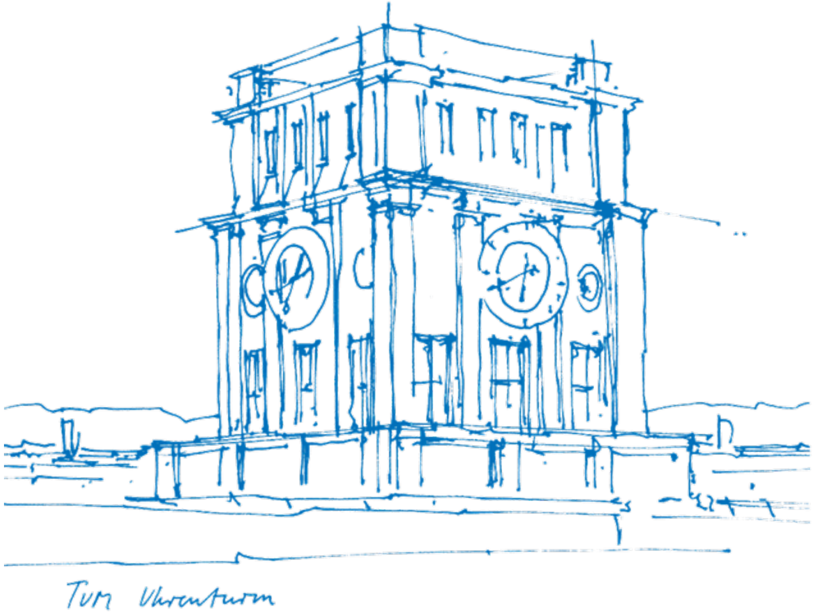
Parallel Programming Tutorial - Dependency and transformations

Amir Raoofy, M.Sc.
Chair for Computer Architecture and Parallel Systems (Prof. Schulz)
Technical University of Munich
23. Mai 2018



Few organizational notes

Organization

- The speedup requirement for assignment 6 is relaxed to 14.0 since you don't access to the server environment.
- Slides from last tutorial are updated, please download the latest version.
- Please evaluate the course, the date will be announced.
- Don't forget that On June 25th we will have a lecture on optimization of sequential programs by M.Sc. Alexis Engelke
- Assignment on SIMD is not yet ready. Hopefully will be published by the end of the week
- Assignment 7 will be published today by evening

Solution for Assignment 5

Solution for Assignment 5 - Sections

- Parallelize the familytree algorithm with OpenMP sections
- Set the correct number of threads
  - use the runtime function `omp_set_num_threads`
- Enable nested parallelism
  - use the runtime function `omp_set_nested`
- Avoid oversubscription
  - Option 1: Use conditional parallelism by utilizing `omp_get_level`
  - Option 2: Use `omp_set_max_active_levels`

```c
#define MAX_NESTING_LEVEL 5

void parallel_traverse(tree *node) {
  if (node != NULL) {
    node->IQ = compute_IQ(node->data);
    genius[node->id] = node->IQ;

    #pragma omp parallel sections {
    #pragma omp section
    parallel_traverse(node->right);
    #pragma omp section
    parallel_traverse(node->left);
    }
  }
}

void traverse(tree *node, int numThreads) {
  omp_set_num_threads( numThreads );
  omp_set_nested( 1 );
  omp_set_max_active_levels( MAX_NESTING_LEVEL );

  parallel_traverse(node);
}
```

Solution for Assignment 5 - Tasks

- Parallelize the familytree algorithm with OpenMP tasks
- Set the correct number of threads
  - use the clause num_threads
- Use a single thread to start the algorithm
- Optimization: Create only one task per recursion

```c
void parallel_traverse(tree *node) {
  if (node != NULL) {

    #pragma omp task
    parallel_traverse(node->right);
    parallel_traverse(node->left);

    node->IQ = compute_IQ(node->data);
    genius[node->id] = node->IQ;
  }
}

void traverse(tree *node, int numThreads) {
  #pragma omp parallel num_threads( numThreads )
  {
    #pragma omp single
    parallel_traverse(node);
  }
}
```

(Data) Dependency Analysis

Dependence Notation

- S1 and S2 are statements

Type	Meaning	Symbol	Alternative Symbols	Example
True dependence	RAW	S1 δ^t S2	δ, δ^f	S1: `x=1` S2: `y=x`
Antidependence	WAR	S1 δ^a S2	δ^{-1}	S1: `y=x` S2: `x=1`
Output dependence	WAW	S1 δ^o S2		S1: `x=1` S2: `x=2`

It's called true dependence, because for second instruction you need the first one to finish.

