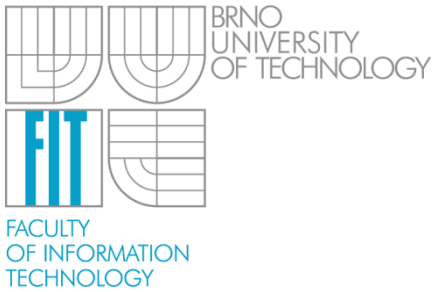
Parallelism in Modern Compilers

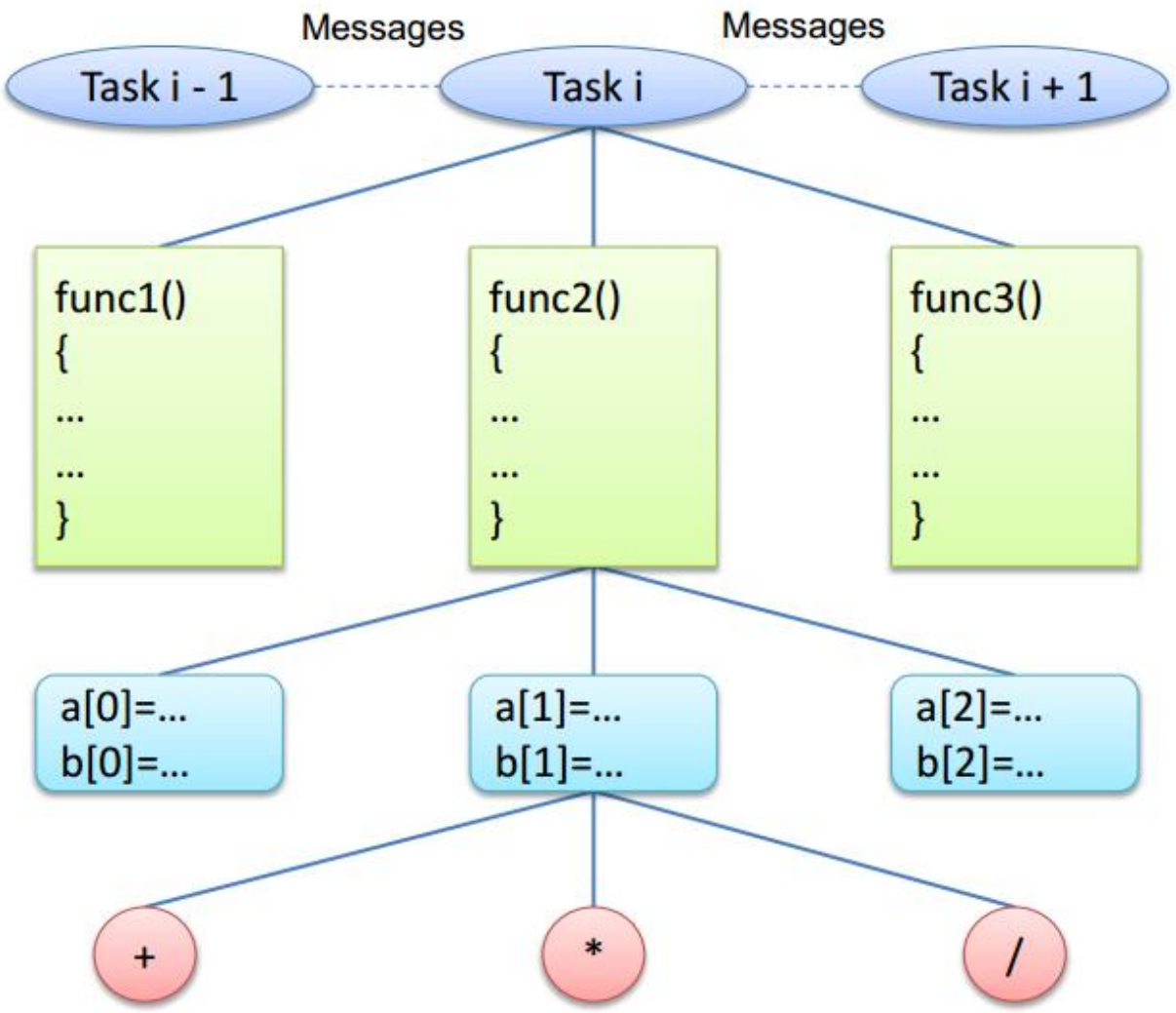
Vojtech Nikl

Brno University of Technology, Faculty of Information Technology
Bozetechova 2, 612 66 Brno

Modern Theoretical Computer Science
2014/2015



What is parallelism?



