Loop dependence analysis

Troels Henriksen
Based on material by Cosmin Oancea

Motivation

Why do we have to look at sequential loops?

[1]A *loop nest* is a collection of multiple loops nested within each other.

Motivation

Why do we have to look at sequential loops?

- A lot of sequential code in C++/Java/Fortran.
- Need to parallelize the implementation of given algorithms.
- Need to optimize parallelism, e.g., cache optimisation requires subscript analysis.

[1] A *loop nest* is a collection of multiple loops nested within each other.

Motivation

Why do we have to look at sequential loops?

- A lot of sequential code in C++/Java/Fortran.
- Need to parallelize the implementation of given algorithms.
- Need to optimize parallelism, e.g., cache optimisation requires subscript analysis.

This will require us to:

1. Identify the loop nests[1] where most of the runtime is spent.
2. Parallelise these loops by analysis about which loops in the nest are parallel.
3. Decide on the manner in which loop nests can be re-written in order to optimise locality of reference, load balancing, etc.

[1] A *loop nest* is a collection of multiple loops nested within each other.

Loop nests

```
for (int i = 0; i < N; i++)
  for (int j = 0; j < M; j++)
    ... loop-nest body ...
```

Loop nests

```
for (int i = 0; i < N; i++)
  for (int j = 0; j < M; j++)
    ... loop-nest body ...
```

- Identify iterations of k-deep nest with k-element vector, e.g. $\vec{k} = (\mathtt{i} = 2, \mathtt{j} = 4)$.
- Ordered lexicographically:

$$(\mathtt{i} = 2, \mathtt{j} = 4) < (\mathtt{i} = 3, \mathtt{j} = 3)$$

  because third iteration of the outer loop is executed before the fourth iteration of the outer loop.
- This is **program order**.

Dependencies and transformations

Loop transformation: A change to a loop nest that ensures all dependencies are still respected.

Example of valid transformation

```
for (int j = 0; i < N; i++)              for (int i = 0; i < N; i++)
  for (int i = 0; i < N; i++)     ⇒        for (int j = 0; i < N; i++)
    A[i][j] = i*j;                             A[i][j] = i*j;
```

Dependencies and transformations

Loop transformation: A change to a loop nest that ensures all dependencies are still respected.

Example of valid transformation

```
for (int j = 0; i < N; i++)
  for (int i = 0; i < N; i++)
    A[i][j] = i*j;
```

⇒

```
for (int i = 0; i < N; i++)
  for (int j = 0; i < N; i++)
    A[i][j] = i*j;
```

Example of invalid transformation

```
int x = 0;
for (int j = 0; i < N; i++)
  for (int i = 0; i < N; i++)
    A[i][j] = x++;
```

⇒

```
int x = 0;
for (int i = 0; i < N; i++)
  for (int j = 0; i < N; i++)
    A[i][j] = x++;
```

Definition of load-store dependencies

True Dep. (RAW)	Anti Dep. (WAR)	Output Dep. (WAW)
`S1  X   = ...` `S2  ... = X`	`S1  ... = X` `S2  X   = ...`	`S1  X = ...` `S2  X = ...`

Loop Dependency:

A dependence from $S1$ to $S2$ exists in a loop nest *iff* there are iterations $\vec{k}$, $\vec{l}$ where $\vec{k} < \vec{l}$ or $\vec{k} = \vec{l}$, and there is a path from S1 to S2, and

1. $S1$ accesses memory location M on iteration $\vec{k}$, and
2. $S2$ accesses memory location M on iteration $\vec{l}$, and
3. one of these accesses is a write.

- **We say that** $S1$ **is the source and** $S2$ **is the sink of the dependence**.
- Dependence depicted with an arrow pointing from source to sink.