- RAW = "read after write"
- WAR = "write after read"
- WAW = "write after write"

Iteration Vector

```
for (i1 = 1; i1 < N1; i1++) {
    for (i2 = 1; i2 < N2; i2++) {
        ...
        for(in = 1; in < Nn; in++) {
            S:        ...
        }
        ...
    }
}
```

- The iteration vector for a statement S in the loop is given by $\vec{i} := (i_1, i_2, \dots, i_n)$ where i_k, $(1 \leq \mathtt{k} \leq n)$, represents the iteration number for the loop at nesting level `k`.
- The set of all possible iteration vectors for S is called *iteration space*.

Iteration Vector - Example

```
for (i = 1; i < 3; i++) {
    for (j = 1; j < 4; j++) {
        S:  ...
    }
}
```

- The iteration space of statement S is $\{(1,1),(1,2),(1,3),(2,1),(2,2),(2,3)\}$

Data Dependence

Informal Definition

There is a data dependence from statement S_1 to statement S_2 (S_2 dependes on S_1), if and only if (1) both statements access the same memory location and at least one of them writes to it, and (2) there is a feasible run-time execution path from S_1 to S_2.

Formal Definition

$\exists M, S_1, S_2, \vec{i}, \vec{j}$:

1. ($\vec{i} < \vec{j}$)[1]or ($\vec{i} = \vec{j}$) [2][3]and there is a path from S_1 to S_2
2. S_1 and S_2 access M on $\vec{i}$ and $\vec{j}$, respectively
3. One of these accesses is a write

[1]called *loop-carried dependence*

[2]called *loop-independent dependence*

[3]The operations $<$ and $=$ are defined componentwise from left to right.

Distance Vector

Definition

- Suppose there is a dependence from statement S_1 on iteration $\vec{i}$ of a loop nest to statement S_2 on iteration $\vec{j}$
- The distance vector is defined as $d(\vec{i}, \vec{j}) = \left[d(\vec{i}, \vec{j})_1, \ldots, d(\vec{i}, \vec{j})_N\right]$,
  where $d(\vec{i}, \vec{j})_k := j_k - i_k$.

Example

The distance vector for the dependence $S[(2,2,2)]\ \delta^t\ S[(3,1,2)]$ of the following loop nest is $(1,-1,0)$.

```
for (i = 1; i < N; i++) {
   for (j = 1; j < M; j++) {
      for (k = 1; k < L; k++) {
         S:    A(i + 1, j - 1, k) = A(i,j,k)
      }
   }
}
```

Direction Vector

Definition

- Suppose there is a dependence from statement S_1 on iteration $\overrightarrow{i}$ of a loop nest to statement S_2 on iteration $\overrightarrow{j}$
- Direction vector $D(\overrightarrow{i}, \overrightarrow{j})_k := \begin{cases} \text{“}<\text{”}, & d(i,j)_k > 0 \\ \text{“}=\text{”}, & d(i,j)_k = 0 \\ \text{“}>\text{”}, & d(i,j)_k < 0 \end{cases}$

Example

The direction vector for the dependence $S[(2,2,2)]\ \delta^t\ S[(3,1,2)]$ of the following example is $(<,>,=)$.

```
for (i = 1; i < N; i++) {
   for (j = 1; j < M; j++) {
      for (k = 1; k < L; k++) {
         S:    A(i + 1,j - 1,k) = A(i,j,k)
      }
   }
}
```

