Dependences

CSE 501
Lecture 17
May 29, 2013
The big picture

What our goals?
• Make execution time as short as possible

The leads us to
• Achieve execution of many (all, in the best case) instructions in parallel
• Find independent instructions
Dependences

We'll focus on *data* dependences. *Control* dependences are another interesting problem.

Here a simple example:

1) \( \pi = 3.14 \)
2) \( r = 5.0 \)
3) \( \text{area} = \pi \times r \times 2 \)

We can't move statement (3) before (1) or (2) without compromising correct results.

We say there's a dependence from (1) to (3) and another from (2) to (3). Alternatively, we say that (3) depends on (1) and (2)
Dependences

Formally, there's a data dependence from statement X to statement Y (Y depends on X) if

- Both statements access the same memory location and at least one of the accesses is a write, and
- There is a feasible run-time execution path from X to Y
Classification

We classify data dependences based on load-store order:

- if a read depends on a write, we call it a *true* (or *flow*) dependence
- if a write depends on a read, we call it an *anti* dependence
- if a write depends on another write, we call it an *output* dependence

Occasionally we talk of *input* dependences, but they aren't significant for parallelization.
Dependences in loops

Loops are where the action is.

Consider these two examples:

\begin{verbatim}
   do i = 1, n
      1)  t = A(i) + B(i)
      2)  A(i+1) = t
   enddo
   1)  t = A(i) + B(i)
   2)  A(i+2) = t
enddo
\end{verbatim}

In both cases, there's a scalar true dependence from (1) to (2). Not so interesting, since our ordinary analyses tells us what we need to know.

More interesting is the true dependence from (2) around the loop to (1).

The dependence exists in both examples, but there's a significant difference.

We need a formalism to describe and distinguish such dependences.
Iteration numbers

The iteration number of a loop is equal to the value of the loop index

Definition
For an arbitrary loop in which the loop index $I$ runs from $L$ to $U$ in steps of $S$, the iteration number $i$ of a specific iteration is equal to the index value $I$ on that iteration.

For example

```
  do i = 0, 10, 2
    <some statement>
  enddo
```

Normalization is an attractive option: stride = 1 and lower bound = 0.

Wolfe likes semi-normalization: stride = 1
Iteration vectors

How about nested loops?

Need to consider the nesting level of a statement

• Given a nest of $N$ loops, the iteration vector $V$ of a particular iteration is a vector of integers that contains the iteration number for each of the enclosing loops

• Thus the iteration vector is $<i_1, i_2, ..., i_N>$ where $i_K$, $1 \leq K \leq N$, represents the iteration number for the loop at nesting level $K$. 
Iteration vectors

For example,

```
  do i = 1, 2
    do j = 1, 2
      1) <some statement>
      enddo
    enddo
  enddo
```

The iteration vector \(<2, 1>\) denotes the instance of (1) executed during the 2nd iteration of the \(i\) loop and the 1st iteration of the \(j\) loop.
Iteration space

The iteration space is the set of all possible iteration vectors for a statement

\[
\begin{align*}
\text{do } & i = 1, 2 \\
\text{do } & j = 1, 2 \\
1) & \quad <\text{some statement}> \\
\text{enddo} \\
\end{align*}
\]

enddo
endo
endo

In this case, the iteration space for (1) is

\[
\{<1, 1>, <1, 2>, <2, 1>, <2, 2>\}
\]
Ordering of iteration vectors

It's useful to define an ordering for iteration vectors

Use an intuitive, lexicographic ordering

Iteration $i$ precedes iteration $j$, denoted $i < j$, iff

- $i_k < j_k$, where $k \leq n$ and $n$ is the length of the vector, and
- $i_l = j_l$, where $l < k$
Loop dependence

There's a dependence from statement $X$ to statement $Y$ in a common nest of loops iff there exists two iteration vectors $i$ and $j$ for the nest such that

- $i < j$ or $i = j$,
- there is a path from $X$ to $Y$ in the body of the loop,
- $X$ accesses memory location $M$ on iteration $i$ and $Y$ accesses $M$ on iteration $j$, and
- one of these accesses is a write.
Transformations

We call a transformation *safe* if the transformed program has the same meaning as the original program.

A *reordering* transformation is an xform that merely changes the order of execution without adding or deleting any executions of any statements.

A reordering xform does not eliminate any dependences, but it might break a dependence, leading to incorrect behavior.

