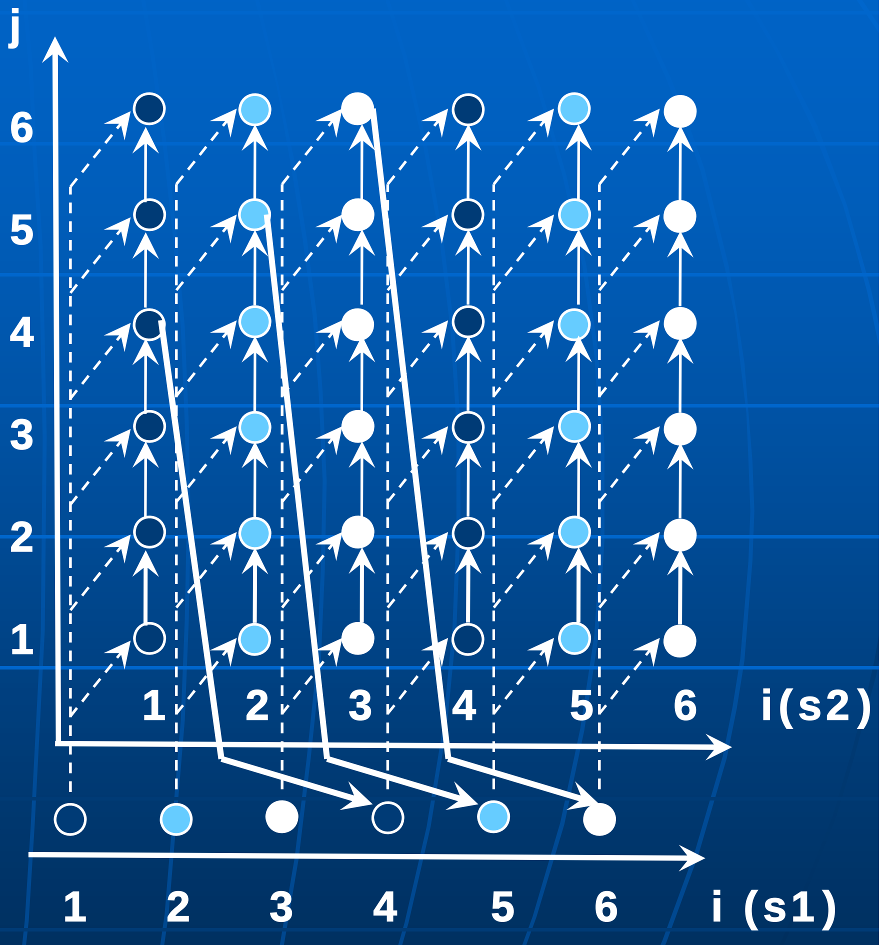
$R_{1,1} := \{ [i] \rightarrow [i,j]: 1 \le i \le n \ \& \ 1 \le j < n \};$

$R_{2,1} := \{ [i,i+3] \rightarrow [i+3]: 1 \le i \le n-3 \};$

$R_{2,2} := \{ [i,j] \rightarrow [i,j+1]: 1 \le i \le n \ \& \ 1 \le j < n \};$

Example of parallelization

```
for i = 1 to n do
s1:     b(i,i) = a(i-3,i)
        for j = 1 to n do
s2:         a(i,j)=a(i,j-1)+b(i,j);
```