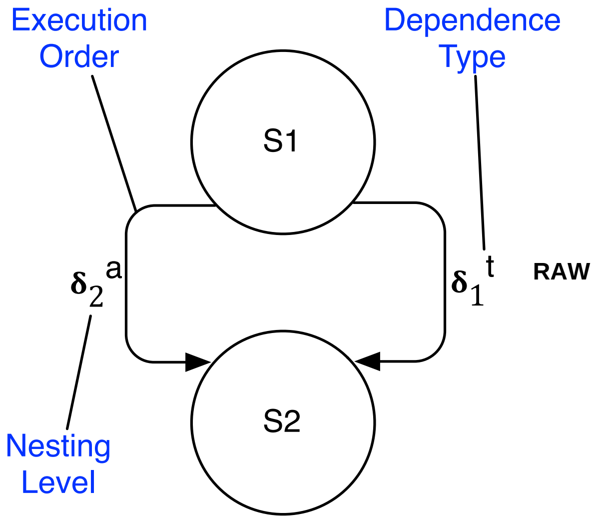
The **level** of a loop-carried dependence is the index of the leftmost non-“$=$” of $D(i,j)$.

Dependence Graphs

- Nodes: The statements of a program
- Edges: The dependences between the statements from the first executed statement to the following one
- Each edge is labeled with the dependence type and the nesting level

Example

```
for (i = 1; i < N; i++) {
    for (j = 1; j < M; j++) {
        S1:    A(i + 1,j) = B(i,j + 1)
        S2:    B(i,j) = A(i,j)
    }
}
```



Example 1

- Give the dependence graph for the following loop.

```
for (i = 0; i < N; i++) {
    S1:  B(i) = A(i)
    S2:  A(i) = A(i) + B(i + 1)
    S3:  C(i) = 2 * B(i)
}
```

- Give the distance and direction vectors for the loop-carried dependencies.

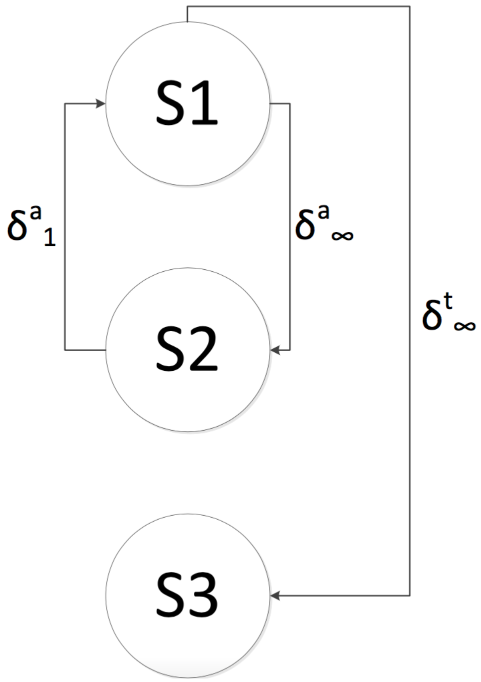
Source	Sink	Dep.Type	Dist. Vector	Dir. Vector	
...	...	...	...	...	...

- Source, Sink: Specify the references in the form *S1:B(i)*
- Type: Loop-independent (l-i) or loop-carried dependence (l-c)
- Dep.Type: True-, Anti-, or Output-Dependence
- Vectors: n-Tuples where n is the depth of the loop nest

Solution for Example 1

```
for (i = 0; i < N; i++) {
    S1:  B(i) = A(i)
    S2:  A(i) = A(i) + B(i + 1)
    S3:  C(i) = 2 * B(i)
}
```

Source	Sink	Dep.Type	Dist. Vector	Dir. Vector
S2: B(i + 1)	S1: B(i)	a	(1)	(<)



Example 2

- Give the dependence graph for the following loop.

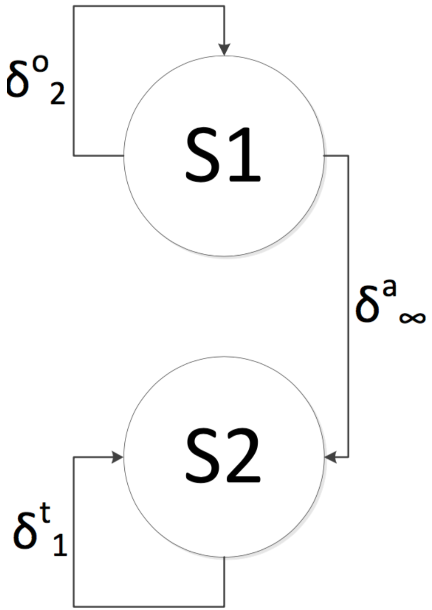
```
for (i = 1; i < N; i++) {
    for (j = 1; j < M; j++) {
        S1:   A(i)   = B(i,j)
        S2:   B(i,j) = B(i - 1,2 * j)
    }
}
```

- Give the distance and direction vectors for the dependencies.