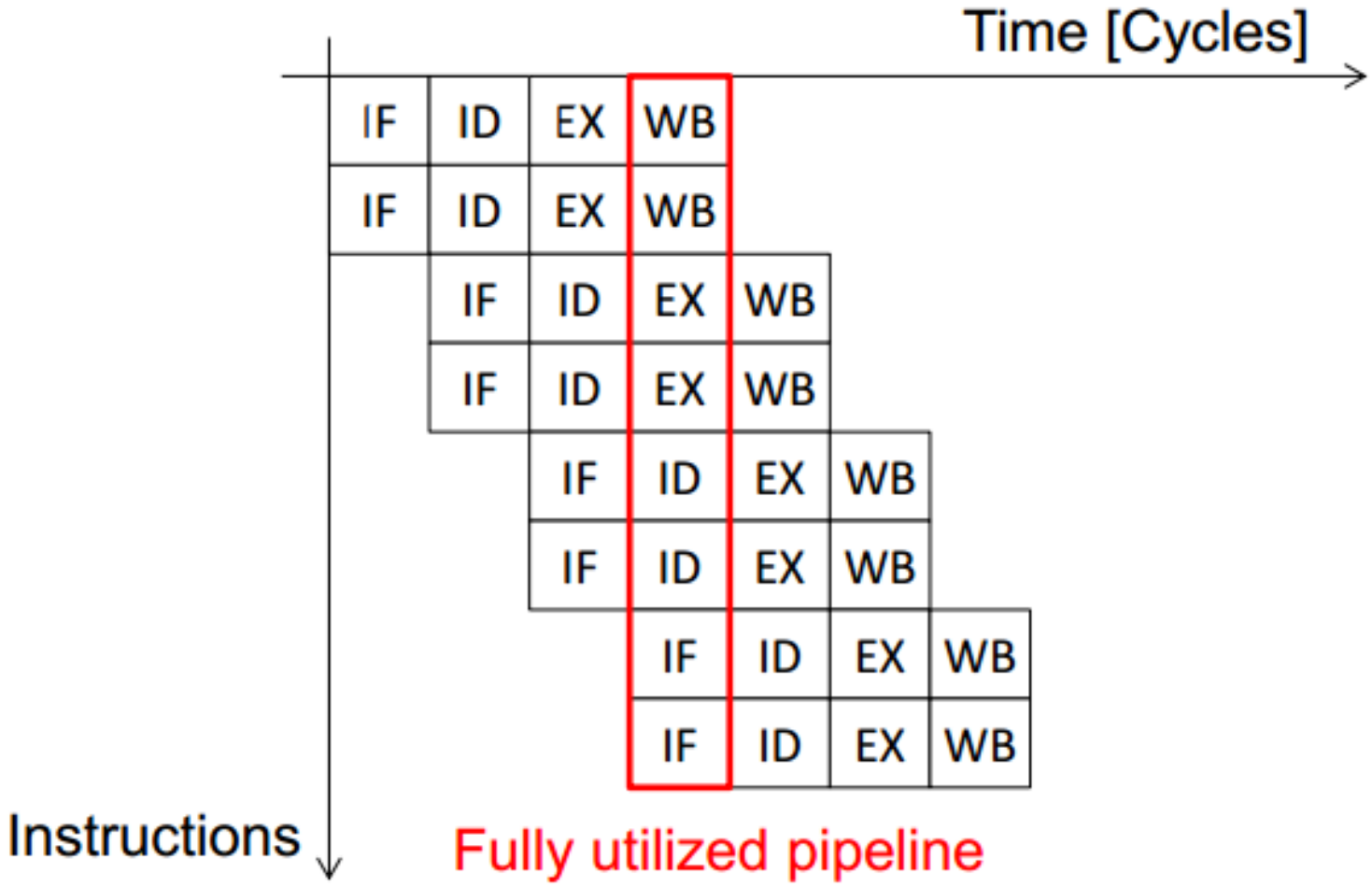
Task level parallelism (TLP) large grain

