Compilation Considerations for Parallel and Vector Architectures

A Few Sources

- Michael Wolfe, High-Performance Compilers for Parallel Computing, Addison-Wesley, 1996.
- Hans Zima, with Barbara Chapman, Superompilers for Parallel and Vector Computers, ACM Press, 1990.
- Thomas Braunl, Parallel Programming, an Introduction, Prentice-Hall, 1993.

Bernstein's Conditions (1966)

- For a statement S:
  - IN(S) = set of variables, registers, or locations used by S
  - OUT(S) = set written to by S
- S_1; S_2 (sequence) is equivalent to $S_1 \| S_2$ (parallel) **provided** that
  - $OUT(S_1) \cap OUT(S_2) = \varnothing$
  - $OUT(S_1) \cap IN(S_2) = \varnothing$
  - $OUT(S_2) \cap IN(S_1) = \varnothing$

Data Dependence

- Expresses constraints on parallel execution, as derived from sequential execution semantics
- Types of Dependence (Kuck, Wolfe, et al.):
  - Flow dependence
  - Anti dependence
  - Output dependence

Flow Dependence

- A variable assigned to in one statement is used in a later one:

A = 5

Flow Dependence

B = A*A

Anti Dependence

- A variable use in one statement is assigned to in a later one:

B = A*A

Anti Dependence

A = 5

Output Dependence

- A variable assigned to in one statement is later re-assigned to:

A = B*B

Output Dependence

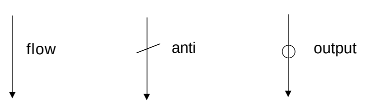
A = 5

Removable Dependences

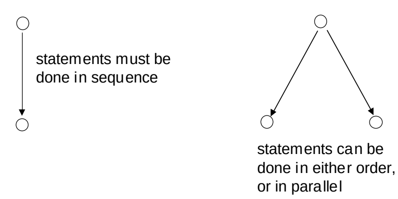
- Anti Dependence and Output Dependence are **removable**
- They are artifacts of using variables as if memory location, rather than purely for their values.
- Flow Dependence is **not removable**; it expresses essential precedence.
- Clarification of whether location- or value-based dependency is being considered will be left to context.

Notation

- $S_1 \delta^f S_2$ means S_2 is flow dependent on S_1
- $S_1 \delta^a S_2$ means S_2 is anti dependent on S_1
- $S_1 \delta^o S_2$ means S_2 is output dependent on S_1



Dependence Relations determine a Partial Order on Statement Execution



Location- vs. Value-Based

- Consider

Value-based | Location-based

A = 5

B = A+7

A = 99

C = A*2

By using a different variable, the dependency is removed

- Consider

Value-based | Location-based

A = 5

B = A+7

AA = 99

C = AA*2

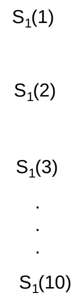
Loops add to the Challenge

- Consider

  for K= 1 to 10

  $S_1(K)$ A[K] = B[K]

Dependency graph



- Conclude: All instances $S_1(K)$ can be done concurrently (since no arrows).

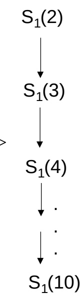
Loops add to the Challenge

- Consider

  for K= 2 to 10

  $S_1(K)$ A[K] = A[K-1]

Dependency graph



- Conclude: All instances $S_1(K)$ must be done in sequence.

Larger offsets allow more concurrency

- Consider

  for K= 3 to 10

  $S_1(K)$ A[K] = A[K-2]

- $S_1(3)$ || $S_1(4)$ is possible
- $S_1(K)$ || $S_1(K+1)$ is possible, K = 3, 5, …
- Similarly, A[K] = A[K-d] will allow degree d concurrency.