$R_{1,1} := \{ [i] \rightarrow [i,j]: 1 \le i \le n \ \& \ 1 \le j < n \};$

$R_{2,1} := \{ [i,i+3] \rightarrow [i+3]: 1 \le i \le n-3 \};$

$R_{2,2} := \{ [i,j] \rightarrow [i,j+1]: 1 \le i \le n \ \& \ 1 \le j < n \} ;$

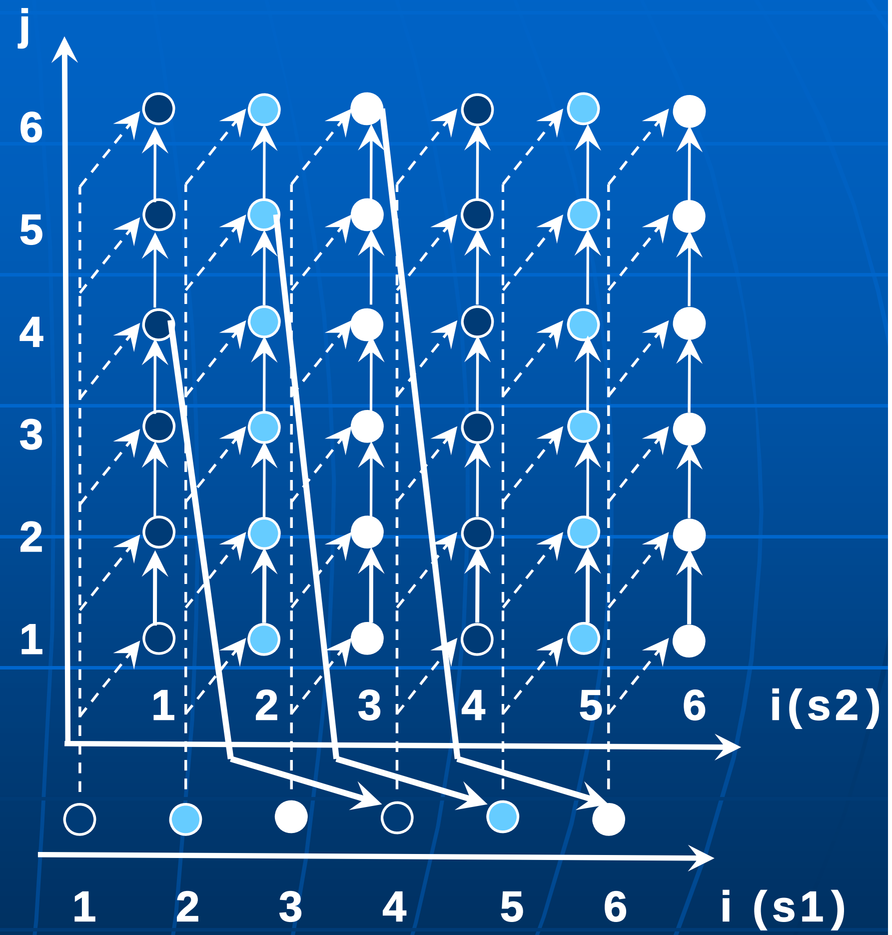
Example of parallelization

$R_{1,1}$:= { [i,-1,1] → [i,j,1]:
1≤i≤n & 1≤j<n }

$R_{2,1}$:= { [i,i+3,2] → [i+3,-1,1]:
1≤i≤n-3}

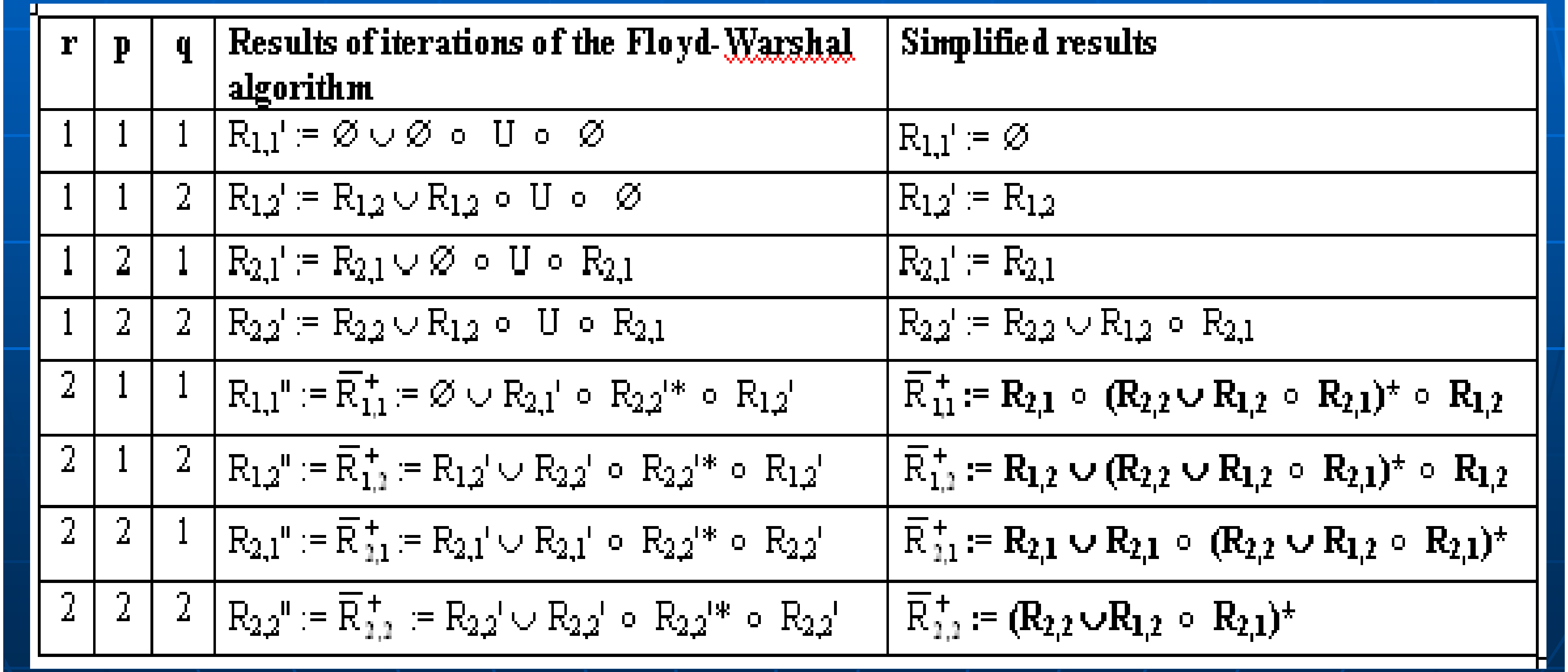
$R_{2,2}$:= { [i,j,2] → [i,j+1,2]:
1≤i≤n & 1≤j<n}

UDS:= { [i,-1,1] : 1 ≤ i ≤ min(n,3) }



Example of parallelization

In order to computer the exact transitive closure R*, we first find relations $\overline{R}^{+}_{1,1}$, $\overline{R}^{+}_{1,2}$, $\overline{R}^{+}_{2,1}$, $\overline{R}^{+}_{2,2}$ according to the modified Floyd-Warshal algorithm.



r	**p**	**q**	**Results of iterations of the Floyd-Warshal algorithm**	**Simplified results**
1	1	1	$R_{1,1}' := \varnothing \cup \varnothing \circ U \circ \varnothing$	$R_{1,1}' := \varnothing$
1	1	2	$R_{1,2}' := R_{1,2} \cup R_{1,2} \circ U \circ \varnothing$	$R_{1,2}' := R_{1,2}$
1	2	1	$R_{2,1}' := R_{2,1} \cup \varnothing \circ U \circ R_{2,1}$	$R_{2,1}' := R_{2,1}$
1	2	2	$R_{2,2}' := R_{2,2} \cup R_{1,2} \circ U \circ R_{2,1}$	$R_{2,2}' := R_{2,2} \cup R_{1,2} \circ R_{2,1}$
2	1	1	$R_{1,1}'' := \overline{R}^{+}_{1,1} := \varnothing \cup R_{2,1}' \circ R_{2,2}'^{*} \circ R_{1,2}'$	$\overline{R}^{+}_{1,1} := \mathbf{R_{2,1} \circ (R_{2,2} \cup R_{1,2} \circ R_{2,1})^{*} \circ R_{1,2}}$
2	1	2	$R_{1,2}'' := \overline{R}^{+}_{1,2} := R_{1,2}' \cup R_{2,2}' \circ R_{2,2}'^{*} \circ R_{1,2}'$	$\overline{R}^{+}_{1,2} := \mathbf{R_{1,2} \cup (R_{2,2} \cup R_{1,2} \circ R_{2,1})^{*} \circ R_{1,2}}$
2	2	1	$R_{2,1}'' := \overline{R}^{+}_{2,1} := R_{2,1}' \cup R_{2,1}' \circ R_{2,2}'^{*} \circ R_{2,2}'$	$\overline{R}^{+}_{2,1} := \mathbf{R_{2,1} \cup R_{2,1} \circ (R_{2,2} \cup R_{1,2} \circ R_{2,1})^{*}}$
2	2	2	$R_{2,2}'' := \overline{R}^{+}_{2,2} := R_{2,2}' \cup R_{2,2}' \circ R_{2,2}'^{*} \circ R_{2,2}'$	$\overline{R}^{+}_{2,2} := \mathbf{(R_{2,2} \cup R_{1,2} \circ R_{2,1})^{*}}$

Results of iterations of the Floyd-Warshal Algorithm

Example of parallelization

Next, we compute R* as follows:

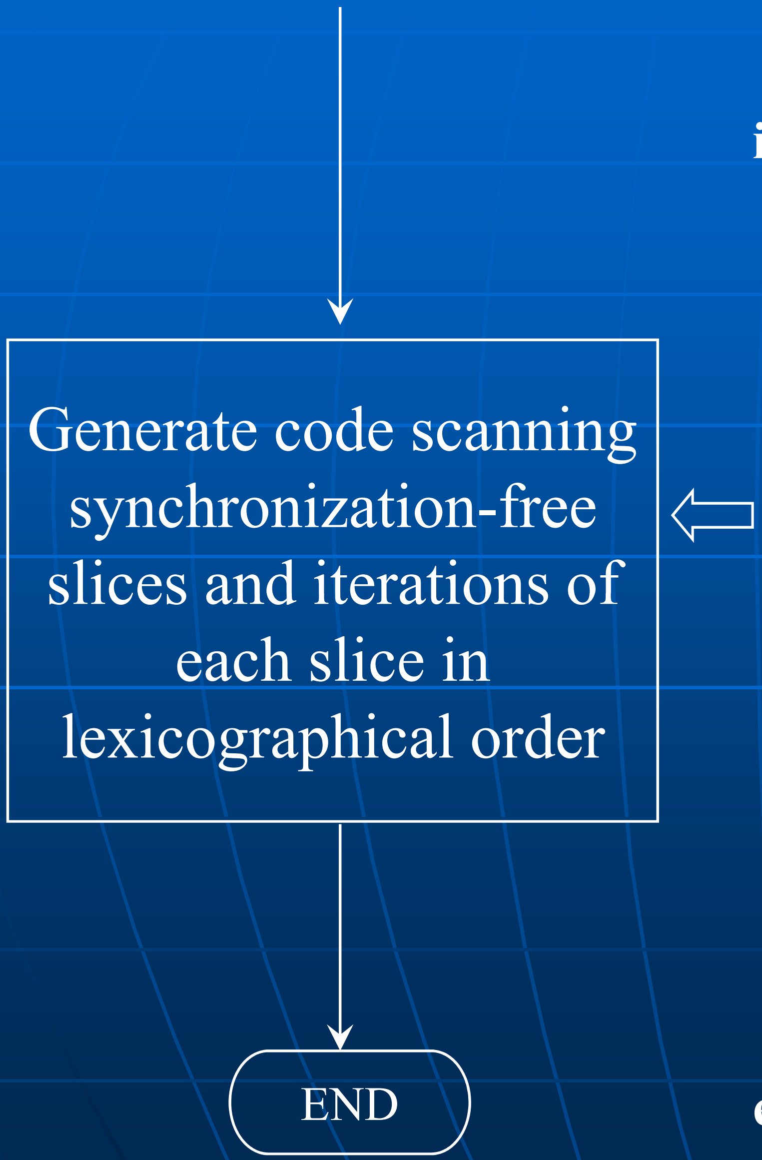
R* $:= \overline{R}^{+}_{1,1} \cup \overline{R}^{+}_{1,2} \cup \overline{R}^{+}_{2,1} \cup \overline{R}^{+}_{2,2} \cup U =$

{[i,-1,1] →[i',-1,1] : ∃ (alpha : i' = i+3alpha & 1≤i≤i'-3 & i'≤n)} ∪{[i,-1,1] → [i,j',2] : 1≤i≤n & 1≤j'≤n & 4≤n} ∪ {[i,-1,1] → [i',j',2] : ∃ (alpha : i+3alpha = i' & 1≤i≤i'-3 & 1≤j'≤n & i'≤n)} ∪ {[i,-1,1] →[i,j',2] : j', i≤n≤3 & 1≤i & 1≤j'} ∪ {[i,i+3,2] →[i+3,-1,1] : 1≤i≤n-3} ∪ {[i,j,2] →[i+3,-1,1] : 1,j-2≤i≤ n-3 & 1≤j} ∪ {[i,j,2] -> [i,j',2]: 1≤j≤j'≤n & 1≤i≤n} ∪ {[i,j,In_3] →[i,j,In_3] }.

R_UCS := ∅

Sources := UDS – R_UCS = {[I,-1,1] : 1 ≤ i ≤ min(n,3) }

Slicing algorithm



Generate code scanning synchronization-free slices and iterations of each slice in lexicographical order

END

if Sources ≠ ∅ **then**

```
genLoops (in: Sources;
        out: OuterLoops, L_I);

foreach I in L_I do
    S_Slice := R* (R_UCS*(I))
    // $\overline{R}^{+}_{L1}$ note: if  R_UCS = ∅  then R_UCS*(I)=I
    genLoops (in: S_Slice;
            out: InnerLoops, L_J);

    foreach J in L_J do
       genLoopBody (in:OuterLoops,InnerLoops,J;
                             out:LoopBody);
```

end

Code generation

- To generate the code, well known techniques can be applied[9]

9 Ancourt C., Irigoin F., Scanning polyhedra with do loops, in: Proceedings of the Third ACM/SIGPLAN Symposium on Principles and Practice of Parallel Programming, ACM Press. (1991) pp. 39-50 .

