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

155Mb/s WLAN 5GHz

<1 Watt

10Gop/s MPEG 4

>100 Gop/s 5 Gtr/s 5 Watt

Reconfigurable Access Terminal

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

---


Approaches to extract CGP

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

---

Approaches to extract CGP

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

Approaches to extract CGP

- Affine transformation framework\(^4\)


Approaches to extract CGP

- Slicing framework\textsuperscript{5}

\textsuperscript{5} Weiser M. \textit{Program slices: formal, psychological, and practical investigations of an automatic program abstraction method}. PhD thesis, University of Michigan, Ann Arbor, MI. (1979)


\textsuperscript{5} Pugh W., Rosser E. Iteration Space Slicing and Its Application to Communication Optimization In \textit{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.

\[ \{ [i,j] : 1 \leq i,j \leq 5 \} \]

Presburger formula

\[ \{ [i,j] : 1 \leq i,j \leq 5 \} \]

Iteration space

Presburger formula

\[ \{ [i,j] \rightarrow [i,j+1] : 1 \leq i < j \leq 5 \} \]

Data dependences

Source

Destination

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

... ▷ ▷ ▷ ...

I3 ▷ I6 ▷ I9 ▷ ...

I2 ▷ I5 ▷ I8 ▷ ...

I1 ▷ I4 ▷ I7 ▷ ...

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\textsuperscript{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.


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 $i+1$. 
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/
We get the information about dependences:

- **Flow 3:**
  - Source: `a(i,j)`
  - Target: `a(i,j-1)`
  - Dependence set: `{[i,j] -> [i,j+1] : 1 <= i <= m && 1 <= j < m}`

- **Flow 5:**
  - Source: `a(i,j)`
  - Target: `a(i,j-1)`
  - Dependence set: `{[i,j] -> [i,j+1] : 1 <= i <= m && 1 <= j < m}`

- **Flow 4:**
  - Source: `b(i,j)`
  - Target: `b(i-1,j)`
  - Dependence set: `{[i,j] -> [i+1,j] : 1 <= i < m && 1 <= j <= m}`

- **Flow 5:**
  - Source: `b(i,j)`
  - Target: `b(i-1,j)`
  - Dependence set: `{[i,j] -> [i+1,j] : 1 <= i < m && 1 <= j <= m}`
Example of parallelization by ATF

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

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.

Find the affine transforms for each SCC

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:

```c
#pragma 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.

```plaintext
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) → (i,j+1) : 1 ≤ i ≤ n && 1 ≤ j < m]}
R2={[(i,j) → (i,j+1) : 1 ≤ i ≤ n && 1 ≤ j < m]}
```
Limitations of ATF

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

\[
R_1 = \{ [i,j] \rightarrow [i,j+1] : 1 \leq i \leq n \land 1 \leq j < m \} \\
R_2 = \{ [i,j] \rightarrow [i,j+1] : 1 \leq i \leq n \land 1 \leq j < m \} \\
\]

\[
\begin{aligned}
C_{11}i + C_{12}j + C_1 &= C_{21}i + C_{22}j + C_2 \\
C_{21}i + C_{22}j + C_2 &= C_{11}i + C_{12}j + C_{12} + C_1 \\
\end{aligned}
\]

\[
\begin{aligned}
C_{11} &= C_{21} = \text{arbitrary value, let it be } n_1, n_1 \geq 0. \\
C_{12} &= C_{22} = 0 \\
\end{aligned}
\]
Limitations of ATF

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