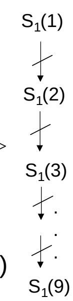
"Forward" Offsets

- Consider

  for K= 1 to 9

  $S_1(K)$ A[K] = A[K+1]

Dependency graph



- Conclude: All instances $S_1(K)$ must be done in sequence (if location-based assumption used)

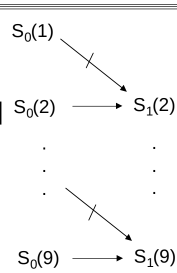
We can *Transform* the Previous Example

for K= 1 to 9

$S_0(K)$ B[K] = A[K+1]

for K= 1 to 9

$S_1(K)$ A[K] = B[K]



Transformation reduces sequence constraints

for K= 1 to 9

$S_0(K)$ B[K] = A[K+1]

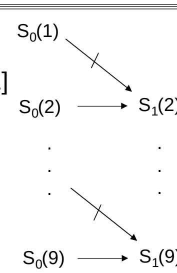
for K= 1 to 9

$S_1(K)$ A[K] = B[K]

F90 style:

B(1 : 9) = A(2 : 10)

A(2 : 10) = B(2 : 10)



The type of transformation just shown can be automated

This is done routinely in compilers for high-performance machines.

Parallel Execution of Loops Strategy

- Try to issue different instances of a loop body to separate processing elements.
- Generally loops occur nested; try to find appropriate nesting level where different instances can be issued.

Similar issue to Parallelization: Vectorization

- Vector machines:
  - Exploit fine-grain parallel operations (+, *, /) on *vector* elements
  - Typically done with *vector registers*
- Vectorizing concentrates on inner loop
- Parallelizing concentrates on outer loops (coarser grain)

Example of Loop Vectorization

- do K = 1 to N
  A[K] = B[K] + C[K]
  D[K] = A[K]*5

uses new A

Vectorizes to (using F90 notation):

-
  A(1:N) = B(1:N) + C(1:N)
  D(1:N) = A(1:N)*5

uses new A

Example of Loop Vectorization

- do K = 1 to N
  A[K] = B[K] + C[K]
  D[K] = A[K+1]*5

uses old A

Vectorizes to (using F90 notation):

-
  D(1:N) = A(2:N+1)*5
  A(1:N) = B(1:N) + C(1:N)

uses old A

Dependence Distance

- The dependence in the previous example can be summarized:

$$S_0(K)\ \delta^f_{(1)}\ S_1(K)$$

- This essentially says:
  - The i^{th} iteration of S_0 must be done before the $i+1^{th}$ iteration of S_1.

Dependence Distance

- In general, there may be a different set of dependence distances for each array:

do K = 2 to N

for A for B

$S_0(K)$ A[K] = B[K-1] $\delta^f_{(0)}$

$S_1(K)$ B[K] = A[K] $\delta^f_{(1)}$

- Each places a constraint on loop restructuring

Both Indices and Loop Direction must be taken into account in determining Dependence Distance

- do K = 2 to N-1

S_0 A[K] = B[K]

S_1 C[K] = A[K-1] $\delta^f_{(1)}$

is similar to

- do K = N-1 to 2 by -1

S_0 A[K] = B[K]

S_1 C[K] = A[K+1] $\delta^f_{(1)}$

in that C[K] gets the *new* value, not the old.

same dependence distance: The i^{th} iteration of S_0 must be done before the $i+1^{th}$ iteration of S_1.

Direction Vectors

- Less precise than Dependence Distances, but frequently used:
  - $\delta^f_{(<)}$ used in place of $\delta^f_{(n)}$ where n > 0
  - $\delta^f_{(=)}$ used in place of $\delta^f_{(0)}$
  - $\delta^f_{(>)}$ used in place of $\delta^f_{(n)}$ where n < 0
- Advantage of using > is that n might not be fixed, as in:

```
do K = 2 to 10
        A[2*K] = B[K]+1
        C[K] = A[K]
```

- Here the dependence distance *increases* with K.