Three loop nests

```
for (int i = 0; i < N; i++)
  for (int j = 0; j < N; j++)
    S1: A[j][i] = A[j][i]...

for (int i = 1; i < N; i++)
  for (int j = 1; j < N; j++) {
    S1: A[j][i] = A[j-1][i-1]...
    S2: B[j][i] = B[j-1][i]...
  }

for (int i = 1; i < N; i++)
  for (int j = 0; j < N; j++)
    S1: A[i][j] = A[i-1][j+1]...
```

- **Which of these are parallel?**
- Can we transform them somehow to be more parallel?
- When is it safe to interchange the loops?

Loop A

```
for (int i = 0; i < N; i++)
  for (int j = 0; j < N; j++)
    S1: A[j][i] = A[j][i]...
```

Example: Iteration $(i = 3, j = 3)$

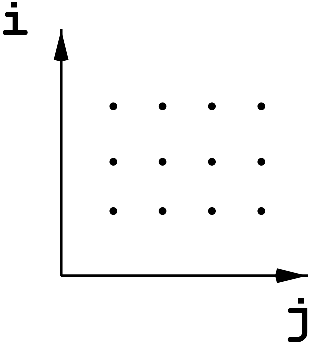
- Reads from $A[3][3]$.
- Writes to $A[3][3]$.
- So no dependency on other iterations—write that as a dot.

Loop A

```
for (int i = 0; i < N; i++)
  for (int j = 0; j < N; j++)
    S1: A[j][i] = A[j][i] ...
```

Example: Iteration $(i = 3, j = 3)$

- Reads from $A[3][3]$.
- Writes to $A[3][3]$.
- So no dependency on other iterations—write that as a dot.



Loop B

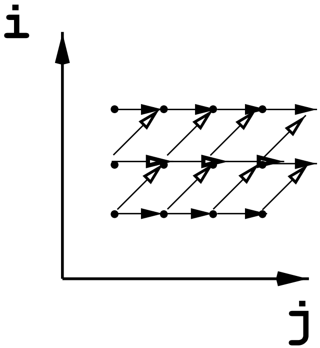
```
for (int i = 1; i < N; i++)
  for (int j = 1; j < N; j++) {
    S1 : A[j][i] = A[j-1][i-1]...
    S2 : B[j][i] = B[j-1][i]...
  }
```

Example: Iteration $(i = 2, j = 3)$

- Reads from $A[2][1], B[2][2]$.
- Writes to $A[3][2], B[3][2]$.
- So iteration (i, j) writes values read by iterations $(i, j+1), (i+1, j+1)$.

Loop B

```
for (int i = 1; i < N; i++)
  for (int j = 1; j < N; j++) {
    S1: A[j][i] = A[j-1][i-1]...
    S2: B[j][i] = B[j-1][i]...
  }
```



Example: Iteration $(i = 2, j = 3)$

- Reads from $A[2][1], B[2][2]$.
- Writes to $A[3][2], B[3][2]$.
- So iteration (i, j) writes values read by iterations $(i, j + 1), (i + 1, j + 1)$.

Loop C

```
for (int i = 1; i < N; i++)
  for (int j = 0; j < N; j++)
    S1: A[i][j] = A[i-1][j+1]...
```

Example: Iteration $(i = 2, j = 3)$

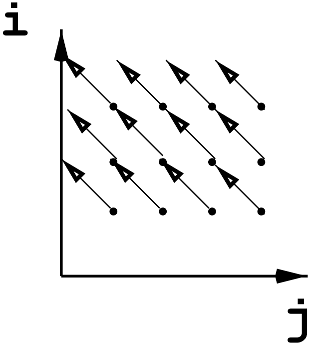
- Reads from $A[1][4]$.
- Writes to $A[2][3]$.
- So iteration (i, j) writes values read by iterations $(i + 1, j - 1)$.