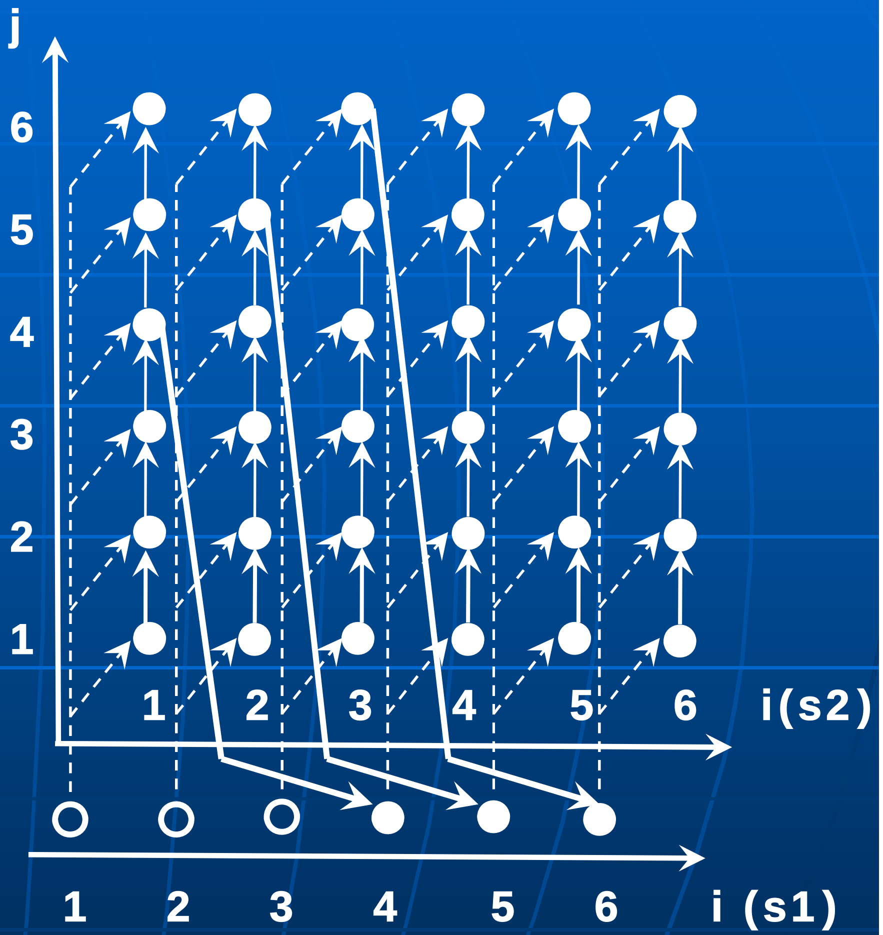
9 Bastoul C. Code Generation in the Polyhedral Model Is Easier Than You Think. In *Proceedings of the PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, Juan-les-Pins. (2004) 7-16

9 Boulet P., Darte A., Silber G.A., Vivien F., Loop parallelization algorithms: from parallelism extraction to code generation, Parallel Computing, 24. (1998), pp. 421-444

9 Quillere F., Rajopadhye S., Wilde D., Generation of efficient nested loops from polyhedra, International Journal of Parallel Programming 28. (2000)

9 Vasilache N., Bastoul C., and Cohen A. Polyhedral code generation in the real world. In *Proceedings of the International Conference on Compiler Construction (ETAPS CC'06)*, LNCS, pp 185--201, Vienna, Austria, March 2006. Springer-Verlag

Example of parallelization



Code generation for set Sources

To generate a nest of outer loops scanning sources of synch.-free slices comprised in set Sources, we apply the Omega code generator and get:

```
for(t1=1; t1<=min(n,3); t1++)
        s1(t1,-1,1);
```

List L_I contains single vector I equal to (t1,-1,1)'.

Example of parallelization

Code generation for set Sources

Generate inner loops to enumerate iterations belonging to the slice with a source represented by vector I=(t1,-1,1)’.

Find set **S_Slice** := R* (R_UCS* (I)) = $\{[i,-1,1]:\ (\text{alpha} : i = t1+3\text{alpha}\ \&\ 1\le t1\le i-3\ \&\ i\le n)\} \cup \{[t1,j,2]: 1\le t1\le n\ \&\ 1\le j\le n\ \&\ 4\le n\}\cup\{[i,j,2]:\ (\text{alpha} : i = t1+3\text{alpha}\ \&\ 1\le t1\le i-3\ \&\ 1\le j\le n\ \&\ i\le n)\} \cup \{[t1,j,2]: 1\le t1\le n\le 3\ \&\ 1\le j\le n\} \cup \{[i,-1,1]: i = t1\}$.

Applying the Omega code generator to set S_Slice, we yield the inner loops.

```
s(t1,-1,1);
for(t2 = 1; t2 <= n; t2++)
    s(t1,t2,2);
for(t3 = t1+3; t3<=n; t3+=3) {
    s(t3,-1,1);
    for(t2 = 1; t2 <= n; t2++)
      s(t3,t2,2);
}
```

List L_J contains single vectors: (t1,-1,1)´, (t1,t2,2)´, (t3,-1,1)´ and (t3,t2,2)´

Example of parallelization

Code generation for set Sources

Generate the body of the inner loops containing statements of the source loop body to be executed at iteration J, and insert the generated code as the body of outer loops.

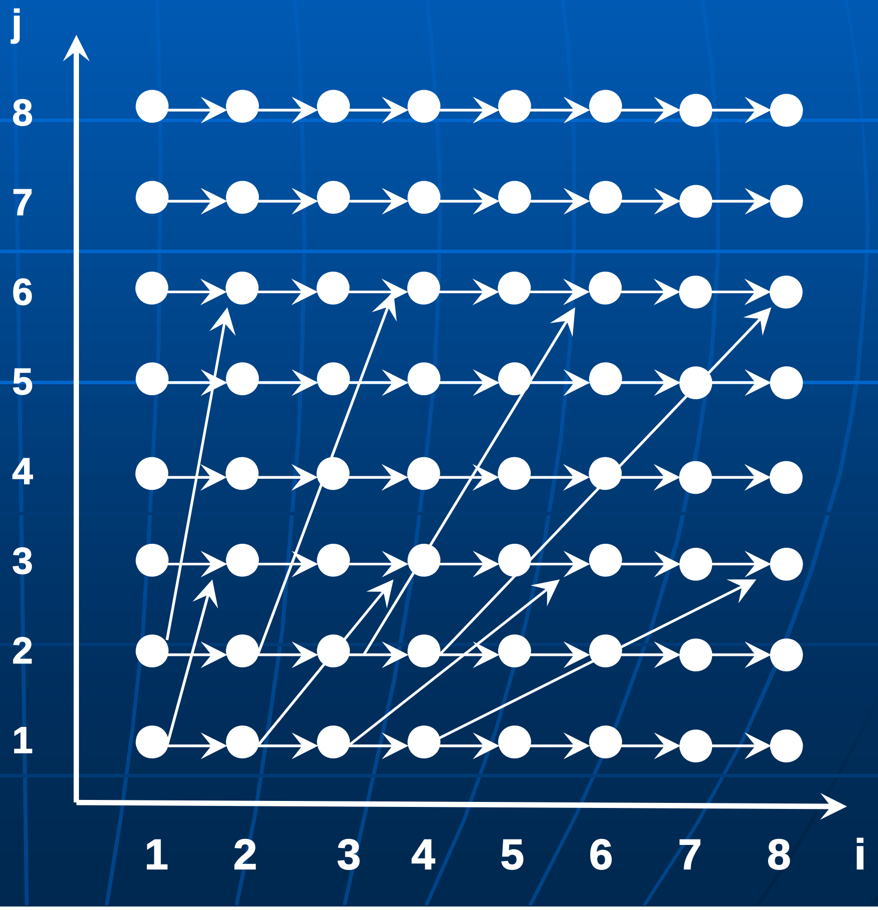
The resulting code is as follows:

```
parfor (t1 = 1; t1 <= min(n,3); t1++) {
     b(t1,t1)=a(t1-3,t1);
     for(t2 = 1; t2 <= n; t2++)
         a(t1,t2)=a(t1,t2-1)+b(t1,t1);
    for(t3=t1+3; t3<= n; t3+= 3) {
         b(t3,t3)=a(t3-3,t3);
         for(t2 = 1; t2 <= n; t2++)
              a(t3,t2)=a(t3,t2-1)+b(t3,t3);
    }
}
```

