CS 610: Loop Transformations

Swarnendu Biswas

Semester 2020-2021-I

CSE, IIT Kanpur

Content influenced by many excellent references, see References slide for acknowledgements.

Copyright Information

Enhancing Program Performance

Fundamental issues

- Adequate fine-grained parallelism
  - Exploit vector instruction sets (SSE, AVX, AVX-512)
  - Multiple pipelined functional units in each core
- Adequate parallelism for SMP-type systems
  - Keep multiple asynchronous processors busy with work
- Minimize cost of memory accesses

Role of a Good Compiler

Try and extract performance automatically

Optimize memory access latency

- **Code restructuring optimizations**
- Prefetching optimizations
- Data layout optimizations
- Code layout optimizations

Loop Optimizations

- Loops are one of most commonly used constructs in HPC program
- Compiler performs many of loop optimization techniques automatically
  - In some cases source code modifications enhance optimizer’s ability to transform code

Reordering Transformations

- A reordering transformation does not add or remove statements from a loop nest
  - Only reorders the execution of the statements that are already in the loop

Do not add or remove statements → Do not add or remove any new dependences

Reordering Transformations

- A reordering transformation does not add or remove statements from a loop nest
  - Only reorders the execution of the statements that are already in the loop

A reordering transformation is valid if it preserves all existing dependences in the loop

Iteration Reordering and Parallelization

- A transformation that reorders the iterations of a level-*k* loop, without making any other changes, is valid if the loop carries no dependence

- Each iteration of a loop may be executed in parallel if it carries no dependences

Data Dependence Graph and Parallelization

- If the Data Dependence Graph (DDG) is acyclic, then vectorization of the program is possible and is straightforward
- Otherwise, try to reduce the DDG to an acyclic graph

Enhancing Fine-Grained Parallelism

Focus on Parallelization of Inner Loops

System Setup

- Setup
  - Vector or superscalar architectures
  - Focus is mostly on parallelizing the inner loops
- We will see optimizations for coarse-grained parallelism later

Loop Interchange (Loop Permutation)

- Switch the nesting order of loops in a **perfect** loop nest
- Can increase parallelism, can improve spatial locality

```
   DO I = 1, N
     DO J = 1, M
S      A(I,J+1) = A(I,J) + B
     ENDDO
   ENDDO
```

(0, 1) → (0, -1)

- Dependence is now carried by the outer loop
- Inner-loop can be vectorized

```
   DO J = 1, M
     DO I = 1, N
S      A(I,J+1) = A(I,J) + B
     ENDDO
   ENDDO
```

(0, 1)

DO J = 1, M

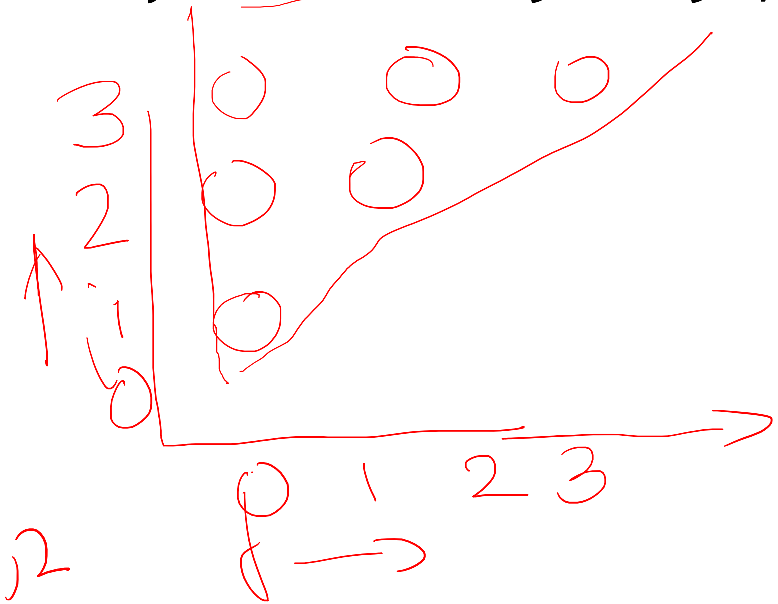
A(1:N, J+1) = A(1:N, J) + B

Interchange of Non-rectangular Loops

```
for (i=0; i<n; i++)
  for (j=0; j<i; j++)
    y[i] = y[i] + A[i][j]*x[j];
```

???

i	j
0	
1	0
2	0, 1
3	0, 1, 2



Interchange of Non-rectangular Loops

```
for (i=0; i<n; i++)
  for (j=0; j<i; j++)
    y[i] = y[i] + A[i][j]*x[j];
```

```
for (j=0; j<n; j++)
  for (i=j+1; i<n; i++)
    y[i] = y[i] + A[i][j]*x[j];
```

Validity of Loop Interchange

- Construct direction vectors for all possible dependences in the loop
  - Also called a direction matrix
- Compute direction vectors after permutation
- Permutation of the loops in a perfect nest is legal iff there are no "-" direction as the leftmost non–"0" direction in any direction vector

Legality of Loop Interchange

(0, 0)

- Dependence is loop-independent

(0, +)

- Dependence is carried by the j^{th} loop, which remains the same after interchange

(+, 0)

- Dependence is carried by the i^{th} loop, relations do not change after interchange

(+, +)

- Dependence relations remain positive in both dimensions

Legality of Loop Interchange

(+, -)

- Dependence is carried by i^{th} loop, interchange results in an illegal direction vector

(0, +)

- Dependence is carried by the j^{th} loop, which remains the same after interchange

(0, -) (-, *)

- Such direction vectors are illegal, should not appear in the original loop

Invalid Loop Interchange

```
do i = 1, n
  do j = 1, n
    C(i, j) = C(i+1, j-1)
  enddo
enddo
```

```
do j = 1, n
  do i = 1, n
    C(i,j) = C(i+1,j-1)
  enddo
enddo
```

anti

(+1, -1)

i j

Why?

(-1, +1)

Validity of Loop Interchange

- Loop interchange is valid for a 2D loop nest if none of the dependence vectors has any negative components
- Interchange is legal: (1,1), (2,1), (0,1), (3,0)
- Interchange is not legal: (1,-1), (3,-2)

Valid or Invalid Loop Interchange?

```
DO J = 1, M
  DO I = 1, N
    A(I,J+1) = A(I+1,J) + B
  ENDDO
ENDDO
```

Validity of Loop Permutation

- Generalization to higher-dimensional loops
- Permute all dependence vectors exactly the same way as the intended loop permutation
- If any permuted vector is lexicographically negative, permutation is illegal
- Example: d1 = (1,-1,1) and d2 = (0,2,-1)
  - ijk -> jik? (1,-1,1) -> (-1,1,1): illegal
  - ijk -> kij? (0,2,-1) -> (-1,0,2): illegal
  - ijk -> ikj? (0,2,-1) -> (0,-1,2): illegal
  - No valid permutation:
    - j cannot be outermost loop (-1 component in d1)
    - k cannot be outermost loop (-1 component in d2)

Valid or Invalid Loop Interchange?

```
DO I = 1, N
  DO J = 1, M
    DO K = 1, L
      A(I+1,J+1,K) = A(I,J,K) + A(I,J+1,K+1)
    ENDDO
  ENDDO
ENDDO
```

$(1,1,0) \rightarrow$ flow

$(1,0,-1)$

$$\begin{array}{c} ijk \end{array} \begin{bmatrix} + & + & 0 \\ + & 0 & - \end{bmatrix}$$

(1) ikj ✓

(2) jki ✗

(3) jik ✓

Benefits from Loop Permutation

```
for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    for (k=0; k<n; k++)
      C[i][j] += A[i][k]*B[k][j];
```

	ikj	kij	jik	ijk	jki	kji
C[i][j]	1	1	0	0	n	n
A[i][k]	0	0	1	1	n	n
B[k][j]	1	1	n	n	0	0

Does Loop Interchange Always Help?

```
do i = 1, 10000
   do j = 1, 1000
      a[i] = a[i] + b[j,i] * c[i]
   end do
 end do
```

do j = 1, ~~100~~ 10000
do i = 1, 1000
a[i] = a[i] + b[j,i]
* c[i]

Understanding Loop Interchange

Pros

- Goal is to improve locality of reference or allow vectorization

Cons

- Need to careful about the iteration order, order of array accesses, and data involved

Loop Shifting

- In a perfect loop nest, if loops at level *i, i*+1,…, *i+n* carry no dependence—that is, all dependences are carried by loops at level less than *i* or greater than *i+n*—it is always legal to shift these loops inside of loop *i+n*+1.
- These loops will not carry any dependences in their new position.

Loops i to i+n

Dependence carried by outer loops

+	0	+	0	0	0
0	+	-	+	+	0
0	0	0	0	+	+
0	0	0	0	0	+

Dependence carried by inner loops

Loop Shift for Matrix Multiply

```
      DO I = 1, N
        DO J = 1, N
          DO K = 1, N
S           A(I,J) = A(I,J) + B(I,K)*C(K,J)
          ENDDO
        ENDDO
      ENDDO
```

(0, 0, <)

Could we perform loop shift?