A reordering xform *preserves* a dependence if it preserves the relative execution order of the source and sink of the dependence.
Fundamental theorem of dependence

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

We say an xform is valid for some program if it preserves all dependences for the program.
Distance vectors

Consider a dependence in a loop nest of $n$ loops
  • statement $X$ on iteration $i$ is the source of the dependence
  • statement $Y$ on iteration $j$ is the sink of the dependence

The *distance vector* is a vector $d(i, j)$ of length $n$ where $d(i, j)_k = j_k - i_k$

We normalize distance vectors for loops where the step size is not 1
Direction vectors

Consider a dependence in a loop nest of $n$ loops

- statement $X$ on iteration $i$ is the source of the dependence
- statement $Y$ on iteration $j$ is the sink of the dependence

The direction vector is a vector $D(i, j)$ of length $n$ where

$$
\text{"<" if } d(i, j)_k > 0 \\
D(i, j)_k = \text{"=" if } d(i, j)_k = 0 \\
\text{">" if } d(i, j)_k < 0$$

We can combine entries to yield things like "$\leq"", "$\geq"", "$\ast"$

Distance vectors and direction vectors summarize many dependences. Distances are more precise, but are not always possible.
Direction vectors

Here's an example

\[
\begin{align*}
do \ i &= 1, \ N \\
do \ j &= 1, \ M \\
do \ k &= 1, \ L \\
1) \quad t &= A(i, j, k) + 10 \\
2) \quad A(i+1, j, k-1) &= t \\
\end{align*}
\]

There's a true dependence from (2) to (1)

- Distance vector = (1, 0, -1)
- Direction vector = (<, =, >)

Might have a combined vector, with distances where it makes sense and directions otherwise.
Direction vectors

Might have a combined vector, with distances where it makes sense and directions otherwise.

Here's an example

\[
\begin{align*}
&\text{do } i = 1, N \\
&\quad \text{do } j = 1, M \\
&\quad \quad \text{do } k = 1, L \\
&\quad \quad \quad t = A(i, j, k) + 10 \\
&\quad \quad \quad A(i+1, 2*j, kk) = t \\
&\quad \text{enddo} \\
&\text{enddo} \\
&\text{enddo}
\end{align*}
\]

There's a true dependence from (2) to (1)

• Combined vector is \((1, <, *)\)
Direction vectors

A dependence cannot exist if it has a direction vector whose leftmost non-"=" component is not "<"

We can use direction vectors to check for the legality of loop xforms:

If we look at the direction vectors for all of the dependences after applying the xform, the form is valid if none of the DVs for dependences that have their source and sink in the loop has a leftmost non-"=" component that is ">"
Loop-carried and loop-independent dependences

In a loop, if statement $Y$ depends on statement $X$, then there are 2 ways this dependence might occur:
- $X$ and $Y$ execute on different iterations - a loop-carried dependence
- $X$ and $Y$ execute on the same iteration - a loop-independent dependence
Loop-carried dependence

A statement $Y$ has a loop-carried dependence on a statement $X$ iff

- $X$ references location $M$ on iteration $i$,
- $Y$ references $M$ on iteration $j$, and
- $d(i, j) > 0$ (or $D(i, j)$ contains a "<" as leftmost non-"=" component)

For example

```c
for (i = 0; i < n; i++) {
    A[i+1] = F[i];
    F[i+1] = A[i];
}
```

There are 2 loop-carried true dependences, both with distance 1. But no loop-independent dependence; we can reorder the statements.
Level

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

For instance

```plaintext
  do i = 1, 10
    do j = 1, 10
      do k = 1, 10
        1) t = A(i, j, 2*k)
        2) A(i, j+1, k) = t
      enddo
    enddo
  enddo
enddo
```

There's a loop-carried true dependence from (2) to (1) with a direction vector (\(=, <, >\)). The level of dependence is 2.
Transformations

The direction vector can guide transforms.

In this example, the DV was (=, <, >) and the dependence was carried by the 2nd loop.

```
  do i = 1, 10
    do j = 1, 10
      do k = 1, 10
        1) t = A(i, j, 2*k)
        2) A(i, j+1, k) = t
      enddo
    enddo
  enddo
enddo
```

Implies that we can do what we will with the inner and outer loops, as long as we leave the middle loop alone.
Loop-independent dependences

Statement $Y$ has a loop-independent dependence on statement $X$ iff there exist iteration vectors $i$ and $j$ such that

- statement $X$ refers to memory location $M$ on iteration $i$,
- statement $Y$ refers to memory location $M$ on iteration $j$, and
- there's a control-flow path between $X$ and $Y$ within an iteration

For example

```
do i = 1, 10
  1) A(i) = ...
  2) ... = A(i)
enddo
```
Loop-independent dependences

Here's a more interesting example

```fortran
  do i = 1, 9
    1) A(i) = ... 
    2) ... = A(10-i)
  enddo
```

No common loops are necessary, for instance

```fortran
  do i = 1, 10
    1) A(i) = ...
    enddo
  do i = 1, 10
    2) ... = A(20-i)
    enddo
```
Simple dependence testing