```latex
\begin{align*}
\text{for } i = 1 \text{ to } n \text{ do} \\
\quad \text{for } j = 1 \text{ to } n \text{ do} \\
\quad \quad a(2i, 3j) &= b(i,j) \\
\quad \quad b(i+1, j) &= a(i, j)
\end{align*}
```

\[ R_1 = \{ [i,j] \rightarrow [2i,3j]: 1 \leq j \& 2i \leq n \& 1 \leq i \& 3j \leq n \} \]

\[ R_2 = \{ [i,j] \rightarrow [i+1,j]: 1 \leq j < n \& 1 \leq j \leq n \} \]
Limitations of ATF

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

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

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

\[
\begin{align*}
C2 \cdot i + C1 \cdot j + C0 &= C2 \cdot 2i + C1 \cdot 3j + C0 \\
C2 \cdot i + C1 \cdot j + C0 &= C1 \cdot (i + 1) + C1 \cdot j + C0
\end{align*}
\]

\[
\begin{align*}
0 &= C2 \cdot i + C1 \cdot 2j \\
0 &= C1 \\
C1 &= 0 \\
C2 &= 0
\end{align*}
\]
Limitations of ATF

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

```plaintext
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)
```

![Diagram illustrating the pattern of loops and their dependencies](image)
Limitations of ATF

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