Slicing algorithm

- Is applicable to perfectly-nested both uniform and non-uniform loops

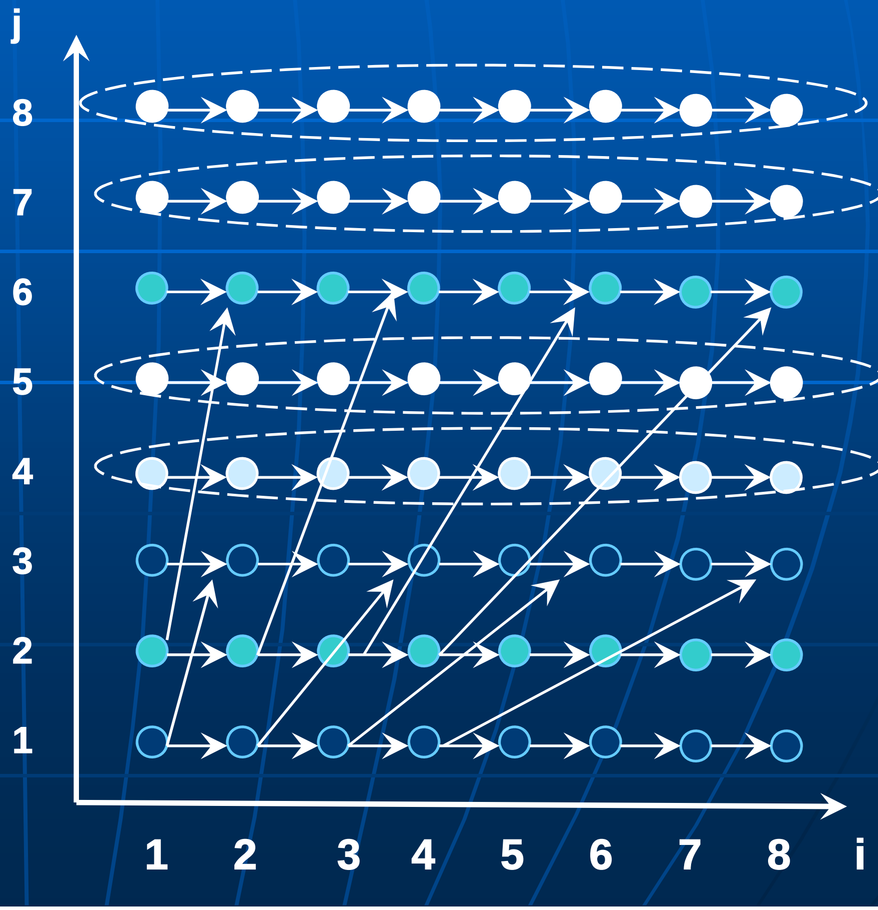
```
for i = 1 to n do
    for j = 1 to n do
        a(2*i, 3*j) = b(i,j)
        b(i+1, j) = a(i, j)
```



Slicing algorithm

- Is applicable to perfectly-nested both uniform and non-uniform loops

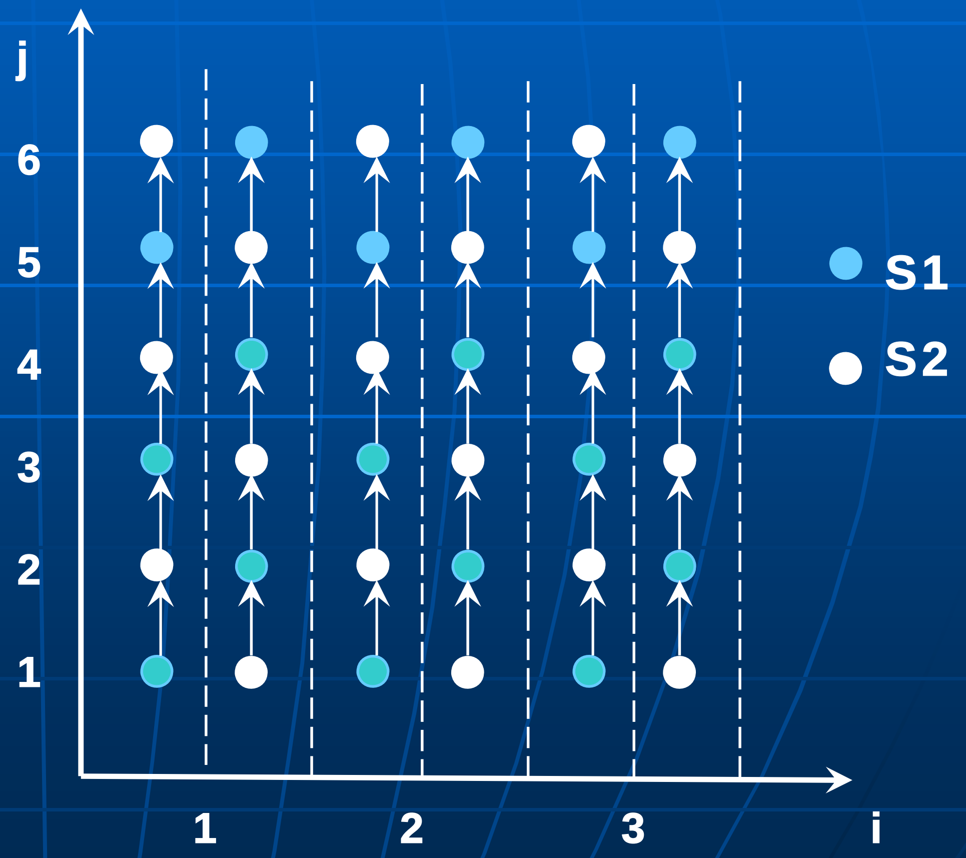
```
for i = 1 to n do
    for j = 1 to n do
        a(2*i, 3*j) = b(i,j)
        b(i+1, j) = a(i, j)
```