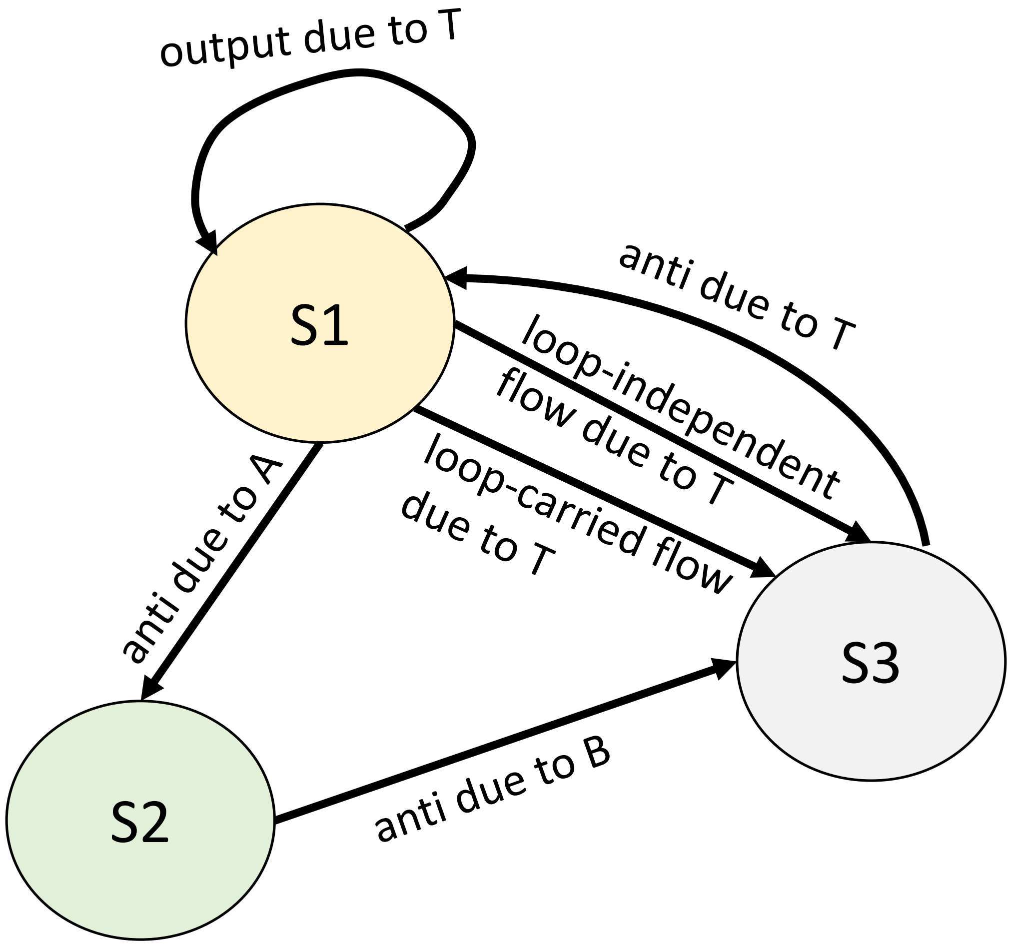
Loop Shift for Matrix Multiply

```
    DO I = 1, N
      DO J = 1, N
        DO K = 1, N
S          A(I,J) = A(I,J) + B(I,K)*C(K,J)
        ENDDO
      ENDDO
    ENDDO
```

```
    DO K = 1, N
         DO I = 1, N
           DO J = 1, N
S             A(I,J) = A(I,J) + B(I,K)*C(K,J)
           ENDDO
         ENDDO
       ENDDO
```

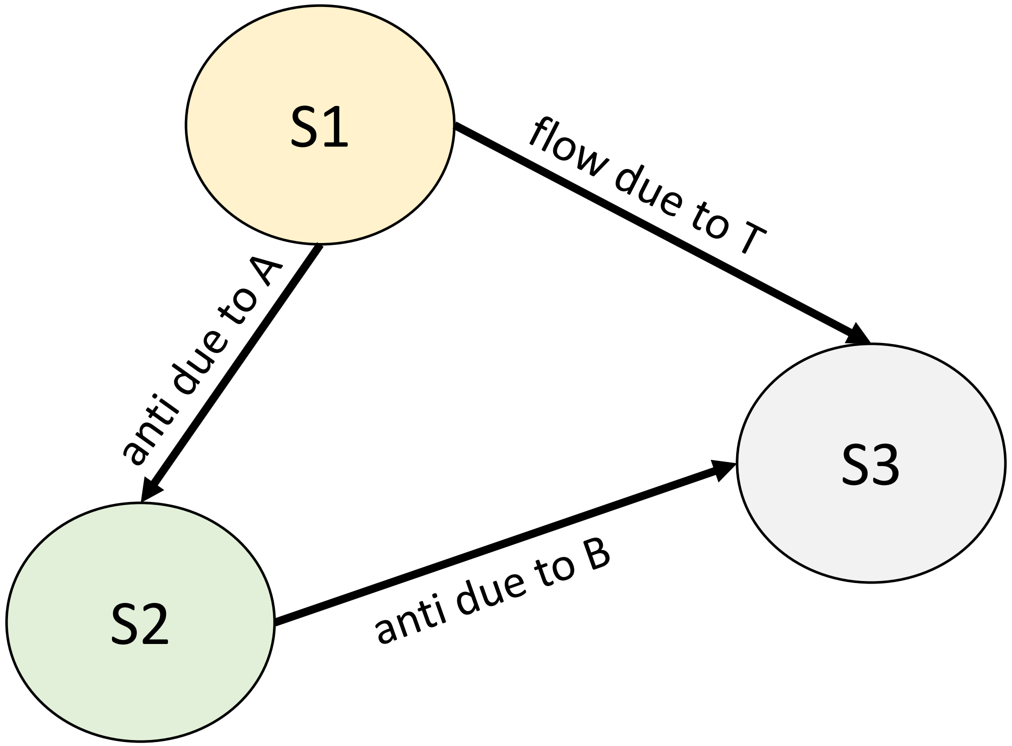
Scalar Expansion

```
    DO I = 1, N
S1    T = A(I)
S2    A(I) = B(I)
S3    B(I) = T
    ENDDO
```



Scalar Expansion

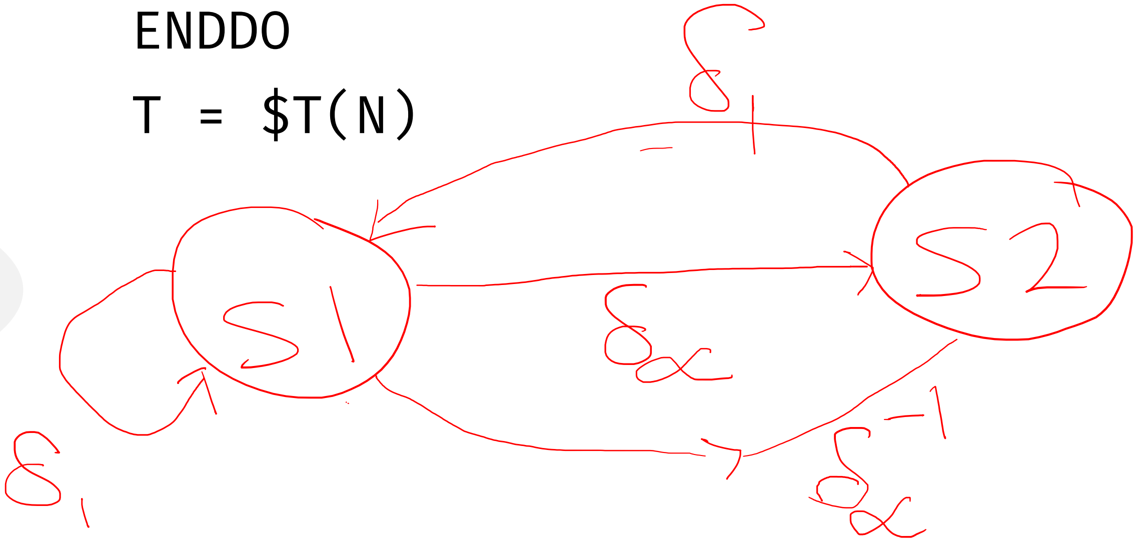
```
      DO I = 1, N
S1      $T(I) = A(I)
S2      A(I) = B(I)
S3      B(I) = $T(I)
      ENDDO
      T = $T(N)
```



Scalar Expansion

```
DO I = 1, N
  T = T + A(I) + A(I-1)
  A(I) = T
ENDDO
```

```
$T(0) = T
DO I = 1, N
  $T(I) = $T(I-1) + A(I) + A(I-1)
  A(I) = $T(I)
ENDDO
T = $T(N)
```



Can we parallelize the I loop?