```plaintext
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)
```

\[ R_1 = \{ [i,j] \rightarrow [2i+2j,2j] : 1 \leq j \leq m \land 1 \leq i \land 2i+2j \leq n \} \]

\[ R_2 = \{ [i,j] \rightarrow [i,j+1] : 1 \leq i \leq n \land 1 \leq j < m \} \]
Limitations of ATF

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

\[
\begin{align*}
C_{11}i + C_{12}j + C_1 &= C_{11}(2i+2j) + C_{12}(2j) + C_1 \\
C_{11}i + C_{12}j + C_1 &= C_{11}i + C_{12}(j+1) + C_1
\end{align*}
\]

\[
\begin{align*}
(C_{11})i + (C_{12}-2C_{11})j &= 0 \\
C_{12} &= 0
\end{align*}
\]

\[
\begin{align*}
C_{12} &= 0 \\
C_{11} &= 0
\end{align*}
\]

*Limitations of the ATF motivate further research aimed at developing more advanced techniques for extracting parallelism*
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.
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).
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.
**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 $S$. 
Definition 4. A slice is independent or \textit{synchronization-free} if there is no dependence between the iterations in slice and the remaining iterations in the iteration space.
Definition 5. The *source(s) of a slice* is the ultimate dependence source(s) that this slice comprises.
Examples of slices

Dependences in loop A:

Dependences in loop B:

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

Two slices with a single ultimate source each

Two slices with multiple ultimate sources each
Modified Floyd-Warshall algorithm

**Input:** 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 \( R_{i,j} \) describes all transitive dependences between statements \( i \) and \( j \) in the SCC.
Slicing algorithm

INPUT:
Dependence relations representing an SCC

BEGIN

Find all ultimate dependence dependence sources

Find sources of slices

Find operations of each slice

OUTPUT:
Parallel code

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

END

Slicing algorithm

BEGIN

INPUT: n - dimension of loop
Set \( S = \{ R_{ij} \mid i,j \in [1,q] \} \)

Foreach relation \( R_{i,j} \in S \) 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

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

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

Find all ultimate dependence sources

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

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

Find all ultimate dependence sources

Find set, UDS, containing ultimate dependence sources:

\[
UDS := \bigcup_{R_{i,j} \in S} \text{dom} R_{i,j} - \bigcup_{R_{i,j} \in S} \text{ran} R_{i,j}
\]
BEGIN

Find all ultimate dependence sources

Find sources of slices

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

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

where $I$ is the identity relation.
BEGIN

Find all ultimate dependence sources

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:

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

Find sources of slices

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

\[ \text{Sources} := UDS - \text{range } R_{UCS} \]
for $i = 1$ to $n$ do
s1: $b(i,i) = a(i-3,i)$
for $j = 1$ to $n$ do
s2: $a(i,j) = a(i,j-1) + b(i,j)$;

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

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

$R_{2,2} := \{ [i,j] \rightarrow [i,j+1]: 1 \leq i \leq n \& 1 \leq j < n \}$ ;
for $i = 1$ to $n$ do
s1:  $b(i,i) = a(i-3,i)$
     for $j = 1$ to $n$ do
s2:  $a(i,j) = a(i,j-1) + b(i,j)$;

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

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

$R_{2,2} := \{ [i,j] \rightarrow [i,j+1]: 1 \leq i \leq n \land 1 \leq j < n \}$;
Example of parallelization

\[ R_{1,1} := \{ [i,-1,1] \rightarrow [i,j,1]: 1 \leq i \leq n \land 1 \leq j < n \} \]

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

\[ R_{2,2} := \{ [i,j,2] \rightarrow [i,j+1,2]: 1 \leq i \leq n \land 1 \leq j < n \} \]

\[ UDS := \{ [i,-1,1]: 1 \leq i \leq \min(n,3) \} \]
Example of parallelization

In order to compute 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-Warshall algorithm.

<table>
<thead>
<tr>
<th>$r$</th>
<th>$p$</th>
<th>$q$</th>
<th>Results of iterations of the Floyd-Warshall algorithm</th>
<th>Simplified results</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>$R_{1,1}' := \emptyset \cup \emptyset \circ U \circ \emptyset$</td>
<td>$R_{1,1}' := \emptyset$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>$R_{1,2}' := R_{1,2} \cup R_{1,2} \circ U \circ \emptyset$</td>
<td>$R_{1,2}' := R_{1,2}$</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>$R_{2,1}' := R_{2,1} \cup \emptyset \circ U \circ R_{2,1}$</td>
<td>$R_{2,1}' := R_{2,1}$</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>$R_{2,2}' := R_{2,2} \cup R_{1,2} \circ U \circ R_{2,1}$</td>
<td>$R_{2,2}' := R_{2,2} \cup R_{1,2} \circ R_{2,1}$</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>$R_{1,1}'' := \overline{R}<em>{1,1}^+ := \emptyset \cup R</em>{2,1}' \circ R_{2,2}'' \circ R_{1,2}'$</td>
<td>$\overline{R}<em>{1,1}^+ := R</em>{2,1} \circ (R_{2,2} \cup R_{1,2} \circ R_{2,1})^* \circ R_{1,2}$</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>$R_{1,2}'' := \overline{R}<em>{1,2}^+ := R</em>{1,2}' \circ R_{2,2}' \circ R_{2,2}'' \circ R_{1,2}'$</td>
<td>$\overline{R}<em>{1,2}^+ := R</em>{1,2} \cup (R_{2,2} \cup R_{1,2} \circ R_{2,1})^* \circ R_{1,2}$</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>$R_{2,1}'' := \overline{R}<em>{2,1}^+ := R</em>{2,1}' \circ R_{2,1}' \circ R_{2,2}'' \circ R_{2,2}'$</td>
<td>$\overline{R}<em>{2,1}^+ := R</em>{2,1} \cup R_{2,1} \circ (R_{2,2} \cup R_{1,2} \circ R_{2,1})^*$</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>$R_{2,2}'' := \overline{R}<em>{2,2}^+ := R</em>{2,2}' \circ R_{2,2}' \circ R_{2,2}'' \circ R_{2,2}'$</td>
<td>$\overline{R}<em>{2,2}^+ := (R</em>{2,2} \cup R_{1,2} \circ R_{2,1})^*$</td>
</tr>
</tbody>
</table>

Results of iterations of the Floyd-Warshall Algorithm
Example of parallelization

Next, we compute $R^*$ as follows:

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

\[
\{[i,-1,1] \rightarrow [i',-1,1] : \exists (\alpha : i' = i+3\alpha \& 1 \leq i \leq i' - 3 \& i' \leq n)\} \cup \{[i,-1,1] \rightarrow [i,j',2] : 1 \leq i \leq n \& 1 \leq j' \leq n \& 4 \leq n\} \cup \{[i,-1,1] \rightarrow [i,j',2] : \exists (\alpha : i+3\alpha = i' \& 1 \leq i \leq i' - 3 \& 1 \leq j' \leq n \& i' \leq n)\} \cup \{[i,-1,1] \rightarrow [i,j',2] : j', i \leq n \leq 3 \& 1 \leq i \& 1 \leq j'\} \cup \{[i,i+3,2] \rightarrow [i+3,-1,1] : 1 \leq i \leq n - 3\} \cup \{[i,j,2] \rightarrow [i+3,-1,1] : 1 \leq j \leq n - 3 \& 1 \leq j\} \cup \{[i,j,2] \rightarrow [i,j',2] : 1 \leq j \leq j' \leq n \& 1 \leq i \leq n\} \cup \{[i,j,In_3] \rightarrow [i,j,In_3]\}.
\]

$$R_{UCS} := \emptyset$$

$$Sources := UDS - R_{UCS} = \{[I,-1,1] : 1 \leq i \leq \min(n,3)\}$$
if Sources ≠ ∅ then
    genLoops (in: Sources;
               out: OuterLoops, L_I);
    foreach I in L_I do
        S_Slice := R* (R_UCS*(I))
        // 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

<table>
<thead>
<tr>
<th>Source</th>
</tr>
</thead>
</table>
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:

\[
\text{for}(t1=1; t1<=\text{min}(n,3); t1++) \quad s1(t1,-1,1);
\]

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 $l=(t1,-1,1)'$.

Find set $S\_Slice := R\ast (R\_UCS\ast (I)) =$

\[
\{[i,-1,1]: (\alpha : i = t1+3\alpha \& 1 \leq t1 \leq i-3 \& i \leq n)\} \cup \{[t1,j,2]: 1 \leq t1 \leq n \& 1 \leq j \leq n \& 4 \leq n\} \cup \{[i,j,2]: (\alpha : i = t1+3\alpha \& 1 \leq t1 \leq i-3 \& 1 \leq j \leq n \& i \leq n)\} \cup \{[t1,j,2]: 1 \leq t1 \leq i-3 \& 1 \leq j \leq n\} \cup \{[i,-1,1]: i = t1\}.
\]

Applying the Omega code generator to set $S\_Slice$, we yield the inner loops.

\[
s(t1,-1,1);
\]

\[
\text{for}(t2 = 1; t2 <= n; t2++)
\]

\[
s(t1,t2,2);
\]

\[
\text{for}(t3 = t1+3; t3<=n; t3+=3) \{
\]

\[
s(t3,-1,1);
\]

\[
\text{for}(t2 = 1; t2 <= n; t2++)
\]

\[
s(t3,t2,2);
\]

\} 

List $L\_J$ contains single vectors:

$(t1,-1,1)', (t1,t2,2)', (t3,-1,1)' \text{ and } (t3,t2,2)'$.
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