Solution for Example 2

```
for (i = 1; i < N; i++) {
    for (j = 1; j < M; j++) {
        S1:   A(i)   = B(i,j)
        S2:   B(i,j) = B(i - 1,2 * j)
    }
}
```

Source	Sink	Dep.Type	Dist. Vector	Dir. Vector
S1: A(i)	S1: A(i)	o	(0,*)	(=, *)
S2: B(i, j)	S2: B(i-1, 2*j)	t	(1,-j)	(<, >)



Example 3

- Give the dependence graph for the following loop.

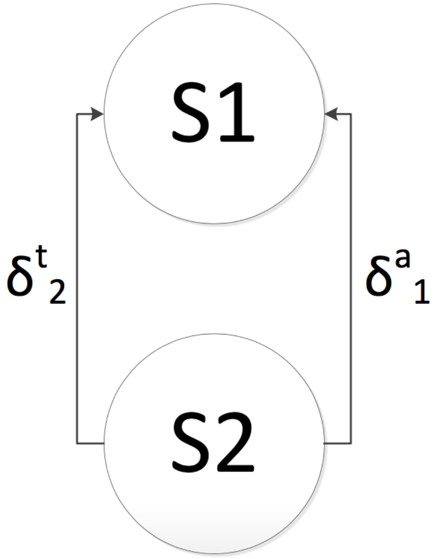
```
for (i = 0; i < N; i++) {
    for (j = 0; j < M; j++) {
        S1:   B(i - 1,j) = C(i,j - 2)
        S2:   C(i,j)    = 2 * B(i,j + 1)
    }
}
```

- Give the distance and direction vectors for the loop-carried dependencies.

Solution for Example 3

```
for (i = 0; i < N; i++) {
    for (j = 0; j < M; j++) {
        S1:   B(i - 1,j) = C(i,j - 2)
        S2:   C(i,j)     = 2 * B(i,j + 1)
    }
}
```

Source	Sink	Dep.Type	Dist. Vector	Dir. Vector
S2: B(i, j+1)	S1: B(i-1, j)	a	(1,1)	$(<, <)$
S2: C(i, j)	S1: C(i, j-2)	t	(0,2)	$(=, <)$