Understanding Scalar Expansion

Pros

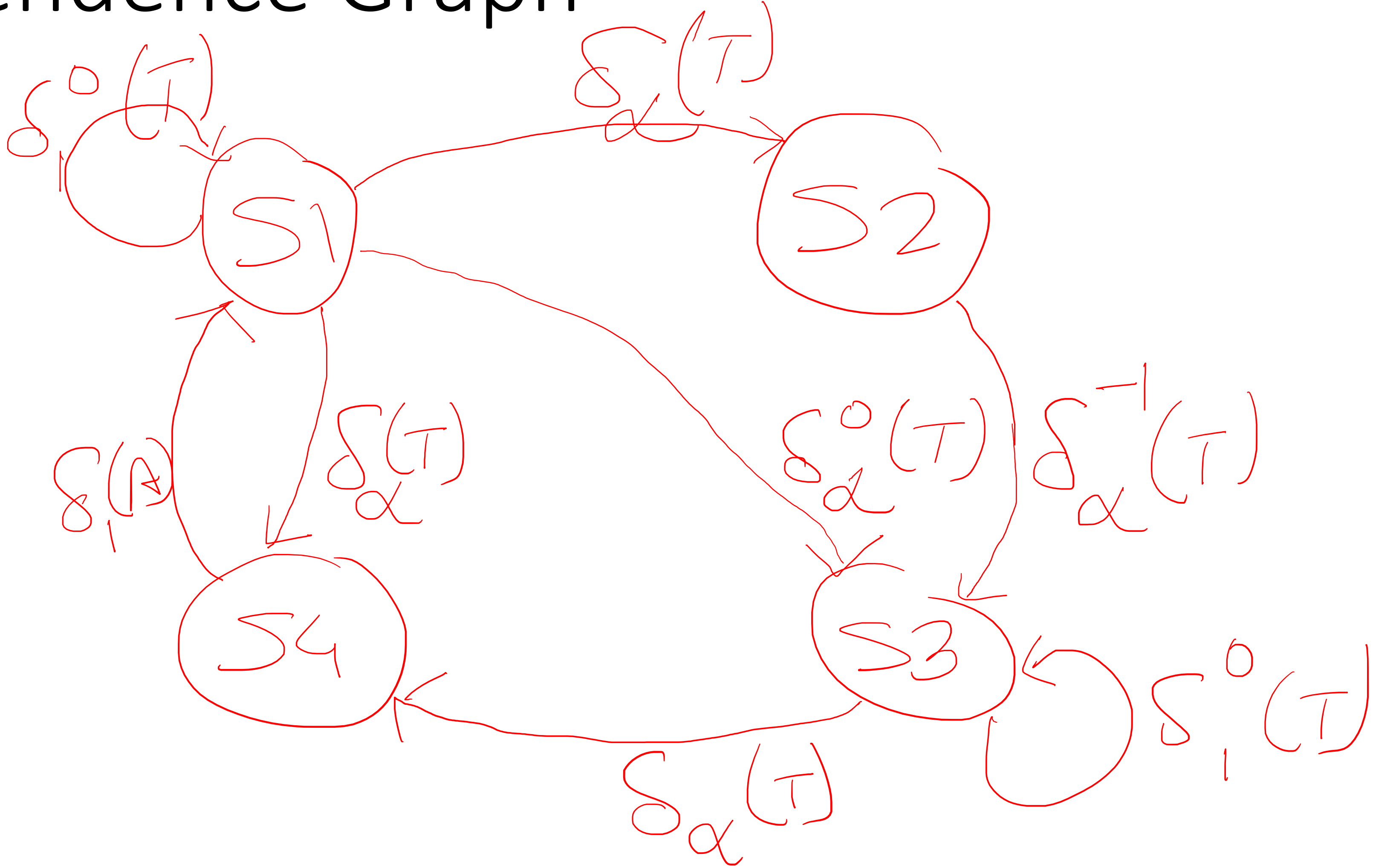
- Eliminates dependences due to reuse of memory locations
- Helps with uncovering parallelism

Cons

- Increases memory overhead
- Complicates addressing

Draw the Dependence Graph

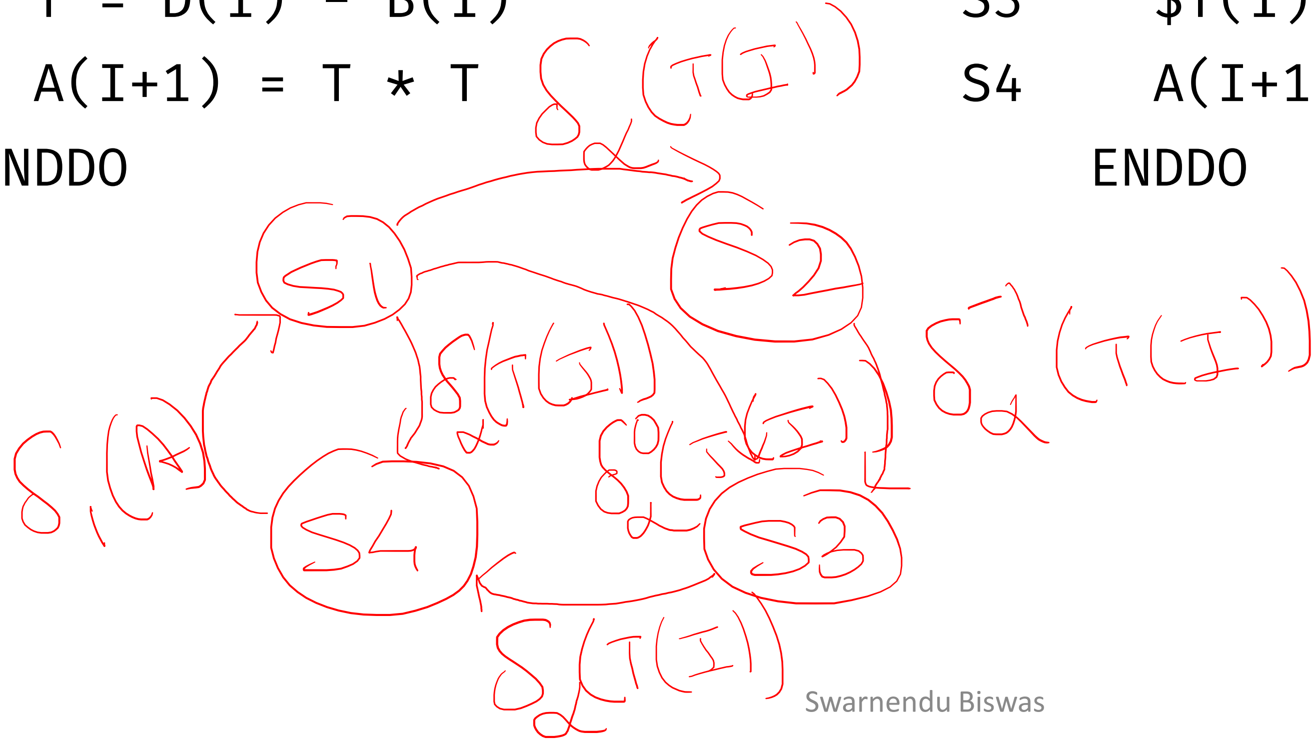
```
      DO I = 1, 100
S1      T = A(I) + B(I)
S2      C(I) = T + T
S3      T = D(I) - B(I)
S4      A(I+1) = T * T
      ENDDO
```



Scalar Expansion Does Not Help!

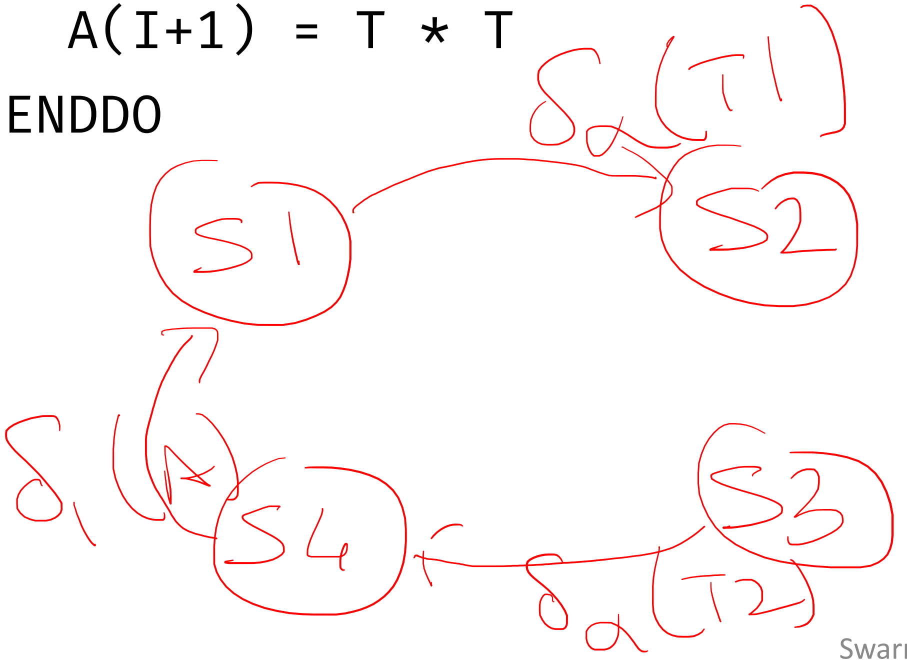
```
    DO I = 1, 100
S1    T = A(I) + B(I)
S2    C(I) = T + T
S3    T = D(I) - B(I)
S4    A(I+1) = T * T
    ENDDO
```

```
    DO I = 1, 100
S1    $T(I) = A(I) + B(I)
S2    C(I) = $T(I) + $T(I)
S3    $T(I) = D(I) - B(I)
S4    A(I+1) = $T(I) * $T(I)
    ENDDO
```