Control level parallelism (CLP) medium grain

Data level parallelism (DLP) fine grain

Instruction level parallelism (ILP) very fine grain

- Implemented in hardware



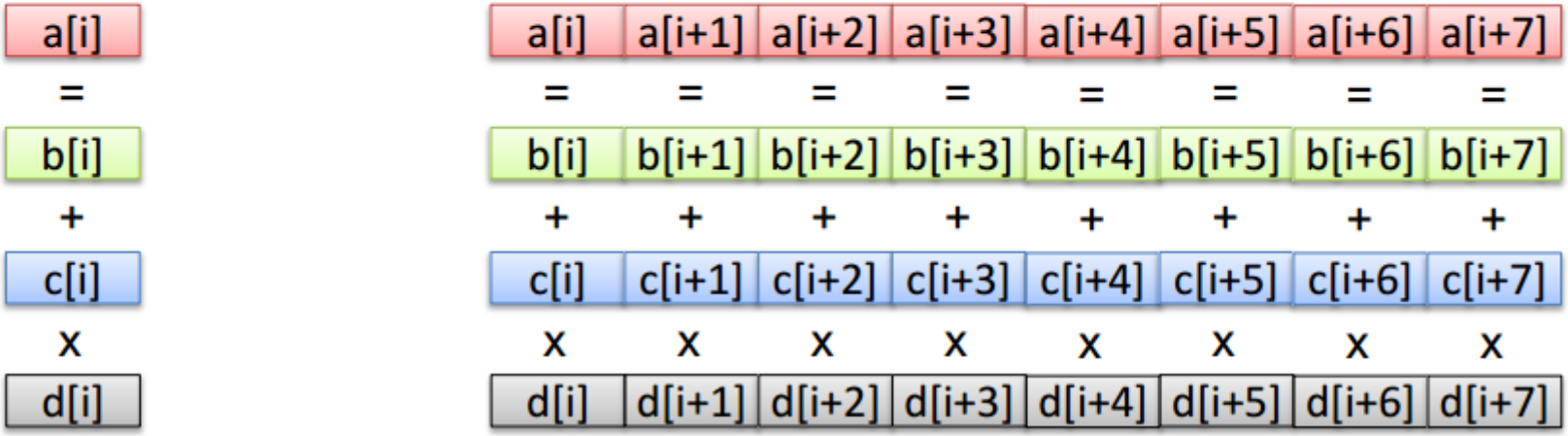
- Vectorization

Scalar version works on one element at a time

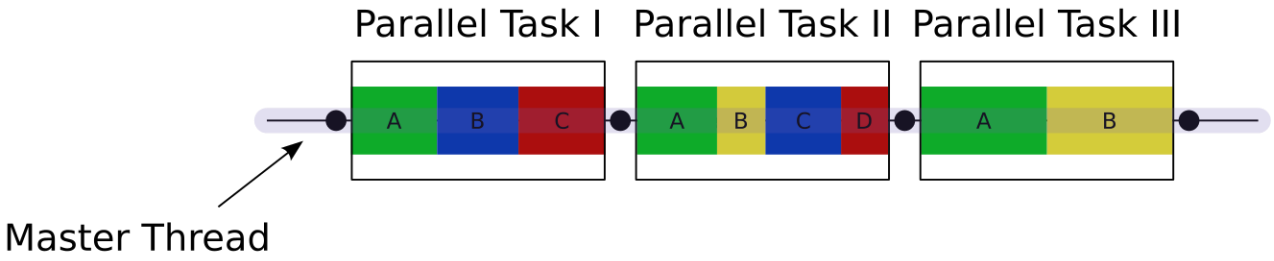
a[i] = b[i] + c[i] x d[i];