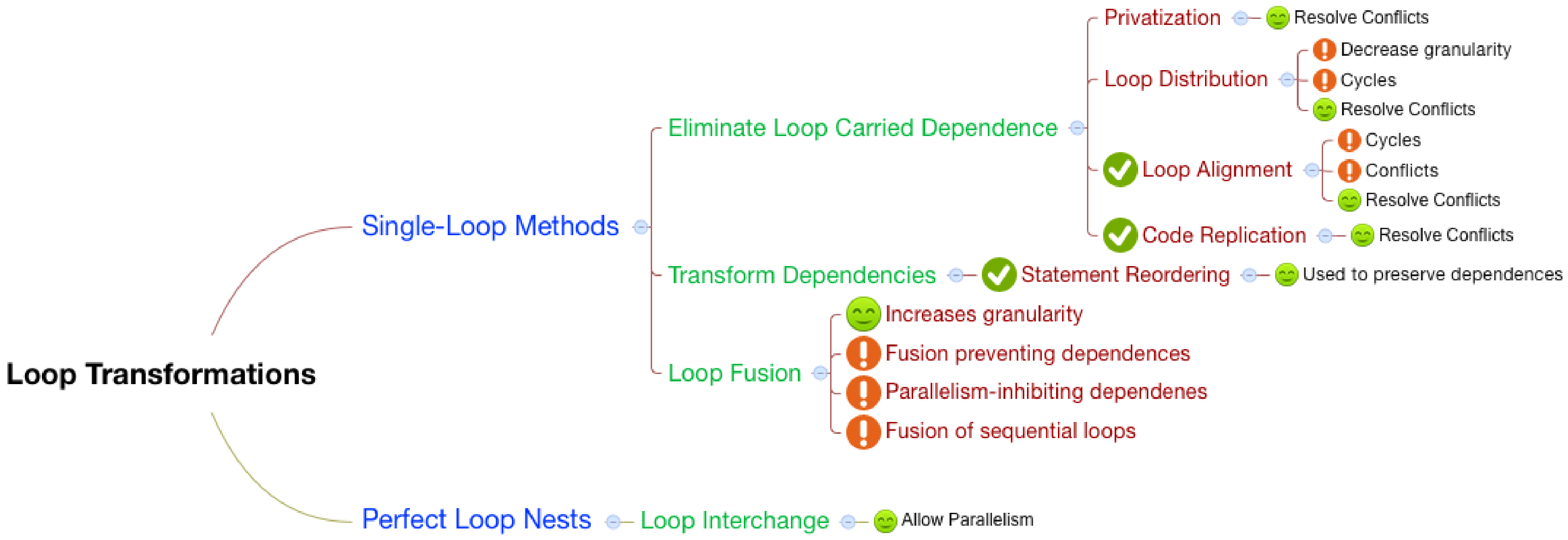
Loop Transformations

Transformations

Theorem

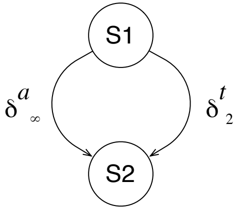
Any reordering transformation that preserves every dependence in a program preserves the meaning of that program.

Transformations - Mindmap



Loop Distribution I

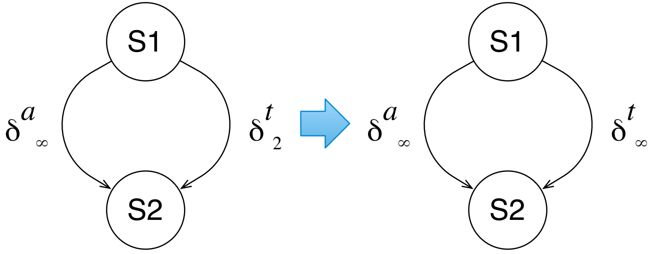
```
for (i=1; i<n; i++) {
    for (j=1; j<m; j++) {
        S1:   A(i,j) = B(i,j)
        S2:   B(i,j) = A(i,j-1)
    }
}
```



Loop Distribution I

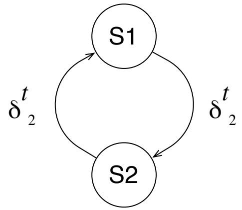
```
for (i=1; i<n; i++) {
    for (j=1; j<m; j++) {
        S1:   A(i,j) = B(i,j)
        S2:   B(i,j) = A(i,j-1)
    }
}
```

```
for (i=1; i<n; i++) {
    for (j=1; j<m; j++) {
        S1:   A(i,j) = B(i,j)
    }
    for (j=1; j<m; j++) {
        S2:   B(i,j) = A(i,j-1)
    }
}
```



Loop Distribution II - Cycle

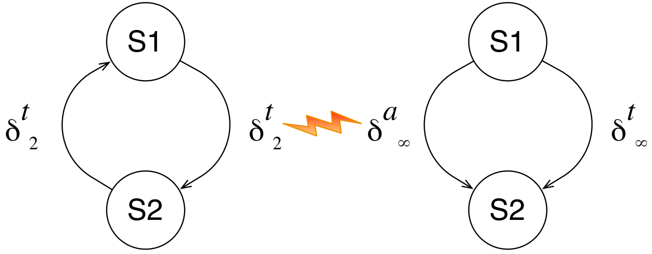
```
for (i=1; i<n; i++) {
    for (j=1; j<m; j++) {
        S1:   A(i,j) = B(i,j)
        S2:   B(i,j+1) = A(i,j-1)
    }
}
```