Here's a simple example

\[
\text{do } i = 1, n \\
1) \quad t = A(i) + B \\
2) \quad A(i+1) = t \\
\text{enddo}
\]

• The iteration at the source (2) is denoted by \(i_0\)
• The iteration at the sink (1) is denoted by \(i_0 + \Delta i\)
• Forming an equality yields \(i_0 + 1 = i_0 + \Delta i\)
• Solving yields \(\Delta i = 1\)

So there's a loop-carried dependence from (2) to (1) with distance vector \((1)\) and direction vector \((<)\)
Simple dependence testing

Another example

```
  do i = 1, 100
    do j = 1, 100
      do k = 1, 100
        1) t = A(i, j, k+1) + B
        2) A(i+1, j, k) = t
      enddo
    enddo
  enddo
```

\[ i_0 + 1 = i_0 + \Delta i \quad j_0 = j_0 + \Delta j \quad k_0 = k_0 + \Delta k + 1 \]
\[ \Delta i = 1 \quad \Delta j = 0 \quad \Delta k = -1 \]

Distance vector = (1, 0, -1)
Direction vector = (<, =, >)
Simple dependence testing

If a loop index does not appear, its distance is unconstrained and its direction is "*"

```
  do i = 1, 100
    do j = 1, 100
      1)  t = A(i) + B(j)
      2)  A(i+1) = t
      enddo
    enddo
  enddo
```

The direction vector here is (<, *)
Simple dependence testing

"*" denotes the union of all 3 directions

```fortran
  do j = 1, 100
    do i = 1, 100
      1) t = A(i) + B(j)
      2) A(i+1) = t
    enddo
  enddo
```

(*, <) denotes \{(<, <), (=, <), (>, <)\}

We interpret (>, <) as a level-1 anti dependence with direction vector (<, >)
Parallelization and vectorization

If a loop carries no dependence, we can run it in parallel.

So loops like this

```c
    do i = 1, n
        X(i) = X(i) + C
        X(i) = X(i) + C
    enddo
```

but not like this

```c
    do i = 1, n
        X(i+1) = X(i) + C
        X(i+1) = X(i) + C
    enddo
```

Sometimes we can vectorize even if there's a loop-carried dependence.
Vectorization

If the distance is $\geq$ length of the vector registers, then we can vectorize correctly.

So loops like this

\begin{verbatim}
  do i = 1, n
    X(i+4) = X(i) + C
  enddo
\end{verbatim}

can be handled in chunks of 4, approximately like this

\begin{verbatim}
  do i = 1, n, 4
    X(i+4:i+7) = X(i:i+3) + C
  enddo
\end{verbatim}
Loop distribution

Consider this loop

\[
\begin{align*}
d & \text{do } i = 1, n \\
1) & \quad A(i+1) = B(i) + C \\
2) & \quad D(i) = A(i) + E \\
\text{enddo}
\end{align*}
\]

The loop carries a dependence between from (1) to (2), preventing trivial parallelization/vectorization. But suppose we distribute the loop...

\[
\begin{align*}
d & \text{do } i = 1, n \\
1) & \quad A(i+1) = B(i) + C \\
\text{enddo} \\
2) & \quad D(i) = A(i) + E \\
\text{enddo}
\end{align*}
\]
Loop distribution

Loop distribution won't break a *cycle* of dependences

```plaintext
  do i = 1, n
    A(i+1) = B(i) + C
    B(i+1) = A(i) + E
  enddo
```

How about this case?

```plaintext
  do i = 1, n
    B(i) = A(i) + E
    A(i+1) = B(i) + C
  enddo
```
Memory hierarchy

It's not all about parallelism.

We can use dependences to significantly improve performance code on a single processor, making better use of registers and cache.

- Scalar replacement
- Unroll and jam
Scalar replacement

Convert array references to register references to improve performance of our coloring-based allocator.

For example

\[
\begin{align*}
do & \ i = 1, \ n \\
 & \quad \text{do } j = 1, \ m \\
 & \quad \quad A(i) = A(i) + B(j) \\
 & \quad \text{enddo} \\
& \text{enddo}
\end{align*}
\]

\[
\begin{align*}
do & \ i = 1, \ n \\
 & \quad \quad t = A(i) \\
 & \quad \quad \text{do } j = 1, \ m \\
 & \quad \quad \quad t = t + B(j) \\
 & \quad \quad \text{enddo} \\
 & \quad \quad A(i) = t \\
 & \text{enddo}
\end{align*}
\]
Dependences and the memory hierarchy

- True or flow - save loads and cache misses
- Anti - save cache misses
- Output - save stores
- Input - save loads

Consistent dependences are most useful

For loop-carried dependences, we like a constant threshold (dependence distance)
Scalar replacement example