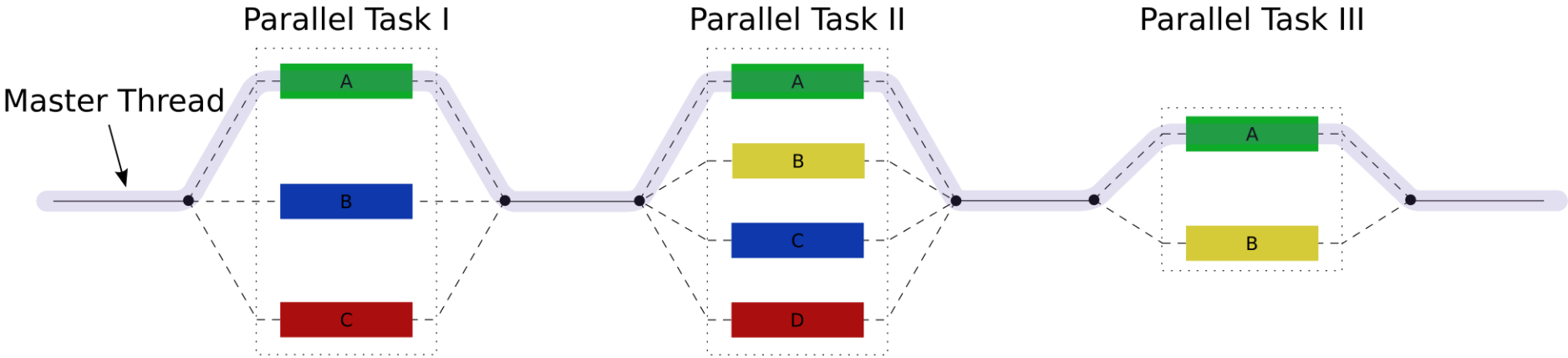
Vector version carries out the same instructions on many elements at a time

a[i:8] = b[i:8] + c[i:8] * d[i:8];



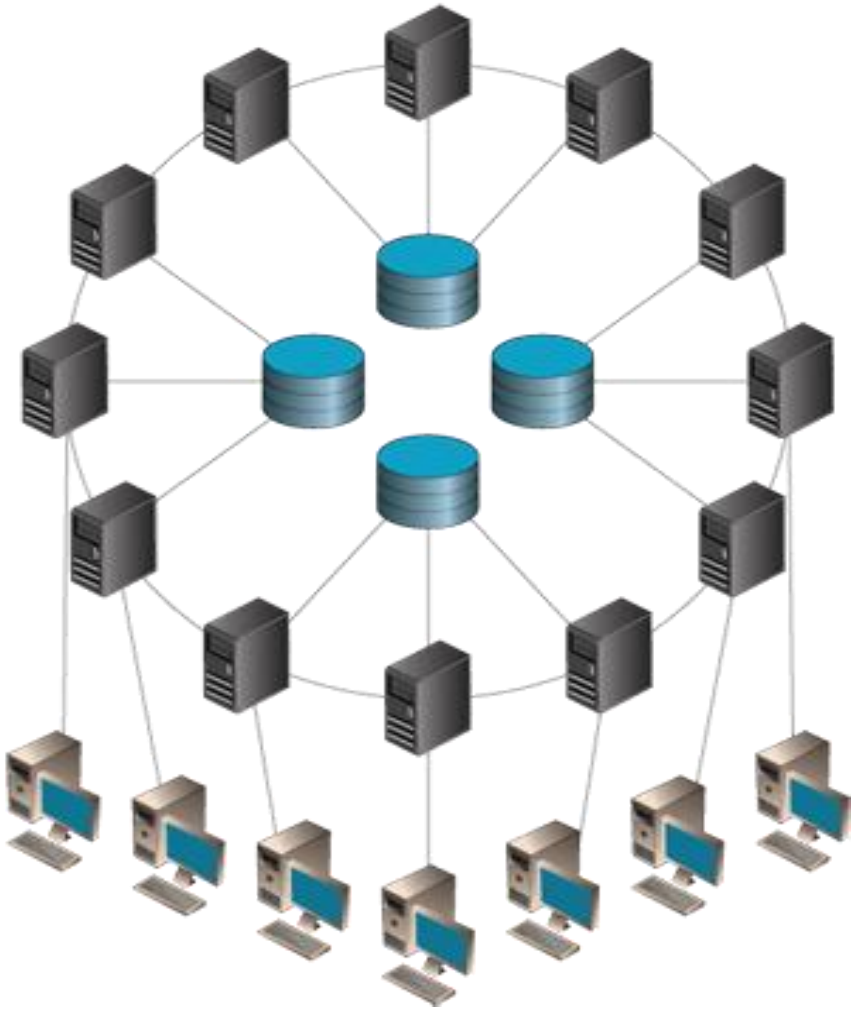
- Threading (shared memory)





Task Level Parallelism

- Message passing interface (distributed memory)