Slicing algorithm

- Permits us to extract more slices than that extracted by ATF

```
for i=1 to n do
   for j=1 to m do
s1:  a(i,j)=b(i,j)+c(i,j)
s2:  c(i,j-1)=a(i,j+1)
```



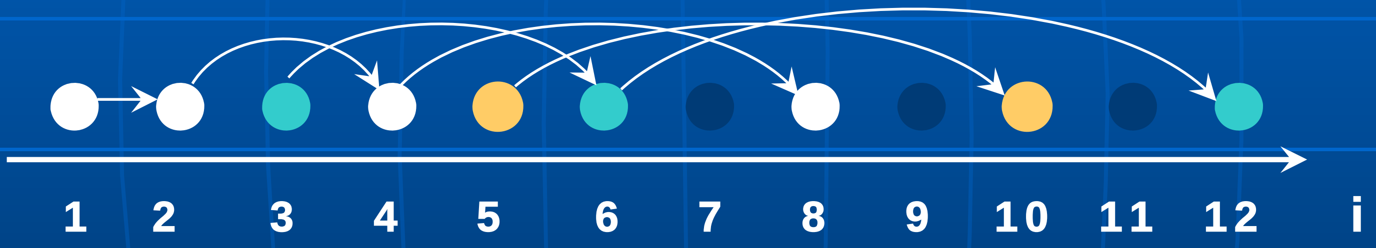
Slicing algorithm

- Can be applied to loops when the following conditions are satisfied:
  - Exact dependence analysis can be performed for these loops
  - Exact transitive closure can be calculated for dependence relations describing dependences in the loops

Presburger arithmetic limitations

```
for i=1 to n do
    a(i)=a(2*i)
```

$R := \{[i] \to [2i] : 1 \leq i,\ 2i \leq n\}.$



Omega does not extract the exact positive transitive closure for this example, because it is represented with non-linear expressions and is of the form:

$$R+ = \{[i] \to [j] : \text{Exists} (k: k \geq 1 \ \&\& \ j=2^k * i \ \&\& \ 1 \leq i, j \leq n)\}$$

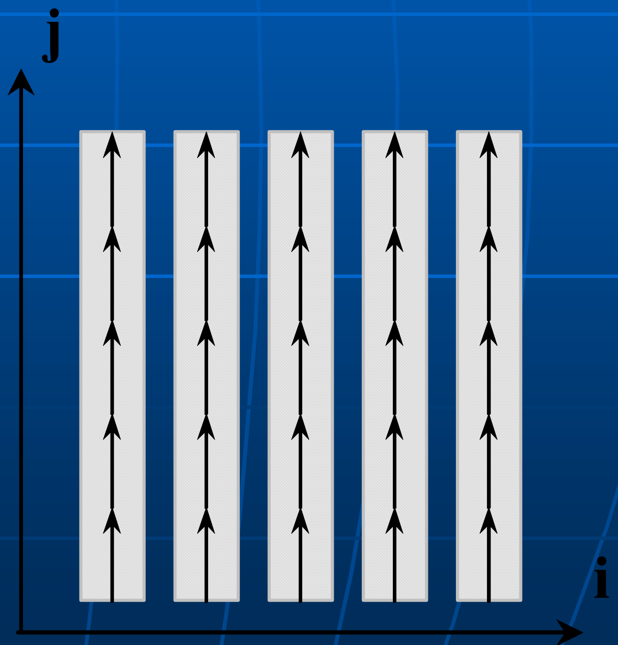
Fine-grained parallelism

- In some cases, code representing slices (coarse-grained parallelism) can be simply transformed into code representing fine-grained parallelism

```
parfor i = 1 to n do
    for j = 1 to n do
        a(i,j) = a(i,j-1)
```

⇩

```
for i = 1 to n do
    parfor j = 1 to n do
        a(i,j) = a(i,j-1)
```



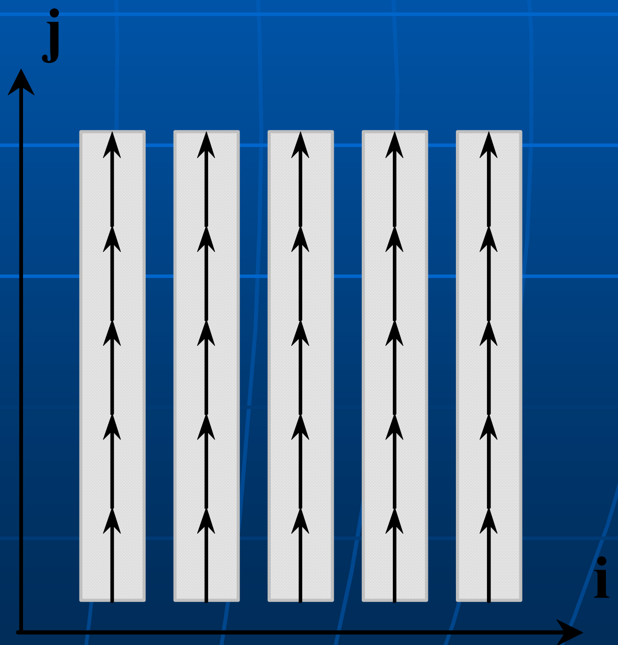
Fine-grained parallelism

- In some cases, code representing slices (coarse-grained parallelism) can be simply transformed into code representing fine-grained parallelism

```
parfor i = 1 to n do
    for j = 1 to n do
        a(i,j) = a(i,j-1)
```

⇩

```
for i = 1 to n do
    parfor j = 1 to n do
        a(i,j) = a(i,j-1)
```



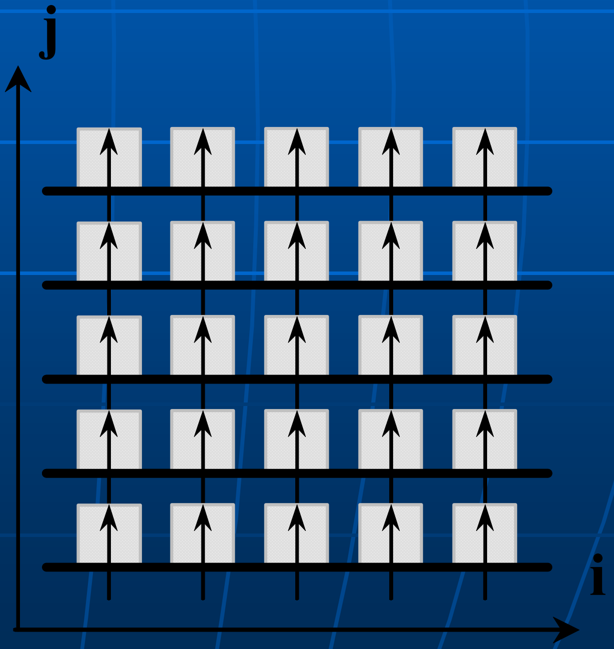
Fine-grained parallelism

- In some cases, code representing slices (coarse-grained parallelism) can be simply transformed into code representing fine-grained parallelism

```
parfor i = 1 to n do
    for j = 1 to n do
        a(i,j) = a(i,j-1)
```