Loop Distribution II - Cycle

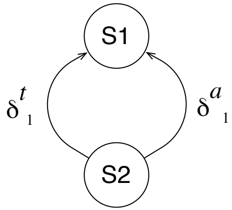
```
for (i=1; i<n; i++) {
    for (j=1; j<m; j++) {
        S1:   A(i,j) = B(i,j)
        S2:   B(i,j+1) = A(i,j-1)
    }
}
```

```
for (i=1; i<n; i++) {
    for (j=1; j<m; j++) {
        S1:   A(i,j) = B(i,j)
    }
    for (j=1; j<m; j++) {
        S2:   B(i,j+1) = A(i,j-1)
    }
}
```



Loop Alignment I

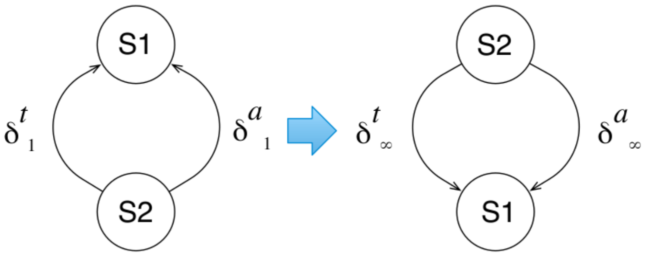
```
for (i=1; i<n; i++) {
    S1:   A(i)   = B(i)
    S2:   B(i+1) = A(i+1)
}
```



Loop Alignment I

```
for (i=1; i<n; i++) {
    S1:   A(i)   = B(i)
    S2:   B(i+1) = A(i+1)
}
```

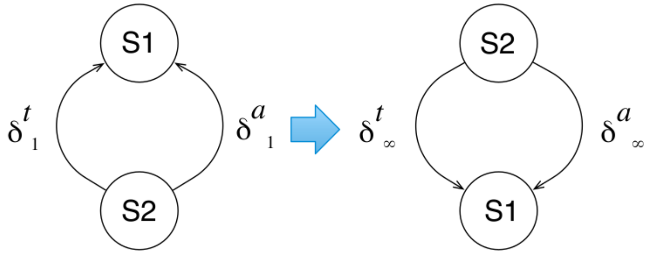
```
for (i=1; i<n; i++) {
    S1:   A(i)   = B(i)
    S2:   B(i+1) = A(i+1)
}
```



Loop Alignment I - Peeling Off Executions

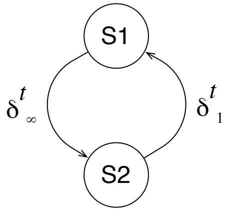
```
for (i=1; i<n; i++) {
    S1:   A(i)   = B(i)
    S2:   B(i+1) = A(i+1)
}
```

```
A(1) = B(1)
for (i=2; i<n; i++) {
    S2:   B(i) = A(i)
    S1:   A(i) = B(i)
}
B(n) = A(n)
```



Loop Alignment II - Cycle

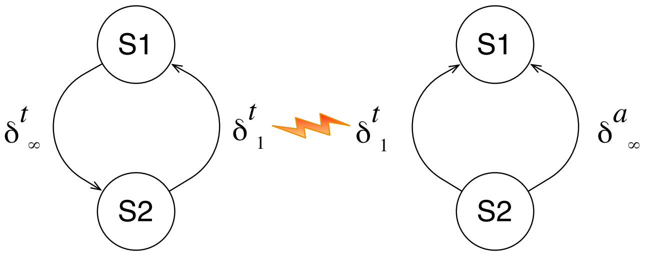
```
for (i=1; i<n; i++) {
    S1:   A(i)   = B(i)
    S2:   B(i+1) = A(i)
}
```



Loop Alignment II - Cycle

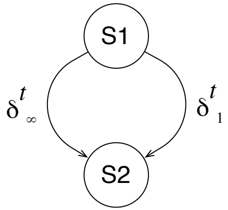
```
for (i=1; i<n; i++) {
    S1:   A(i)   = B(i)
    S2:   B(i+1) = A(i)
}
```

```
for (i=1; i<n+1; i++) {
    S1:   if (i<n) A(i) = B(i)
    S2:   if (i>1) B(i) = A(i-1)
}
```