Enhancing Parallelism

Motivation

- Many compiler optimizations are based on the idea of either:
  - Reordering statements (or smaller units)
  - Executing them in parallel
- Goal: do this without changing the semantics of the program.
- Problem: **Data dependencies**

Data Dependence

- Given two program statements **a** and **b**, **b** depends on **a** if:
  - **b** follows **a**
  - they share the same memory location
  - one of them writes to it
- Written: **b** δ **a**
- Example:
  - **a**: x = y + 1;
  - **b**: z = x * 3;

Because **b** depends on **a**, the two statements cannot be reordered, nor can they be run in parallel

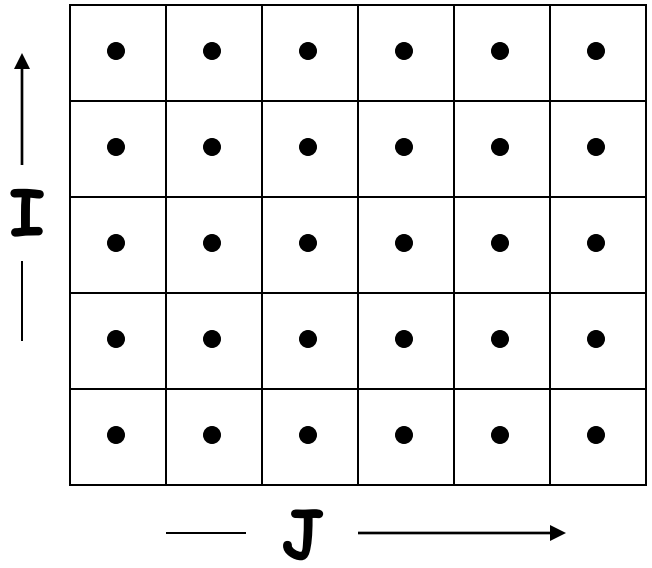
Dependence Classification

- A dependence, **a** δ **b**, is one of the following:
  - true of flow dependence:
    - **a** writes a location that **b** later reads
    - (read-after write or RAW)
  - anti-dependence
    - **a** reads a location that **b** later writes
    - (write-after-read or WAR)
  - output dependence
    - **a** writes a location that **b** later writes
    - (write-after-write or WAW)

Iteration Space

- Example

```
do I = 1, 5
  do J = I, 6
    . . .
  enddo
enddo
```



- Written out in lexicographic order the iteration space is:

  (1,1), (1,2),…, (1,6),(2,2),(2,3),…

- Equivalent to sequential execution order

Vector Order

- Let R^n be the set of all real n-vectors, ($n > 1$)
- A lexicographic order $<_u$ on these vectors is a relation:

$$i <_u j \text{ on vectors } i = \{ i_1 \dots i_n \}$$
$$j = \{ j_1 \dots j_n \}$$
$$\text{if}$$
$$i_1 = j_1, j_1 = j_2 \dots \text{ and } i_u < j_u$$

- The leading element of a vector is the first non-zero element
- A negative vector has: leading element < 0
- A positive vector has: leading element > 0

Distance/Direction Vectors

- Given 2 n-vectors i,j

  $i = (i_1, \ldots i_n)$

  $j = (j_1, \ldots j_n)$

- Their distance vector = $(j_1 - i_1, j_2 - i_2, \ldots)$
  - Represents the number of iterations between accesses to the same location
- Their direction vector

  $= (sig\ (j_1 - i_1), sig(j_2 - i_2), \ldots)$
  - Represents the direction in iteration space

Distance/Direction Vectors

- It is often convenient to deal with *in-completely specified* direction vectors

Example 1:

{(0, 0, 0, 1), (0, -1, 0, 1), (0, 0, 1, 1), (0, -1, 1, 1)}

==> {(0, <= 0, >= 0, 1)}

Example 2:

{(0, -1, 0, -1), (0, 0, 0, -1), (0, 1, 0, -1)}

==> {(0, *, 0, -1)}

Distance/Direction Vectors

- Let *i, j* denote two vectors in R^n and *s* their direction vector. Then i < j if *s* has one of the following n forms:

  (1, *, *, …, *)
  (0, 1, *, …, *)
  (0, 0, 1, *, …, *)

  (0, 0, …, 0, 1).