Scalar Renaming

```
     DO I = 1, 100
S1     T = A(I) + B(I)
S2     C(I) = T + T
S3     T = D(I) - B(I)
S4     A(I+1) = T * T
     ENDDO
```




```
     DO I = 1, 100
S1     T1 = A(I) + B(I)
S2     C(I) = T1 + T1
S3     T2 = D(I) - B(I)
S4     A(I+1) = T2 * T2
     ENDDO
     T = T2
```

Loop Peeling

- Splits any problematic **first or last few** iterations from the loop body
- Change from a loop-carried dependence to loop-independent dependence

S

```
DO I = 1, N
  A(I) = A(I) + A(1)
ENDDO
```

A(1) = A(1) + A(1)
A(2) = A(2) + A(1)
A(3) = A(3) + A(1)

```
A(1) = A(1) + A(1)
DO I = 2, N
  A(I) = A(I) + A(1)
ENDDO
```

A(1) = A(1) + A(1)
A[2:N] = A[2:N] + A(1)

Loop Peeling

- Splits any problematic **first or last few** iterations from the loop body
- Change from a loop-carried dependence to loop-independent dependence

```
int p = 10;
for (int i = 0; i < 10; ++i) {
  y[i] = x[i] + x[p];
  p = i;
}
```

y[0] = x[0] + x[10]
y[1] = x[1] + x[0]

```
y[0] = x[0] + x[10];
for (int i = 1; i < 10; ++i) {
  y[i] = x[i] + x[i-1];
}
```

} p x

https://en.wikipedia.org/wiki/Loop_splitting

Loop Splitting

assume N is divisible by 2

```
DO I = 1, N
  A(I) = A(N/2) + B(I)
ENDDO
```

```
M = N/2
DO I = 1, M-1
  A(I) = A(N/2) + B(I)
ENDDO
A(M) = A(N/2) + B(I)
DO I = M+1, N
  A(I) = A(N/2) + B(I)
ENDDO
```

Understanding Loop Peeling and Splitting

Pros

- Transformed loop carries no dependence, can be parallelized

Cons

Draw the Dependence Graph

```
      DO I = 1, N
        DO J = 1, N
S         A(I,J) = A(I-1,J) + A(I,J-1)
        ENDDO
      ENDDO
```

$I_0 = I_0 + \Delta I$

$J_0 = J_0 - 1 + \Delta J$

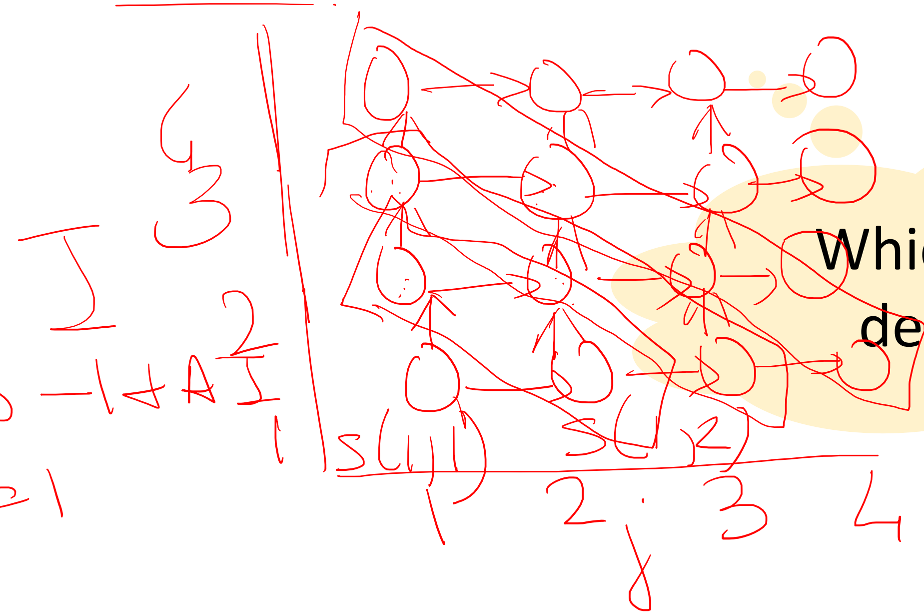
$\Delta J = 1$ (0, 1)

Delta flow

$J_0 = J_0 - 1 + \Delta J$

$\Delta J = 1$

(1, 0)

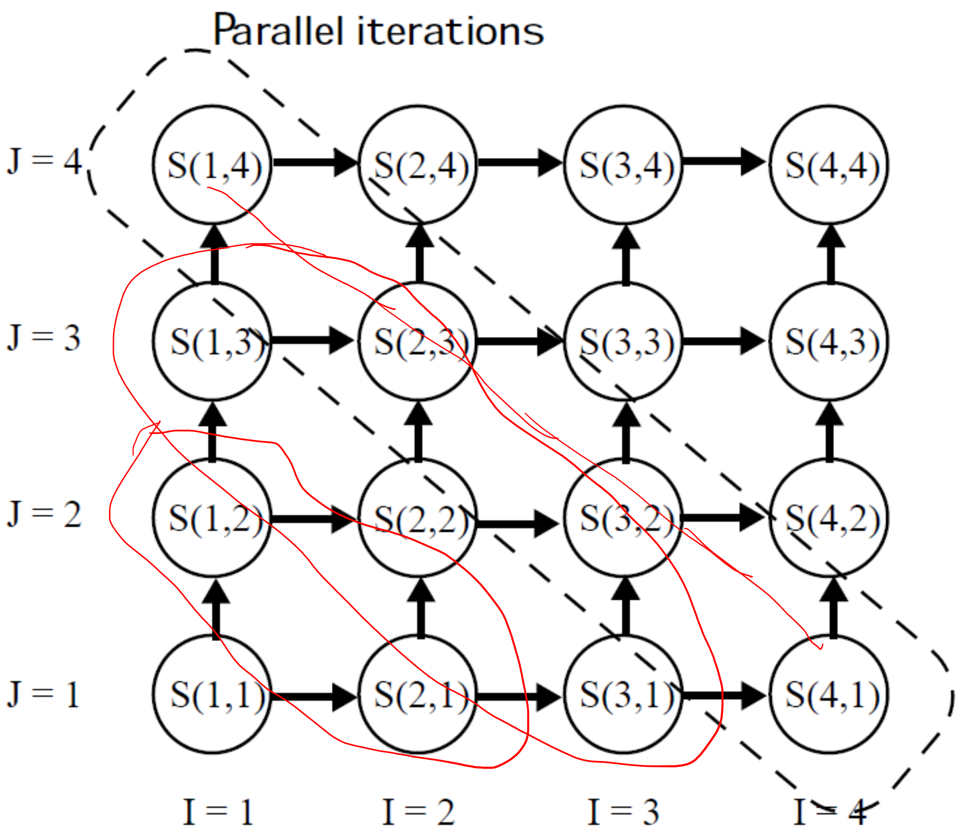


Which loops carry dependences?

$$\begin{bmatrix} + & 0 \\ 0 & + \end{bmatrix}$$

Loop Skewing

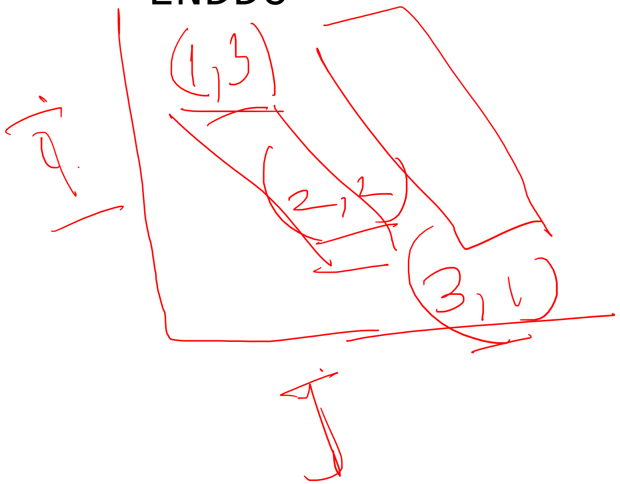
```
    DO I = 1, N
      DO J = 1, N
S       A(I,J) = A(I-1,J) + A(I,J-1)
      ENDDO
    ENDDO
```