Example

- do K = 2 to N

S_0 A[K] = B[K]

S_1 C[K] = A[K-1] $\delta^f_{(1)}$ or $\delta^f_{(<)}$

do K = 1 to N-1

S_0 A[K] = B[K]

S_1 C[K] = A[K+1] $\delta^a_{(1)}$ or $\delta^a_{(<)}$

doacross

- doacross K = M to N

is equivalent to HPF's INDEPENDENT annotation:

Each loop body is done independently of the other, possibly in parallel (There is still sequencing *within* the body.)

doacross Example

- Original loop

```
do K = 1 to N
        A[K] = C[K]
        B[K] = A[K]
```

$\delta^f_{(=)}$

- is optimized to

```
doacross K = 1 to N
        A[K] = C[K]
        B[K] = A[K]
```

Non-doacross Example

- Original loop
  ```
  do K = 2 to N
      A[K] = C[K]
      B[K] = A[K-1]
  ```
  $\delta^f_{(<)}$
- cannot be optimized using doacross alone.
- We could provide additional synchronization on the use of A[K-1] to do it, but it wouldn't be pure doacross.

Loops that "Carry" Dependence

- As we saw, loops having only $\delta^f_{(=)}$ are optimizable using doacross.
- A loop with $\delta^f_{(<)}$ or $\delta^f_{(>)}$ carries the/a dependence that constrains parallel execution.

Nested Loops

- For nested loops, a *vector* of dependences is used, e.g. $\delta^f_{(=, <)}$ or $\delta^a_{(=, =)}$ with one component per loop nest.
- When loops are nested, the *outermost* loop with a $\delta^f_{(<)}$ or $\delta^f_{(>)}$ *carries* the dependence.

Nested Loop Example

- ```
  do K = 2 to N
      do J = 2 to N
          A[K, J] = B[K, J]
          B[K, J] = A[K, J-1]
  ```
  for A: $\delta^f_{(=, <)}$ for B: $\delta^a_{(=, =)}$
- The inner loop carries the dependence for A; no loop carries the dependence for B.
- Therefore the outer can be parallelized using doacross.

Nested Loop Example

- ```
  do K = 2 to N
      do J = 2 to N
          A[K, J] = B[K, J]
          B[K, J] = A[K, J-1]
  ```
  for A: $\delta^f_{(=, <)}$ for B: $\delta^a_{(=, =)}$
- ```
  doacross K = 2 to N
      do J = 2 to N
          A[K, J] = B[K, J]
          B[K, J] = A[K, J-1]
  ```

Exercise

- How to parallelize:

  ```
  do K = 2 to N
      do J = 2 to N
          A[K, J] = C[K, J]
          B[K, J] = A[K-1, J]
  ```
  ```

Exercise

- How to parallelize:

```
do K = 2 to N
    do J = 2 to N
        A[K, J] = C[K, J]
        B[K, J] = A[K-1, J]
```

for A $\delta^f_{(<, =)}$

- The outer loop carries the dependency

Exercise

- do K = 2 to N
    do J = 2 to N
        A[K, J] = C[K, J]
        B[K, J] = A[K-1, J]

for A $\delta^f_{(<, =)}$

- Parallel:

```
do K = 2 to N
    doacross J = 2 to N
        A[K, J] = C[K, J]
        B[K, J] = A[K-1, J]
```

Loop Interchanging

- do K = 1 to N
    do J = 1 to N
S_1 A[K, J] = A[K, J-1] + A[K, J+1]

- S_1 $\delta^f_{(=, <)}$ S_1 implies inner loop cannot be vectorized.
- No dependencies of form $\delta^f_{(<, >)}$ implies loops can be interchanged

Loop Interchanging

- do K = 1 to N
    do J = 1 to N
        A[K, J] = A[K, J-1] + A[K, J+1]

⇩

- do J = 1 to N
    do K = 1 to N
        A[K, J] = A[K, J-1] + A[K, J+1]

- Now have $\delta^f_{(<, =)}$

Loop Interchanging

- do J = 1 to N
    do K = 1 to N
        A[K, J] = A[K, J-1] + A[K, J+1]

⇩

- Now have $\delta^f_{(<, =)}$
- Execute as:

```
do J = 1 to N
    A[1:N, J] = A[1:N, J-1] + A[1:N, J+1]
```