Loop Alignment III - Conflict

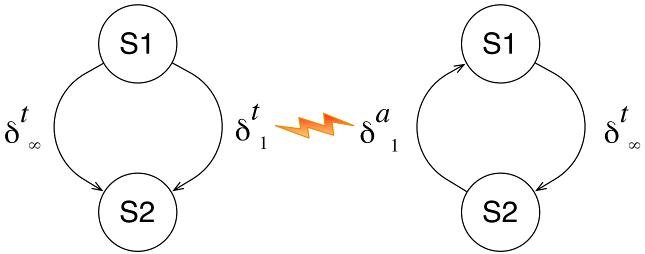
```
for (i=1; i<n; i++) {
    S1: A(i) = B(i)
    S2: C(i) = A(i) + A(i-1)
}
```



Loop Alignment III - Conflict

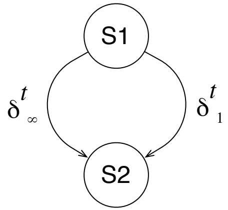
```
for (i=1; i<n; i++) {
    S1: A(i) = B(i)
    S2: C(i) = A(i) + A(i-1)
}
```

```
for (i=0; i<n; i++) {
    S1: if (i>0)   A(i)   = B(i)
    S2: if (i<n+1) C(i+1) = A(i+1)+A(i)
}
```



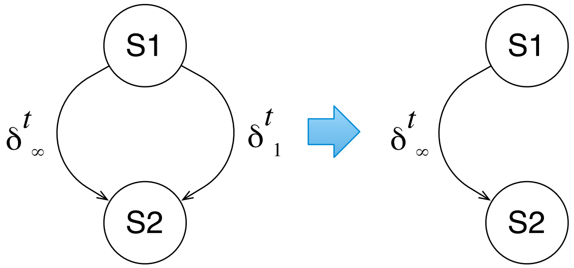
Code Replication

```
for (i=1; i<n; i++) {
    S1:   A(i) = B(i)
    S2:   C(i) = A(i) + A(i-1)
}
```



Code Replication

```
for (i=1; i<n; i++) {
    S1:   A(i) = B(i)
    S2:   C(i) = A(i) + A(i-1)
}
```

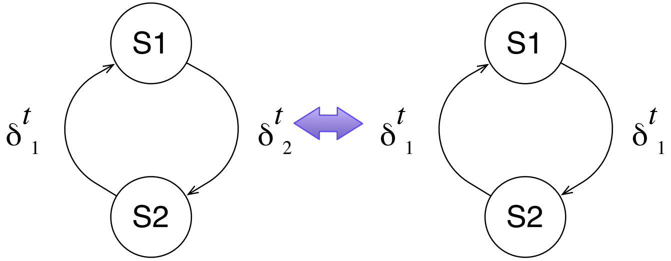



```
for (i=1; i<n; i++) {
    private(T)
    S1:   A(i)    = B(i)
        if (i=1) T = A(0)
        else     T = B(i-1)
    S2:   C(i) = A(i) + T
}
```

Statement Reordering

```
for (i=1; i<10; i++) {
    S1:   A(i+1) = F(i)
    S2:   F(i+1) = A(i)
}
```

```
for (i=1; i<10; i++) {
    S2:   F(i+1) = A(i)
    S1:   A(i+1) = F(i)
}
```



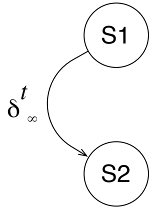
Transformations

Theorem

Alignment, replication, and statement reordering are sufficient to eliminate all carried dependences in a single loop that contains no recurrence and in which the distance of each dependence is a constant independent of the loop index.