⇩

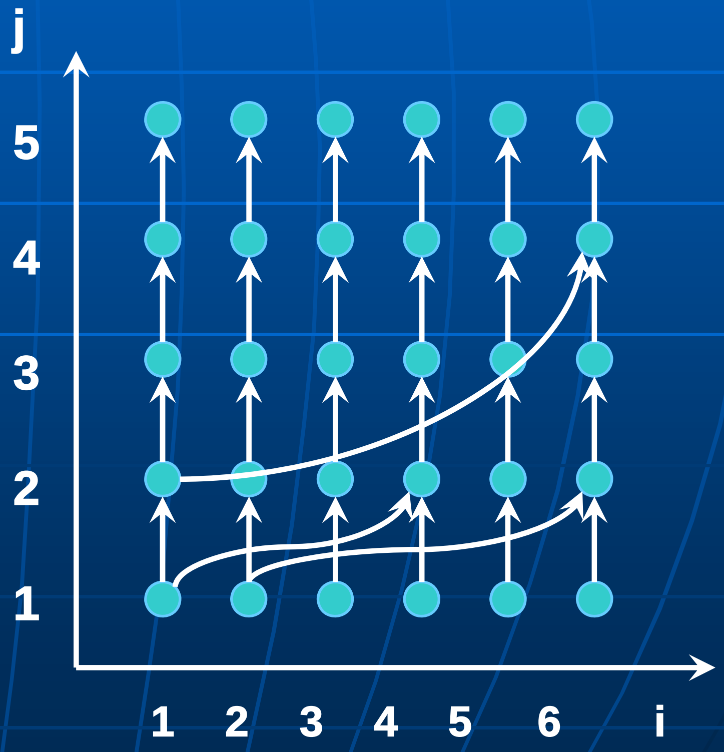
```
for i = 1 to n do
    parfor j = 1 to n do
        a(i,j) = a(i,j-1)
```



Further research

- Development of approaches to extract slices requiring *synchronization*

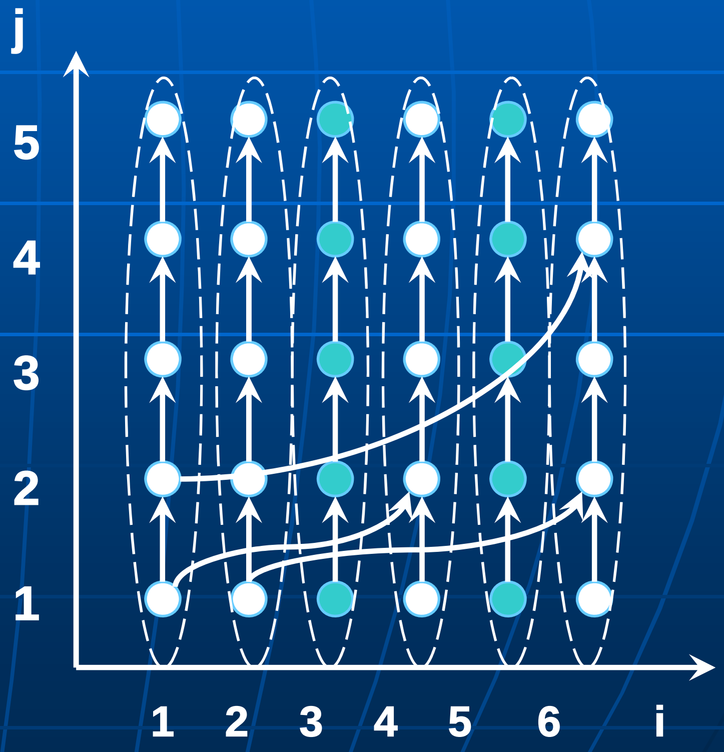
```
for i=1 to n do
  for j=1 to m do
    a(i,j)=a(2*i+2*j,2*j)+a(i,j-1)
```



Further research

- Development of approaches to extract slices requiring *synchronization*

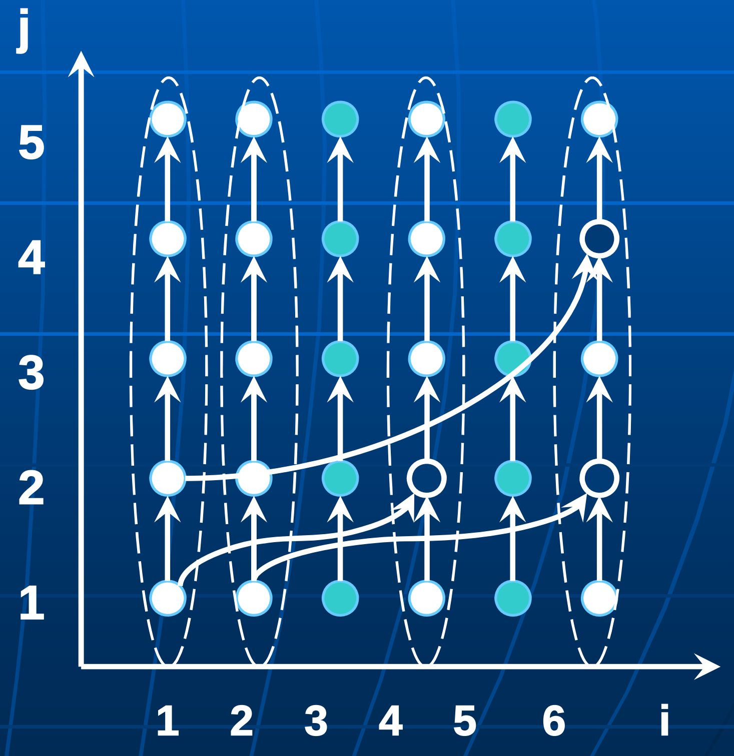
```
for i=1 to n do
  for j=1 to m do
    a(i,j)=a(2*i+2*j,2*j)+a(i,j-1)
```