Loop Skewing

```
   DO I = 1, N
     DO J = 1, N
S      A(I,J) = A(I-1,J) + A(I,J-1)
     ENDDO
   ENDDO
```

```
   DO I = 1, N
     DO j = I+1, I+N
S      A(I,j-I) = A(I-1,j-I) + A(I,j-I-1)
     ENDDO
   ENDDO
```



$j = J + I$

iteration space

Loop Skewing

$j_0 - I_0 = j_0 - I_0 - 1 + \Delta j_0$

$\Delta j_0 = 1 \quad (0,1)$

0,

```
    DO I = 1, N
      DO j = I+1, I+N
S       A(I,j-I) = A(I-1,j-I) + A(I,j-I-1)
      ENDDO
    ENDDO
```

flow

$(+,+)$

$(0,+)$

$I_0 = I_0 - 1 + \Delta I$

$\Delta I = 1$

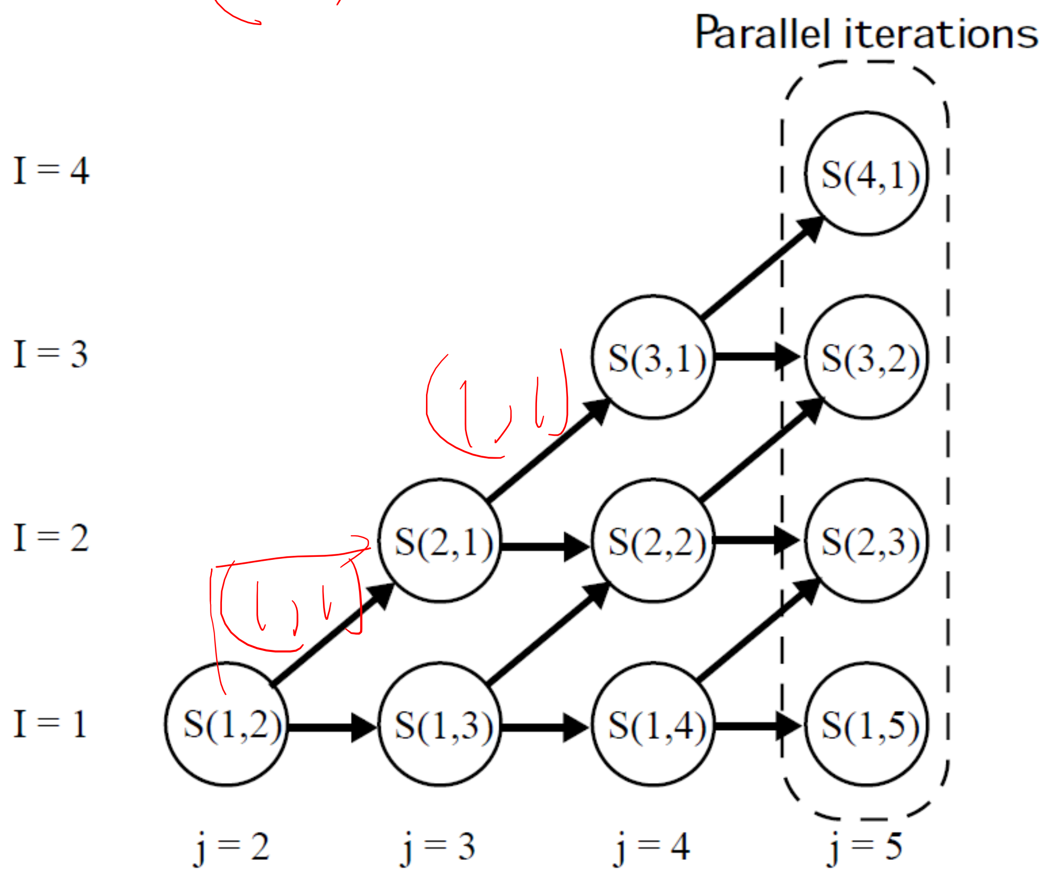
$\cancel{j_0} - \cancel{I_0} = \cancel{j_0} - (\cancel{I_0} + \Delta I_0) + \Delta j_0$

$\Delta j_0 = \Delta I_0 = 1$

$(1,0)$

$(0,1)$

$(1,1)$



Perform Loop Interchange

```
    DO I = 1, N
      DO j = I+1, I+N
S       A(I,j-I) = A(I-1,j-I) + A(I,j-I-1)
      ENDDO
    ENDDO
```

???

Which loop carries the dependence?

(0, 1) (0, +)

(1, 1) (+, +)

interchange

j I

Perform Loop Interchange

```
    DO I = 1, N
      DO j = I+1, I+N
S       A(I,j-I) = A(I-1,j-I) + A(I,j-I-1)
      ENDDO
    ENDDO
```

```
    DO j = 2, N+N
      DO I = max(1,j-N), min(N,j-1)
S       A(I,j-I) = A(I-1,j-I) + A(I,j-I-1)
      ENDDO
    ENDDO
```

parallelize

Understanding Loop Skewing

Pros

- Reshapes the iteration space to find possible parallelism
- Allows for loop interchange in future

Cons

- Resulting iteration space can be trapezoidal
- Irregular loops are not very amenable for vectorization
- Need to be careful about load imbalance

Loop Unrolling (Loop Unwinding)

```
for (i = 0; i < n; i++) {
  a[i] = a[i-1] + a[i] + a[i+1];
}
```

```
for (i = 0; i < n; i+ = 4) {
  a[i] = a[i-1] + a[i] + a[i+1];
  a[i+1] = a[i] + a[i+1] + a[i+2];
  a[i+2] = a[i+1] + a[i+2] + a[i+3];
  a[i+3] = a[i+2] + a[i+3] + a[i+4];
}
int f = n % 4;
for (i = n - f ; i < n; i ++) {
  a[i] = a[i-1] + a[i] + a[i+1];
}
```

Loop Unrolling (Loop Unwinding)

- Reduce number of iterations of loops
  - Add statement(s) to do work of missing iterations
- JIT compilers try to perform unrolling at run-time

```
for (i = 0; i < n; i++) {
  for (j = 0; j < 2*m; j++) {
    loop-body(i, j);
  }
}
```

```
for (i = 0; i < n; i++) {
  for (j = 0; j < 2*m; j+=2) {
    loop-body(i, j);
    loop-body(i, j+1);
  }
}
```

2-way unrolled

Inner Loop Unrolling

```
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
    y[i] = y[i] + a[i][j]*x[j];
  }
}
```

```
for (i=0; i<n; i++) {
  for (j=0; j<n; j+=4) {
    y[i] = y[i] + a[i][j]*x[j];
    y[i] = y[i] + a[i][j+1]*x[j+1];
    y[i] = y[i] + a[i][j+2]*x[j+2];
    y[i] = y[i] + a[i][j+3]*x[j+3];
  }
}
```

Inner Loop Unrolling

```
for (i=0; i<n; i++) {
  for (j=0; j<n; j+=4) {
    y[i] = y[i] + a[i][j]*x[j];
    y[i] = y[i] + a[i][j+1]*x[j+1];
    y[i] = y[i] + a[i][j+2]*x[j+2];
    y[i] = y[i] + a[i][j+3]*x[j+3];
  }
}
```

```
for (i=0; i<n; i++) {
  for (j=0; j<n; j+=4) {
    y[i] = y[i] + a[i][j]*x[j]
                + a[i][j+1]*x[j+1]
                + a[i][j+2]*x[j+2]
                + a[i][j+3]*x[j+3];
  }
}
```

Outer Loop Unrolling

```
for (i=0; i<2*n; i++)
  for(j=0; j<m; j++)
    loop-body(i,j);
```

```
for (i=0; i<2*n; i+=2) {
  for(j=0; j<m; j++) {
    loop-body(i,j)
  }
  for(j=0; j<m; j++) {
    loop-body(i+1,j)
  }
}
```

Does this work?