Scalar replacement with a loop-independent dependence

\[
\begin{align*}
do & \ i = 1, n \\
& \quad A(i) = B(i) + C \\
& \quad X(i) = K*A(i) \\
& enddo
\end{align*}
\]

\[
\begin{align*}
do & \ i = 1, n \\
& \quad t = B(i) + C \\
& \quad A(i) = t \\
& \quad X(i) = K*t \\
& enddo
\end{align*}
\]

Saves a load per iteration
Scalar replacement example

Scalar replacement with a loop-carried dependence spanning a single iteration

\[
\begin{align*}
\text{do } i &= 1, n \\
A(i) &= B(i-1) \\
B(i) &= C(i) + D
\end{align*}
\]

Enddo

Saves a load per iteration

\[
\begin{align*}
t &= B(0) \\
\text{do } i &= 1, n \\
A(i) &= t \\
t &= C(i) + D \\
B(i) &= t \\
\text{Enddo}
\end{align*}
\]
Scalar replacement example

Scalar replacement with a loop-carried dependence spanning multiple iterations

do i = 1, n
   A(i) = B(i-1) + B(i+1)
enddo

Saves a load per iteration

But what about those copies?
Unrolling to eliminate copies

\[ t_1 = B(0) \]
\[ t_2 = B(1) \]
\[ \text{do } i = 1, n \]
\[ \quad t_3 = B(i+1) \]
\[ \quad A(i) = t_1 + t_3 \]
\[ \quad t_1 = t_2 \]
\[ \quad t_2 = t_3 \]
\[ \text{enddo} \]
\[ \text{do } i = 1, n \% 3 \]
\[ \quad t_3 = B(i+1) \]
\[ \quad A(i) = t_1 + t_3 \]
\[ \quad t_1 = t_2 \]
\[ \quad t_2 = t_3 \]
\[ \text{enddo} \]
\[ \text{do } i = n \% 3 + 1, n, 3 \]
\[ \quad t_3 = B(i+1) \]
\[ \quad A(i+0) = t_1 + t_3 \]
\[ \quad t_1 = B(i+2) \]
\[ \quad A(i+1) = t_2 + t_1 \]
\[ \quad t_2 = B(i+3) \]
\[ \quad A(i+2) = t_3 + t_2 \]
\[ \text{enddo} \]
Unroll and jam

Remember this example?
We'd like to take advantage of the re-use of the B values.

\[
\begin{align*}
do & \quad i = 1, \quad n \\
do & \quad j = 1, \quad m \\
& \quad A(i) = A(i) + B(j) \\
& \quad A(i+0) = A(i+0) + B(j) \\
& \quad A(i+1) = A(i+1) + B(j) \\
enddo & \quad enddo & \quad enddo
\end{align*}
\]

Unroll the outer loop, then fuse the copies of the inner loop

Notice that the two uses of B are now easy to fix.
Unroll and jam

More scalar replacement

do i = 1, n, 2
  do j = 1, m
    A(i+0) = A(i+0) + B(j)
    A(i+1) = A(i+1) + B(j)
  enddo
enddo

do i = 1, n, 2
  a0 = A(i+0)
  a1 = A(i+1)
  do j = 1, m
    a0 = a0 + b0
    a1 = a1 + b0
  enddo
  b0 = B(j)
  A(i+0) = a0
  A(i+1) = a1
end
Unroll and jam

Pretty cool, but is it always legal? Or profitable?

Nope.

How do we tell?

We look at the pattern of dependences.
Unroll and jam

More deeply nested loops offer more flexibility

do i = 1, l
  do j = 1, m
    do k = 1, n
      A(i, j) += B(i, k)*C(k, j)
    enddo
  enddo
enddo

do i = 1, l, 2
  do j = 1, m, 2
    do k = 1, n
      a00 = A(i+0, j+0)
      a01 = A(i+0, j+1)
      a10 = A(i+1, j+0)
      a11 = A(i=1, j+1)
      do k = 1, n
        b0 = B(i+0, k)
        b1 = B(i+1, k)
        c0 = C(k, j+0)
        c1 = C(k, j+1)
        a00 += b0*c0
        a01 += b0*c1
        a10 += b1*c0
        a11 += b1*c1
      enddo
      A(i+0, j+0) = a00
      A(i+0, j+1) = a01
      A(i+1, j+0) = a10
      A(i+1, j+1) = a11
    enddo
  enddo
enddo
Balance

By adjusting the amount we unroll and jam, we change the loop balance, the ratio of flops to memory references.

With 2D loops, we can improve the balance. With 3D loops, we can match the machine's balance (given enough registers).

All of these ideas can have big effects on cache behavior.

Combining these (and many other xforms) can do a lot for the performance with dense linear algebra.

Finding the best combination is a tough problem. This is where the polyhedral model is supposed to help.