Further research

- Development of approaches to extract slices requiring *synchronization*

```
for i=1 to n do
  for j=1 to m do
    a(i,j)=a(2*i+2*j,2*j)+a(i,j-1)
```



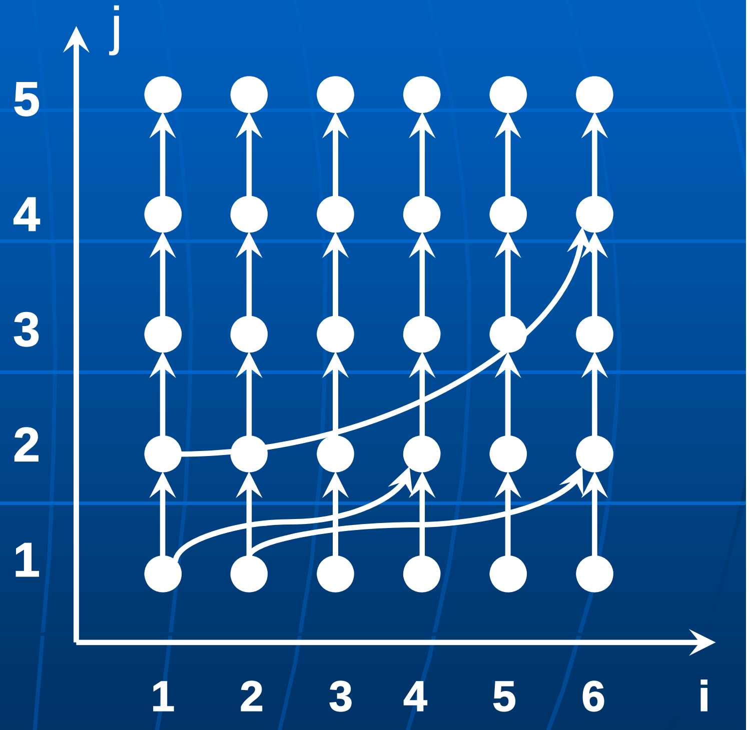
Further research

- Calculation of exact transitive closure described by non-linear forms
- Derivation of approaches to generate code scanning elements of sets represented with non-linear forms
- Experiments with benchmarks

Further research

- Development of approaches combining ATF with the slicing framework

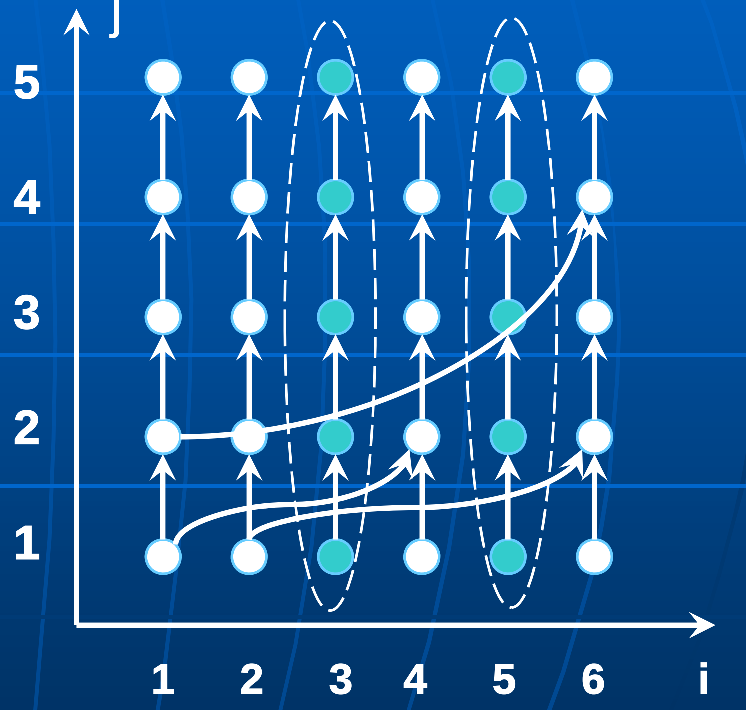
```
for i=1 to n do
  for j=1 to m do
    a(i,j)=a(2*i+2*j,2*j)+a(i,j-1)
```



Further research

- Development of approaches combining ATF with the slicing framework

```
for i=1 to n do
  for j=1 to m do
    a(i,j)=a(2*i+2*j,2*j)+a(i,j-1)
```

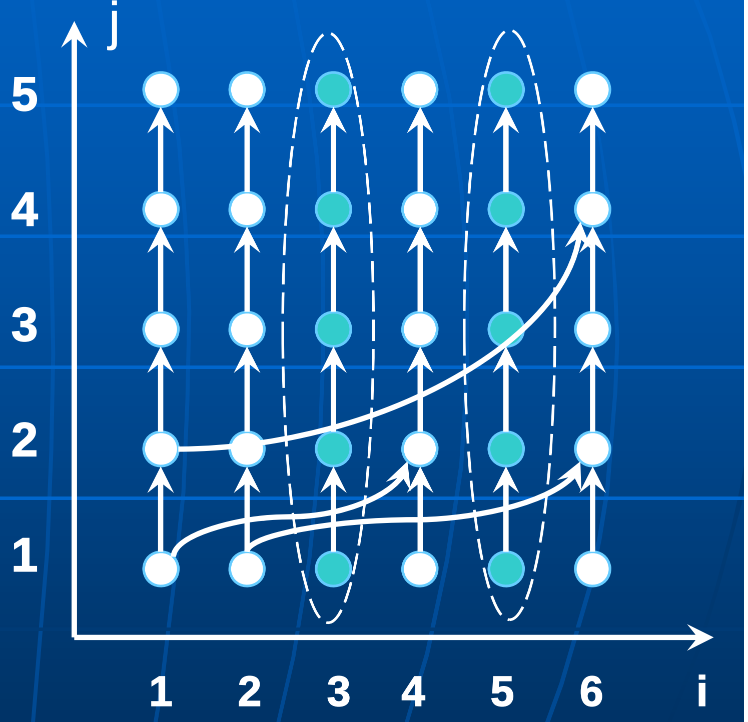


- extract subdomains of a loop by means of the slicing framework,
- to each subdomain, apply the ATF (time partitioning).

Further research

- Development of approaches combining ATF with the slicing framework

```
for i=1 to n do
  for j=1 to m do
    a(i,j)=a(2*i+2*j,2*j)+a(i,j-1)
```



Such a hybrid technique could permit us for less complexity in comparison with that of the slicing framework

Thank you very much for your attention!

Thank you very much for your attention!