Loop Fusion I

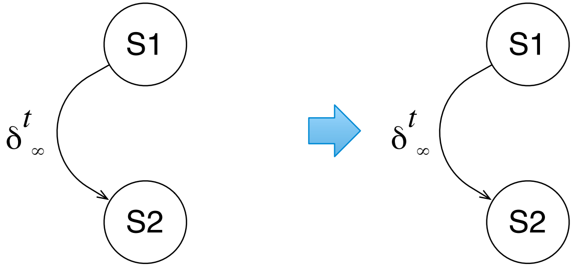
```
for (i=1; i<n; i++) {
    S1:   A(i) = B(i+1)
}
for (i=1; i<n; i++) {
    S2:   C(i) = A(i) + B(i)
}
```



Loop Fusion I

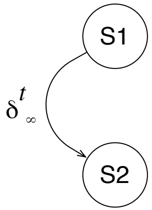
```
for (i=1; i<n; i++) {
    S1:   A(i) = B(i+1)
}
for (i=1; i<n; i++) {
    S2:   C(i) = A(i) + B(i)
}
```

```
for (i=1; i<n; i++) {
    S1:   A(i) = B(i+1)
    S2:   C(i) = A(i) + B(i)
}
```



Loop Fusion II - Fusion preventing Dependency

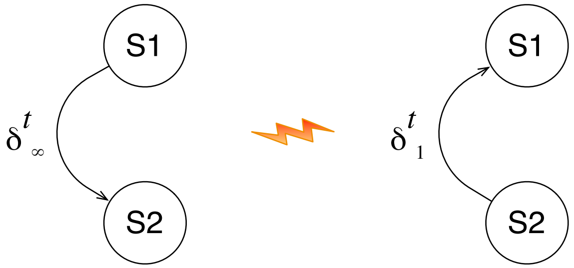
```
for (i=1; i<n; i++) {
    S1:   A(i) = B(i+1)
}
for (i=1; i<n; i++) {
    S2:   C(i) = A(i+1) + B(i)
}
```



Loop Fusion II - Fusion preventing Dependency

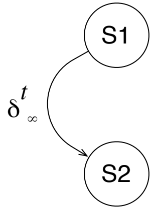
```
for (i=1; i<n; i++) {
    S1:   A(i) = B(i+1)
}
for (i=1; i<n; i++) {
    S2:   C(i) = A(i+1) + B(i)
}
```

```
for (i=1; i<n; i++) {
    S1:   A(i) = B(i+1)
    S2:   C(i) = A(i+1) + B(i)
}
```



Loop Fusion III - Parallelism inhibiting Dependency

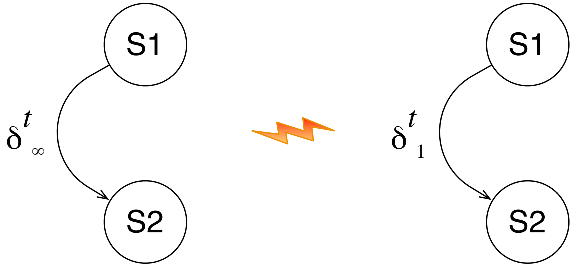
```
for (i=1; i<n; i++) {
    S1:   A(i+1) = B(i+1)
}
for (i=1; i<n; i++) {
    S2:   C(i) = A(i) + B(i)
}
```



Loop Fusion III - Parallelism inhibiting Dependency

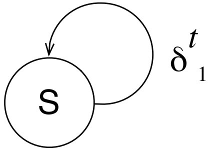
```
for (i=1; i<n; i++) {
    S1:   A(i+1) = B(i+1)
}
for (i=1; i<n; i++) {
    S2:   C(i) = A(i) + B(i)
}
```

```
for (i=1; i<n; i++) {
    S1:   A(i+1) = B(i+1)
    S2:   C(i)   = A(i) + B(i)
}
```

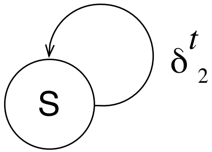


Loop Interchange

```
for (i=1; i<n; i++) {
    for(j=1; j<m; j++) {
        S:    A(i+1,j) = A(i,j) + B(i,j)
    }
}
```




```
for (j=1; j<m; j++) {
    for(i=1; i<n; i++) {
        S:    A(i+1,j) = A(i,j) + B(i,j)
    }
}
```



Assignment 7

Assignment 7: Loop Transformations (will be updated)

- Apply loop fusion to the loop in `loop_fusion_seq.c`
- Parallelize the loop with OpenMP in `loop_fusion_par.c` and upload it