Loop C

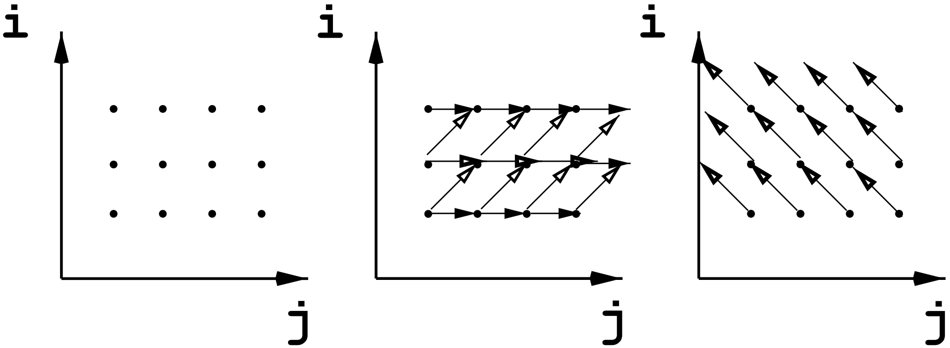
```
for (int i = 1; i < N; i++)
  for (int j = 0; j < N; j++)
    S1: A[i][j] = A[i-1][j+1]...
```

Example: Iteration $(i = 2, j = 3)$

- Reads from $A[1][4]$.
- Writes to $A[2][3]$.
- So iteration (i, j) writes values read by iterations $(i + 1, j - 1)$.



Loop-Nest Dependencies



- Wouldn't want to visualise for for more than two dimensions.
- **How can we summarize this information?**

Aggregate Dependencies via Direction Vectors

Dependency Direction

For a dependency from $S1$ in iteration $\vec{k}$ (*source*) to $S2$ in $\vec{l}$ (*sink*) where $\vec{k} \leq \vec{l}$, the elements of the *direction vector* $\vec{D}(\vec{k}, \vec{l})$ are defined by:

1. $D_i(\vec{k}, \vec{l}) =$ "<" if $k_i < l_i$,
2. $D_i(\vec{k}, \vec{l}) =$ "=" if $k_i = l_i$,
3. $D_i(\vec{k}, \vec{l}) =$ ">" if $k_i > l_i$,
4. $D_i(\vec{k}, \vec{l}) =$ "$*$" if cannot be determined.

- If the source is a write and the sink a read, then a RAW dependency.
- If the source is a read, then WAR.
- If both are writes, then WAW.

A direction vector cannot have $>$ as the first non-$=$ symbol, as that would mean that the sink depends on a future iteration.

How to compute the direction vectors

- For any two statements $S1$ and $S2$ that may access the same array A (and at least one of the accesses is a write),
- in two symbolic iterations $I^1 \equiv (i_1^1, \dots i_m^1)$ and $I^2 = (i_1^2, \dots i_m^2)$ (such that $I^1 < I^2$)
- on indices $A[e_1^1] \dots [e_n^1]$ and $A[e_1^2] \dots [e_n^2]$, respectively,
- then the direction vectors may be derived from the equations

$$\begin{cases} e_1^1 = e_1^2 \\ \dots \\ e_n^1 = e_n^2 \end{cases}$$

The system of equations models the definition of a dependency: both accesses need to refer to the same memory location!

Computing direction vectors for loop A

```
for (int i = 0; i < N; i++)
  for (int j = 0; j < N; j++)
    S1: A[j][i] = A[j][i]...
```

Computing direction vectors for loop A

```
for (int i = 0; i < N; i++)
  for (int j = 0; j < N; j++)
    S1 : A[j][i] = A[j][i] ...
```

Produces this set of equations for S_1:

$$\begin{cases} i_1 = i_2 \\ j_1 = j_2 \end{cases}$$

Computing direction vectors for loop A

```
for (int i = 0; i < N; i++)
  for (int j = 0; j < N; j++)
    S1 : A[j][i] = A[j][i] ...
```