Outer Loop Unrolling

```
for (i=0; i<2*n; i++)
  for(j=0; j<m; j++)
    loop-body(i,j);
```

2-way outer unroll does not increase operation-level parallelism

```
for (i=0; i<2*n; i+=2) {
  for(j=0; j<m; j++) {
    loop-body(i,j)
  }
  for(j=0; j<m; j++) {
    loop-body(i+1,j)
  }
}
```

Outer Loop Unrolling + Inner Loop Jamming

```
for (i=0; i<2*n; i++)
  for(j=0; j<m; j++)
    loop-body(i,j);
```

```
for (i=0; i<2*n; i+=2) {
  for(j=0; j<m; j++) {
    loop-body(i,j)
    loop-body(i+1,j)
  }
}
```

Legality of Unroll and Jam

```
DO I = 1, N*2
  DO J = 1, M
    A(I+1,J-1) = A(I,J)+B(I,J)
  ENDDO
ENDDO
```

S(1,2) → S(2,1)

flow W→R

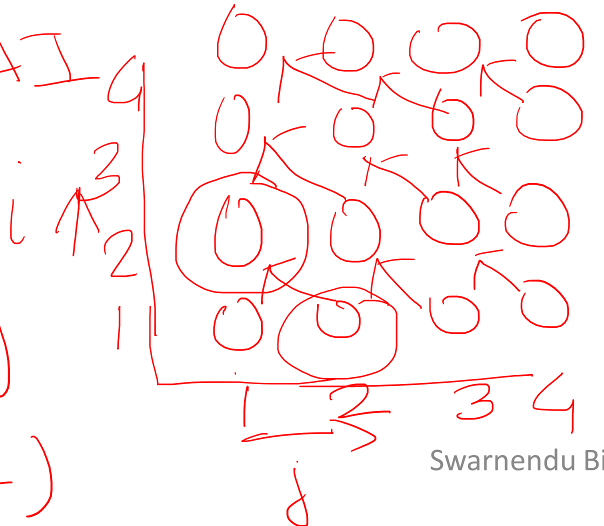
$I_0 + 1 = I_0 + \Delta I$

$\Delta I = 1$

$\Delta J = -1$

(1, -1)

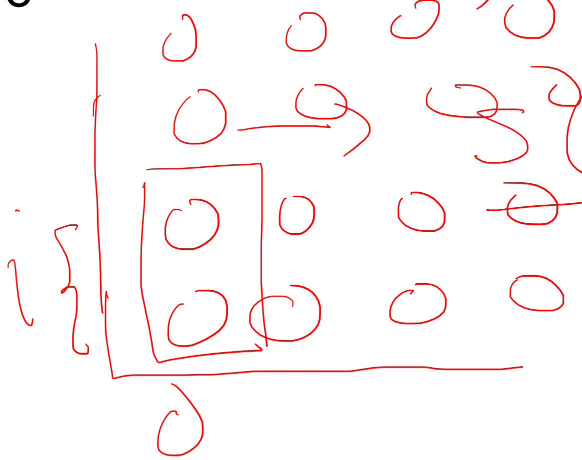
(+, -)



```
DO I = 1, N*2, 2
  DO J = 1, M
    A(I+1,J-1) = A(I,J)+B(I,J)
    A(I+2,J-1) = A(I+1,J)+B(I+1,J)
  ENDDO
ENDDO
```

S(1,1) S(2,1)

S(1,2)



Validity Condition for Loop Unroll/Jam

- Sufficient condition can be obtained by observing that complete unroll/jam of a loop is equivalent to a loop permutation that moves that loop innermost, without changing order of other loops
- If such a loop permutation is valid, unroll/jam of the loop is valid
- Example: 4D loop ijkl; d1 = (1,-1,0,2), d2 = (1,1,-2,-1)
  - i: d1-> (-1,0,2,1) => invalid to unroll/jam
  - j: d1-> (1,0,2,-1); d2 -> (1,-2,-1,1) => valid to unroll/jam
  - k: d1 -> (1,-1,2,0); d2 -> (1,1,-1,-2) => valid to unroll/jam
  - l: d1 and d2 are unchanged; innermost loop always unrollable

Understanding Loop Unrolling

Pros

- Small loop bodies are problematic, reduces control overhead of loops
- Increases operation-level parallelism in loop body
- Allows other optimizations like reuse of temporaries across iterations

Cons

- Increases the executable size
- Increases register usage
- May prevent function inlining

Loop Tiling

- Improve data reuse by chunking the data in to smaller blocks (tiles)
  - The block is supposed to fit in the cache
- Tries to exploit spatial and temporal locality of data

```
for (i = 0; i < N; i++) {
  …
}
```

```
for (j = 0; j < N; j +=B) {
  for (i = j; i < min(N, j+B); j++) {
    …
  }
}
```

MVM with 2x2 Blocking

```
int i, j, a[100][100], b[100], c[100];
int n = 100;
for (i = 0; i < n; i++) {
  c[i] = 0;
  for (j = 0; j < n; j++) {
    c[i] = c[i] + a[i][j] * b[j];
  }
}
```

```
int i, j, x, y, a[100][100], b[100], c[100];
int n = 100;
for (i = 0; i < n; i += 2) {
  c[i] = 0;
  c[i + 1] = 0;
  for (j = 0; j < n; j += 2) {
    for (x = i; x < min(i + 2, n); x++) {
      for (y = j; y < min(j + 2, n); y++) {
        c[x] = c[x] + a[x][y] * b[y];
      }
    }
  }
}
```

https://en.wikipedia.org/wiki/Loop_nest_optimization

Loop Tiling

- Determining the tile size
  - Difficult theoretical problem, usually heuristics are applied
  - Tile size depends on many factors

Validity Condition for Loop Tiling

- A contiguous band of loops can be tiled if they are fully permutable
- A band of loops is fully permutable of all permutations of the loops in that band are legal
- Example: d = (1,2,-3)
  - Tiling all three loops ijk is not valid, since the permutation kij is invalid
  - 2D tiling of band ij is valid
  - 2D tiling of band jk is valid

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    for (k = 0; k < n; k++)
      loop_body(i,j,k)

for (it = 0; it < n; it+=T)
  for (jt = 0; tj < n; j+=T)
    for (i = it; i < it+T; i++)
      for (j = jt; j < jt+T; j++)
        for (k = 0; k < n; k++)
          loop_body(i,j,k)
```

Creating Coarse-Grained Parallelism

Find Work For Threads

- Setup
  - Symmetric multiprocessors with shared-memory
  - Threads are running on each core, and coordinating execution with occasional synchronization

  - A basic synchronization element is a barrier
  - A barrier in a program forces all processes to reach a certain point before execution continues.

- Challenge: Balance the granularity of parallelism with communication overheads

Challenges in Coarse-Grained Parallelism

Minimize communication and synchronization overhead while evenly load balancing across the processors

- Running everything on one processor achieves minimal communication and synchronization overhead

- Very fine-grained parallelism achieves good load balance, but benefits possibly are outweighed by frequent communication and synchronization

Challenges in Coarse-Grained Parallelism

Minimize communication an… while evenly load balancin…

- Runnin… process… communicatio… synchronization overhead

…ism …balance, but …y are outweighed …frequent communication and synchronization

One expectation from an optimizing compiler is to find the sweet spot

Few Ideas to Try

- Single loop
  - Carries a dependence ➔ Try transformations to eliminate the loop carried dependence
  - For example, loop distribution and scalar expansion
  - Decide on the granularity of the new parallel loop
- Perfect loop nests
  - Try loop interchange to see if the dependence level can be changed

Privatization

- Privatization is similar in flavor to scalar expansion
- Temporaries can be given separate namespaces for each iteration

```
    DO I = 1,N
S1     T = A(I)
S2     A(I) = B(I)
S3     B(I) = T
    ENDDO
```

```
    PARALLEL DO I = 1,N
      PRIVATE t
S1    t = A(I)
S2    A(I) = B(I)
S3    B(I) = t
    ENDDO
```

Privatization

- A scalar variable x in a loop L is said to be **privatizable** if every path from the loop entry to a use of x inside the loop passes through a definition of x
- No use of the variable is upward exposed, i.e., the use never reads a value that was assigned outside the loop
- No use of the variable is from an assignment in an earlier iteration

Privatization

- If all dependences carried by a loop involve a privatizable variable, then loop can be parallelized by making the variables private

- Preferred compared to scalar expansion

Why?

Privatization

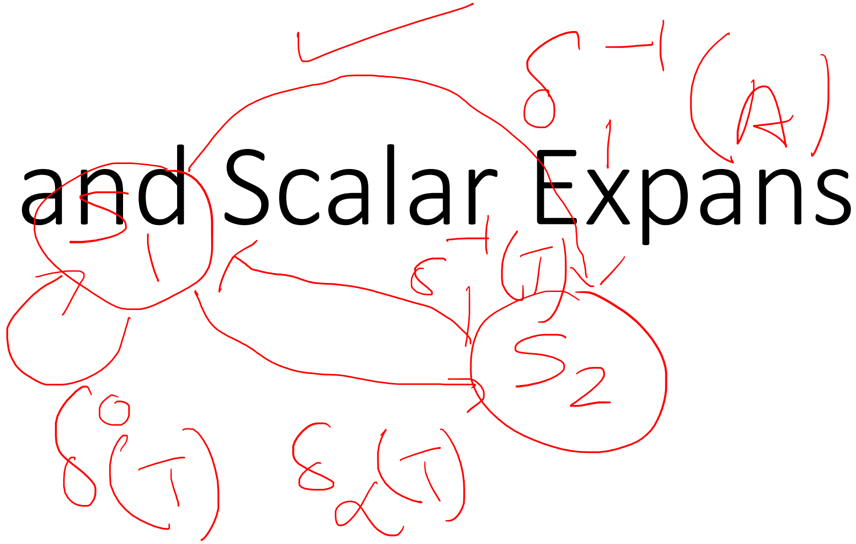
- If all dependences carried by a loop involve a privatizable variable, then loop can be parallelized by making the variables private

- Preferred compared to scalar expansion
  - Less memory requirement
  - Scalar expansion may suffer from false sharing

- However, there can be situations where scalar expansion works but privatization does not

Privatization and Scalar Expansion