```plaintext
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)
```

```plaintext
1    2     3   4    5    6    7    8    i
```

```plaintext
1    2     3   4    5    6    7    8    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
    $s_1$: $a(i,j) = b(i,j) + c(i,j)$
    $s_2$: $c(i,j-1) = a(i,j+1)$
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
\[ a(i) = a(2i) \]

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

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:

\[ R^+ = \{ [i] \rightarrow [j] : \exists k ( k \geq 1 \land j = 2^k i \land 1 \leq i, j \leq n ) \} \]
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)
    end for
end parfor
```

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

```
parfor j = 1 to n do
  for i = 1 to n do
    a(i,j) = a(i,j-1)
  endfor
endfor
```
Fine-grained parallelism

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

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

i

j

parfor j = 1 to n do
  a(i,j) = a(i,j-1)
end
```
Further research

- Development of approaches to extract slices requiring synchronization

```plaintext
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)
```

Diagram:

```
1   2   3   4   5   6
```

```
i
1  2  3  4  5  6
```

```
1  2  3  4  5  6
```

```
1  2  3  4  5  6
```
Further research

- Development of approaches to extract slices requiring *synchronization*

```plaintext
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
    $a(i,j)=a(2i+2j,2j)+a(i,j-1)$
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

```latex
\begin{align*}
  &\text{for } i=1 \text{ to } n \text{ do} \\
  &\quad \text{for } j=1 \text{ to } m \text{ do} \\
  &\quad \quad a(i,j) = a(2i+2j,2j) + a(i,j-1)
\end{align*}
```

- 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!