Produces this set of equations for S_1:

$$\begin{cases} i_1 = i_2 \\ j_1 = j_2 \end{cases}$$

This implies $i_1 = i_2, j_1 = j_2$, so the direction vector is

`[=,=]`

Computing direction vectors for loop B

```
for (int i = 1; i < N; i++)
  for (int j = 1; j < N; j++) {
    S1 : A[j][i] = A[j-1][i-1]...
    S2 : B[j][i] = B[j-1][i]...
  }
```

Computing direction vectors for loop B

```
for (int i = 1; i < N; i++)
  for (int j = 1; j < N; j++) {
    S1: A[j][i] = A[j-1][i-1]...
    S2: B[j][i] = B[j-1][i]...
  }
```

Assuming (i_1, j_1) is the read from a given element of an array:

$$S_1 : \begin{cases} i_1 - 1 = i_2 \\ j_1 - 1 = j_2 \end{cases} \qquad S_2 : \begin{cases} i_1 = i_2 \\ j_1 - 1 = j_2 \end{cases}$$

Computing direction vectors for loop B

```
for (int i = 1; i < N; i++)
  for (int j = 1; j < N; j++) {
    S1: A[j][i] = A[j-1][i-1]...
    S2: B[j][i] = B[j-1][i]...
  }
```

Assuming (i_1, j_1) is the read from a given element of an array:

$$S_1 : \begin{cases} i_1 - 1 = i_2 \\ j_1 - 1 = j_2 \end{cases} \qquad S_2 : \begin{cases} i_1 = i_2 \\ j_1 - 1 = j_2 \end{cases}$$

- $i_1 > i_2, j_1 > j_2$.
- If (i_1, j_1) is source, then direction vector is [>,>], which is **illegal**.
- So direction vector: [<,<]

Computing direction vectors for loop B

```
for (int i = 1; i < N; i++)
  for (int j = 1; j < N; j++) {
    S1: A[j][i] = A[j-1][i-1]...
    S2: B[j][i] = B[j-1][i]...
  }
```

Assuming (i_1, j_1) is the read from a given element of an array:

$$S_1 : \begin{cases} i_1 - 1 = i_2 \\ j_1 - 1 = j_2 \end{cases}$$

- $i_1 > i_2, j_1 > j_2$.
- If (i_1, j_1) is source, then direction vector is [>,>], which is **illegal**.
- So direction vector: [<,<]

$$S_2 : \begin{cases} i_1 = i_2 \\ j_1 - 1 = j_2 \end{cases}$$

- Similar reasoning.
- Direction vector: [=,<]

Computing direction vectors for loop C

```
for (int i = 1; i < N; i++)
  for (int j = 0; j < N; j++)
    S1: A[i][j] = A[i-1][j+1]...
```

Computing direction vectors for loop C

```
for (int i = 1; i < N; i++)
  for (int j = 0; j < N; j++)
    S1: A[i][j] = A[i-1][j+1]...
```

Produces this set of equations for S_1:

$$\begin{cases} i_1 - 1 = i_2 \\ j_1 + 1 = j_2 \end{cases}$$

Computing direction vectors for loop C

```
for (int i = 1; i < N; i++)
  for (int j = 0; j < N; j++)
    S1 : A[i][j] = A[i-1][j+1]...
```

Produces this set of equations for S_1:

$$\begin{cases} i_1 - 1 = i_2 \\ j_1 + 1 = j_2 \end{cases}$$

Implies $i_1 > i_2, j_1 < j_2$, but which of $(i_1, j_1), (i_2, j_2)$ is the sink and which is the source?

- Picking (i_1, j_1) as source gives us `[>,<]`, which is illegal.
- So it must be `[<,>]`.

Summary

- Dependencies constrain how we can reorder the statements of a program.
- A dependency connects *source* and *sink* statements, and the source must run before the sink.
- Direction vectors are a tool we can use to qualify the dependencies in a loop nest.
- This is fiddly material. *Read the course notes!*