```
DO I = 1, N
  T = A(I) + B(I)
  A(I-1) = T
ENDDO
```




```
DO I = 1, N
  PRIVATE T
  T = A(I) + B(I)
  A(I-1) = T
ENDDO
```

I	W	R
1	A(0)	A(1)
2	A(1)	A(2)
3	A(2)	A(3)
4	A(3)	A(4)

Privatization and Scalar Expansion

```
DO I = 1, N
  T = A(I) + B(I)
  A(I-1) = T
ENDDO
```

```
DO I = 1, N
  PRIVATE T
  T = A(I) + B(I)
  A(I-1) = T
ENDDO
```

```
PARALLEL DO I = 1, N
  T$(I) = A(I) + B(I)
  A(I-1) = T$(I)
ENDDO
```

Privatization and Scalar Expansion

```
DO I = 1, N
  T = A(I) + B(I)
  A(I-1) = T
ENDDO
```

```
DO I = 1, N
  PRIVATE T
  T = A(I) + B(I)
  A(I-1) = T
ENDDO
```

1

```
PARALLEL DO I = 1, N
  T$(I) = A(I) + B(I)
ENDDO
```

m

barrier

(n-1)th

2

```
PARALLEL DO I = 1, N
  A(I-1) = T$(I)
ENDDO
```

m

Loop Distribution (Loop Fission)

```
      DO I = 1, 100
        DO J = 1, 100
S1        A(I,J) = B(I,J) + C(I,J)
S2        D(I,J) = A(I,J-1) * 2.0
        ENDDO
      ENDDO
```

S_1

– flow W → R

$J_0 = J_0 + \Delta J \Rightarrow 0$

$J_0 = J_0 - 1 + \Delta J \Rightarrow 1$

(0, 1)

- How to eliminate loop-carried dependences?

Loop Distribution (Loop Fission)

```
    DO I = 1, 100
      DO J = 1, 100
S1        A(I,J) = B(I,J) + C(I,J)
S2        D(I,J) = A(I,J-1) * 2.0
      ENDDO
    ENDDO
```

(0,1)

(0,0)

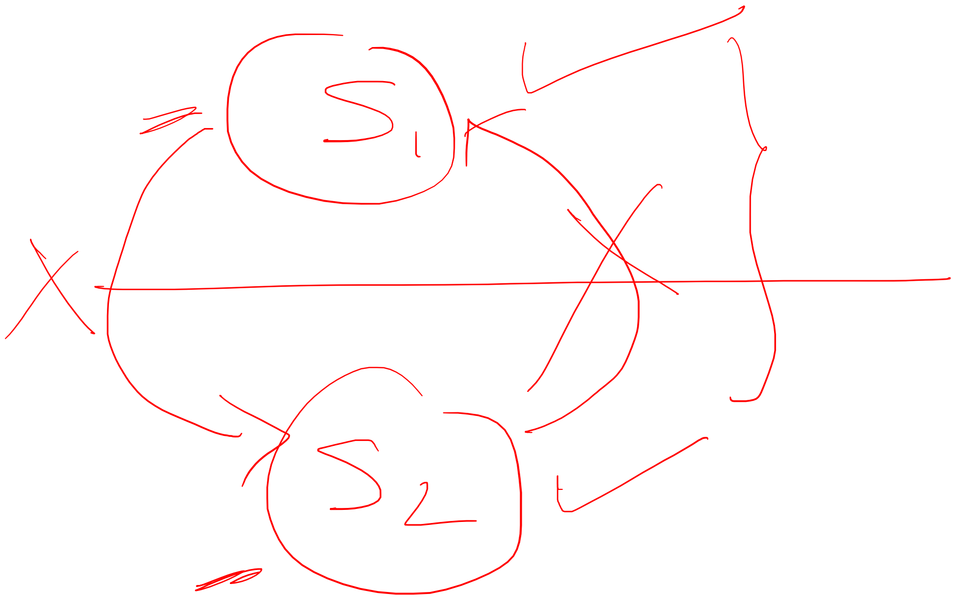
```
    DO I = 1, 100
      DO J = 1, 100
S1        A(I,J) = B(I,J) + C(I,J)
      ENDDO

      DO J = 1, 100
S2        D(I,J) = A(I,J-1) * 2.0
      ENDDO
    ENDDO
```

consume

- Goal is to eliminate loop-carried dependences

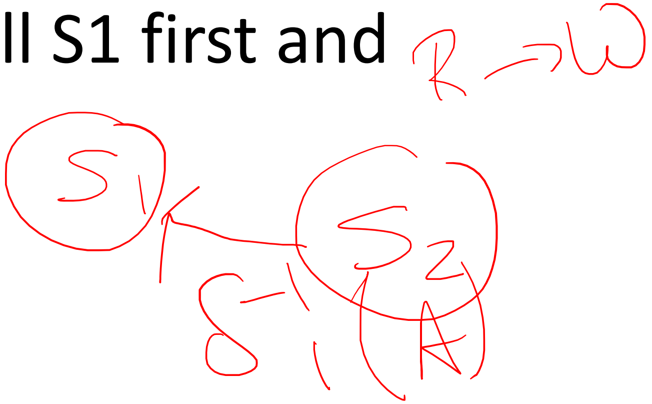
Validity Condition for Loop Distribution

- Sufficient (but not necessary) condition: A loop with two statements can be distributed if there are no dependences from any instance of the **later** statement to any instance of the **earlier** one
  - Generalizes to more statements



Validity Condition for Loop Distribution

- Example: Loop distribution is not valid (executing all S1 first and then all S2)




```
      For I = 1, N
S1      A(I) = B(I) + C(I)
S2      E(I) = A(I+1) * D(I)
      EndFor
```

- Example: Loop distribution is valid

```
      For I = 1, N
S1      A(I) = B(I) + C(I)
S2      E(I) = A(I-1) * D(I)
      EndFor
```

Understanding Loop Distribution

Pros

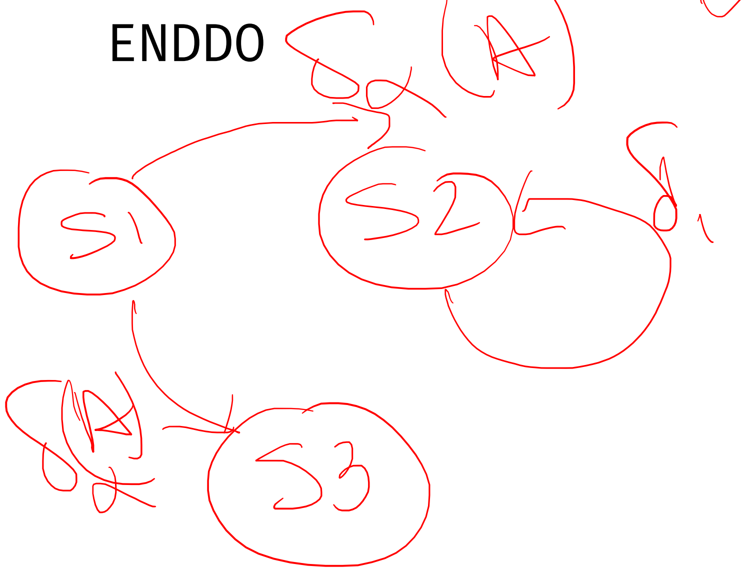
- Execute source of a dependence before the sink
- Reduces the memory footprint of the original loop
  - For both data and code

Cons

- Can increase the synchronization required between dependence points

How to deal with the loop?

```
      DO I = 1, N
S1       A(I) = B(I) + 1
S2       C(I) = A(I) + C(I-1)
S3       D(I) = A(I) + X
      ENDDO
```




```
L1  DO I = 1, N
      A(I) = B(I) + 1
    ENDDO
L2  DO I = 1, N
      C(I) = A(I) + C(I-1)
    ENDDO
L3  DO I = 1, N
      D(I) = A(I) + X
    ENDDO
```

Loop Fusion (Loop Jamming)

```
L1  DO I = 1, N
      A(I) = B(I) + 1
    ENDDO
L2  DO I = 1, N
      C(I) = A(I) + C(I-1)
    ENDDO
L3  DO I = 1, N
      D(I) = A(I) + X
    ENDDO
```

```
L1  PARALLEL DO I = 1, N
      A(I) = B(I) + 1
L3    D(I) = A(I) + X
    ENDDO
L2  DO I = 1, N
      C(I) = A(I) + C(I-1)
    ENDDO
```

Loop Fusion Allowed?