- Notation

  (0, 1, -1) ⇔ (=, >, <)

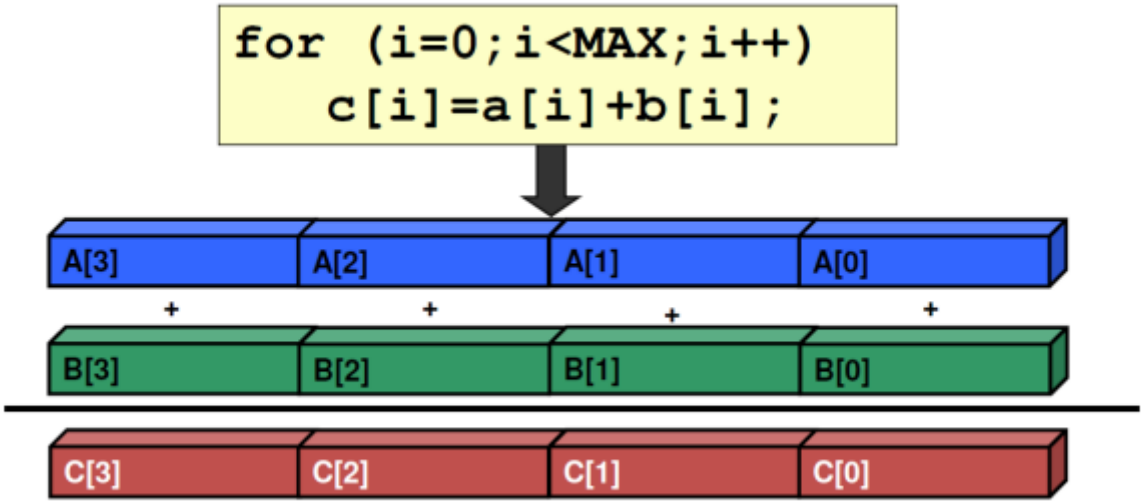
Loop Parallelization

- It is valid to convert a sequential loop into a parallel loop if the loop carries no dependences.

- **Proof**
  - Iteration reordering is fine if the loop carries no dependencies
  - Dependence violation may occur due to interleaving of statements from different loop iterations
    - dependences at a level inside the parallelized loop cannot cause a violation since they are executed sequentially
    - dependences at a level outside the parallelized loop are preserved

Loop Parallelization

- Modern compilers analyse loops in serial code identification for vectorization



- Intel compiler: `-xSSE/-xAVX`
  `-ftree-vectorize`
  `-vec-report`

Loop Parallelization

- Formally, a statement in a loop can be directly vectorized if the statement is not included in any cycle of dependence.
- This simple algorithm misses opportunities for vect.

```
for (i=0; i<N; i++)
  for (j=0; j<M; j++)
    A[i+1, j] = A[i,j] + c;
```

  - Direction D = (<,=)

- However, we can vectorize the loop as follows

```
for (i=0; i<N; i++)
  A[i+1, 0:M-1] = A[i,0:M-1] + c;
```

Loop Interchange

- Loop interchange switches the nesting order of two loops in a perfect loop nest
- Consider a perfect nest of loops $L = (L_1, L_2, \ldots, L_m)$

L_1: **do** $I_1 = p_1, q_1, \theta_1$

L_2: **do** $I_2 = p_2, q_2, \theta_2$

$\vdots$ $\vdots$

L_m: **do** $I_m = p_m, q_m, \theta_m$

$H(I_1, I_2, \ldots, I_m)$

enddo

$\vdots$

enddo

enddo

Loop Interchange

- The direction vector of a dependence after applying a permutation on the loop order of a perfect loop nest is obtained by applying the same permutation to the original direction vector of the dependence

```
for (i=1; i<=N; i++)                    for (j=1; j<=M; j++)
 for (j=1; j<=M; j++)          =>        for (i=1; i<=N; i++)
  A[i,j+1] = A[i, j] + c;                 A[i,j+1] = A[i, j] + c;
```

D = (=, <) D = (<,=)

- Intuitively, the direction vector of an interchange preventing dependence is D = (<,>)

Scalar Expansion

- Scalar expansion is the replacement of a scalar by a temporary array that provides a different storage location for the scalar in each iteration

- Scalar expansion is not "free"
  - extra memory
  - additional memory accesses
  - more complex addressing

- Main challenge: determining profitable expansions by looking at the dependence graph

Scalar Expansion

1)
```
for (j=1; j<=M; j++) {
  for (i=1; i<=N; i++) {
    t = 0;
    for (k=1; k<=L; k++)
      t += A[i,k] * B[k,j];
    C[i,j] = t;
}}
```

2)
```
for (j=1; j<=M; j++) {
  for (i=1; i<=N; i++) {
    _t[i] = 0;
    for (k=1; k<=L; k++)
      _t[i] += A[i,k] * B[k,j];
    C[i,j] = _t[i];
}}
```

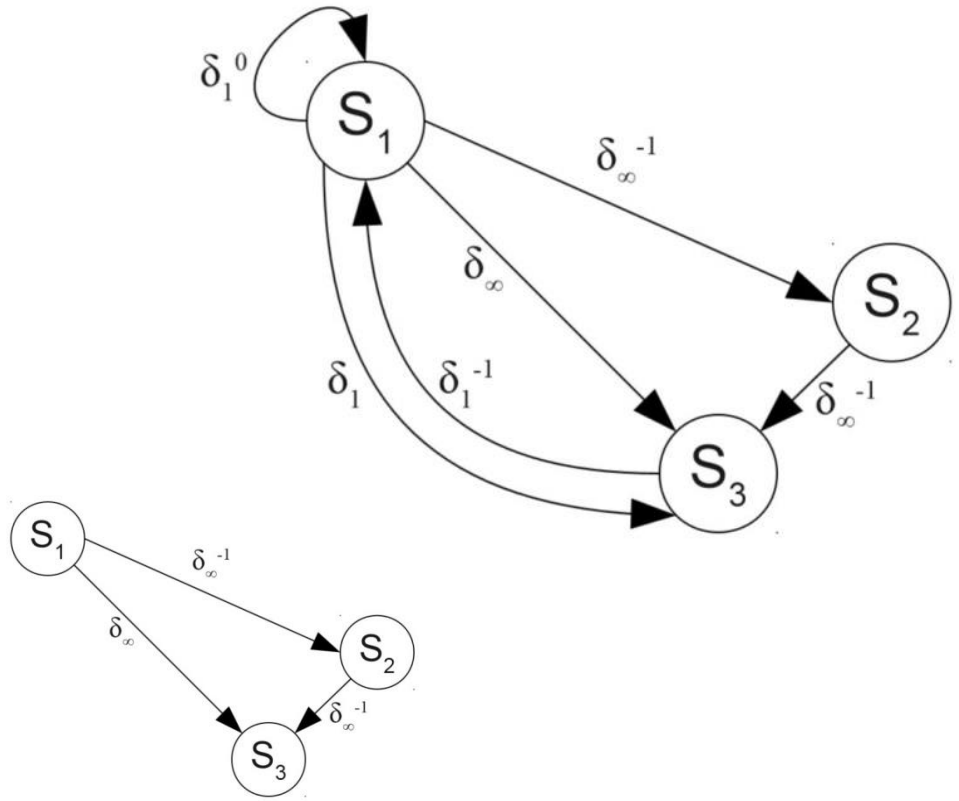
3)
```
for (j=1; j<=M; j++) {
  for (i=1; i<=N; i++)
    _t[i] = 0;
  for (i=1; i<=N; i++)
    for (k=1; k<=L; k++)
      _t[i] += A[i,k] * B[k,j];
  for (i=1; i<=N; i++)
    C[i,j] = _t[i];
}
```

Scalar Expansion

- In a DG, edges that arise from the reuse of memory locations can be eliminated by scalar expansion. Such edges are called detectable.
- The other type of edges arise from the reuse of values; those must always be preserved.

```
   for (i=1; i<=N ;i++)
S1:   tmp  = A[i];
S2:   A[i] = B[i];
S3:   B[i] = tmp;
```

```
   for (i=1; i<=N ;i++)
S1: tmp[i] = A[i];
S2: A[i]   = B[i];
S3: B[i]   = tmp[i];
```



Conclusion

- Parallelism enhances the overall performance
- Compilers focus mainly on *data level parallelism*
- 3 main techniques explained
  - Loop Parallelization (Vectorization)
  - Loop Interchange
  - Scalar Expansion

Thank you for your attention!