```
     DO I = 1, N
S1     A(I) = B(I) + C
     ENDDO
     DO I = 1, N
S2     D(I) = A(I+1) + E
     ENDDO
```

I	R	W
1	A(2)	D(1)
2	A(3)	A(2)

```
     DO I = 1, N
S1     A(I) = B(I) + C
S2     D(I) = A(I+1) + E
     ENDDO
```

No

Loop Fusion Allowed?

```
    DO I = 1, N
S1    A(I) = B(I) + C
    ENDDO
    DO I = 1, N
S2    D(I) = A(I-1) + E
    ENDDO
```

I	W	B
1	A(1)	A(0)
2	A(2)	A(1)
3	A(3)	A(1)

```
    DO I = 1, N
S1    A(I) = B(I) + C
S2    D(I) = A(I-1) + E
    ENDDO
```

$I+1 \Rightarrow I-1$

Yes

Validity Condition for Loop Fusion

- Loop-independent dependence between statements in two different loops (i.e., from S1 to S2)
- Dependence is *fusion-preventing* if fusing the two loops causes the dependence to be carried by the combined loop in the reverse direction (from S2 to S1)

A(J+1)

Understanding Loop Fusion

Pros

- Reduce overhead of loops
- May improve temporal locality

Cons

- May decrease data locality in the fused loop

Loop Interchange

```
DO I = 1, N
  DO J = 1, M
    A(I+1,J) = A(I,J) + B(I,J)
  ENDDO
ENDDO
```

Which loop carries a dependence?

Loop Interchange

```
DO I = 1, N
  DO J = 1, M
    A(I+1,J) = A(I,J) + B(I,J)
  ENDDO
ENDDO
```

Loop I carries a dependence

Parallelizing J is good for vectorization, but not from coarse-grained parallelism

Loop Interchange

```
DO I = 1, N
  DO J = 1, M
    A(I+1,J) = A(I,J) + B(I,J)
  ENDDO
ENDDO
```

```
DO J = 1, M
  DO I = 1, N
    A(I+1,J) = A(I,J) + B(I,J)
  ENDDO
ENDDO
```

Dependence-free loops should move to the outermost level

```
PARALLEL DO J = 1, M
  DO I = 1, N
    A(I+1,J) = A(I,J) + B(I,J)
  ENDDO
END PARALLEL DO
```

Loop Interchange

Vectorization

- Move dependence-free loops to innermost level

Coarse-grained Parallelism

- Move dependence-free loops to outermost level

Loop Interchange

```
DO I = 1, N
  DO J = 1, M
    A(I+1,J+1) = A(I,J) + B(I,J)
  ENDDO
ENDDO
```

How about this?

Condition for Loop Interchange

- In a perfect loop nest, a loop can be parallelized at the outermost level if and only if the column of the direction matrix for that nest contains only “0” entries

Code Generation Strategy

1) Continue till there are no more columns to move
    1) Choose a loop from the direction matrix that has all "0" entries in the column
    2) Move it to the outermost position
    3) Eliminate the column from the direction matrix
2) Pick loop with most "+" entries, move to the next outermost position
    1) Generate a sequential loop
    2) Eliminate the column
    3) Eliminate any rows that represent dependences carried by this loop
3) Repeat from Step 1

Loop Interchange

```
DO I = 1, N
  DO J = 1, M
    DO K = 1, L
      A(I+1,J,K) = A(I,J,K) + X1
      B(I,J,K+1) = B(I,J,K) + X2
      C(I+1,J+1,K+1) = C(I,J,K) + X3
    ENDDO
  ENDDO
ENDDO
```

$$\begin{bmatrix} + & 0 & 0 \\ 0 & 0 & + \\ + & + & + \end{bmatrix}$$

Can we permute the loops?

Loop Interchange

+	0	0
0	0	+
+	+	+

```
DO I = 1, N
  DO J = 1, M
    DO K = 1, L
      A(I+1,J,K) = A(I,J,K) + X1
      B(I,J,K+1) = B(I,J,K) + X2
      C(I+1,J+1,K+1) = C(I,J,K) + X3
    ENDDO
  ENDDO
ENDDO
```

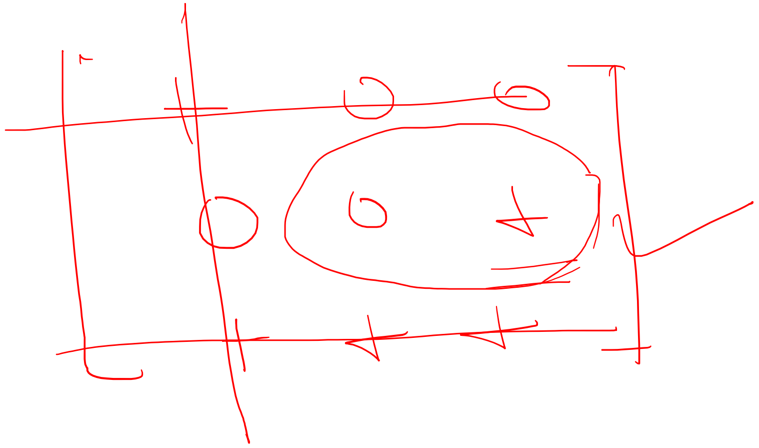
Since there are no columns with all “0” entries, none of the loops can be parallelized at the outermost level

Generated Code

I → seq

```
DO I = 1, N
  PARALLEL DO J = 1, M
    DO K = 1, L
      A(I+1,J,K) = A(I,J,K) + X1
      B(I,J,K+1) = B(I,J,K) + X2
      C(I+1,J+1,K+1) = C(I,J,K) + X3
    ENDDO
  END PARALLEL DO
ENDDO
```

How did we pick loop J?



How can we parallelize this loop?

```
DO I = 2, N+1
  DO J = 2, M+1
    DO K = 1, L
      A(I,J,K) = A(I,J-1,K+1) + A(I-1,J,K+1)
    ENDDO
  ENDDO
ENDDO
```

$$\begin{array}{cc} & \begin{array}{ccc} & J & \end{array} \\ \begin{array}{c} \\ I \end{array} & \begin{bmatrix} 0 & + & - \\ + & 0 & - \end{bmatrix} \end{array}$$

Construct the direction matrix

How can we parallelize this loop?

```
DO I = 2, N+1
   DO J = 2, M+1
      DO K = 1, L
         A(I,J,K) = A(I,J-1,K+1) + A(I-1,J,K+1)
      ENDDO
   ENDDO
ENDDO
```

0	**+**	~~**-**~~ +
+	**0**	~~**-**~~ +

K I J

parallelize

No single loop carries all the dependences

Loop Reversal

```
DO I = 2, N+1
  DO J = 2, M+1
    DO K = 1, L
      A(I,J,K) = A(I,J-1,K+1) + A(I-1,J,K+1)
    ENDDO
  ENDDO
ENDDO
```

```
DO I = 2, N+1
  DO J = 2, M+1
    DO K = L, 1, -1
      A(I,J,K) = A(I,J-1,K+1) + A(I-1,J,K+1)
    ENDDO
  ENDDO
ENDDO
```

$$\begin{bmatrix} 0 & + & + \\ + & 0 & + \end{bmatrix}$$

Loop Reversal

- When the iteration space of a loop is reversed, the direction of dependences within that reversed iteration space are also reversed. Thus, a “+" dependence becomes a “-" dependence, and vice versa

```
DO I = 2, N+1
  DO J = 2, M+1
    DO K = L, 1, -1
      A(I,J,K) = A(I,J-1,K+1) + A(I-1,J,K+1)
    ENDDO
  ENDDO
ENDDO
```

0	+	+
+	0	+

Perform Loop Interchange

```
DO K = L, 1, -1
  DO I = 2, N+1
    DO J = 2, M+1
      A(I,J,K) = A(I,J-1,K+1) + A(I-1,J,K+1)
    ENDDO
  ENDDO
ENDDO
```

Parallelize loops I and J

+	0	+
+	+	0

Understanding Loop Reversal

Pros

- Increases options for performing other optimizations

Cons

Which Transformations are Most Important?

- Flow dependences by nature are difficult to remove
  - Try to reorder statements as in loop peeling, loop distribution
- Techniques like scalar expansion, privatization can be very useful
  - Loops often use scalars for temporary values

Challenges for Real-World Compilers

- Conditional execution
- Symbolic loop bounds
- Indirect memory accesses
- …

References

- R. Allen and K. Kennedy – Optimizing Compilers for Multicore Architectures.
- S. Midkiff – Automatic Parallelization: An Overview of Fundamental Compiler Techniques.
- P. Sadayappan and A. Sukumaran Rajam – CS 5441: Parallel Computing, Ohio State University.