Compiling for Parallelism & Locality

Last Time
– Garbage Collection Implementation

Today
– Parallelism and locality
– Data dependences and loops

Example 1: Loop Permutation for Improved Locality

Sample code: Assume Fortran’s Column Major Order array layout

```
Example 2: Parallelization

Can we parallelize the following loops?

```
do i = 1,100
   A(i) = A(i)+1
endo
```
```
do i = 1,100
   A(i) = A(i-1)+1
endo
```

```
1 2 3 4 5 ...
i
```
```
1 2 3 4 5 ...
i
```

```
Yes
```
```
No
```

Example 2: Parallelization

Data Dependences

Recall
– A data dependence defines ordering relationship two between statements
– In executing statements, data dependences must be respected to preserve correctness

Example

```
s_1: a := 5;
s_2: b := a + 1;
```
```
s_1: a := 6;
s_2: b := a + 1;
```
Data Dependences and Loops

How do we identify dependences in loops?

```
do i = 1,5
   A(i) = A(i-1) + 1
endo
```

Simple view
- Imagine that all loops are fully unrolled
- Examine data dependences as before

```
A(1) = A(0) + 1
A(2) = A(1) + 1
A(3) = A(2) + 1
A(4) = A(3) + 1
A(5) = A(4) + 1
```

Problems
- Impractical
- Lose loop structure

Dependence Analysis for Loops

Big picture
- To improve data locality and parallelism we often focus on loops
- To transform loops, we must understand data dependences in loops
- Since we can’t represent all iterations of a loop, we need some abstractions
- The basic question: does a transformation preserve all dependences?

Today and Next Time
- Basic abstractions and machinery
- Friday
  - Its application to loop transformations

Data Dependence Terminology

We say statement $s_2$ depends on $s_1$
- True (flow) dependence: $s_1$ writes memory that $s_2$ later reads
- Anti-dependence: $s_1$ reads memory that $s_2$ later writes
- Output dependences: $s_1$ writes memory that $s_2$ later writes
- Input dependences: $s_1$ reads memory that $s_2$ later reads

Notation: $s_1 \delta s_2$
- $s_1$ is called the source of the dependence
- $s_2$ is called the sink or target
- $s_1$ must be executed before $s_2$

Dependences and Loops

Loop-independent dependences
```
do i = 1,100
   A(i) = B(i) + 1
   C(i) = A(i) * 2
endo
```

Dependences within the same loop iteration

Loop-carried dependences
```
do i = 1,100
   A(i) = B(i) + 1
   C(i) = A(i-1) * 2
endo
```

Dependences that cross loop iterations
Iteration Spaces

Idea
- Explicitly represent the iterations of a loop nest

Example
```
    do i = 1,6
        do j = 1,5
            A(i,j) = A(i-1,j-2)+1
        enddo
    enddo
```

Iteration Space
- A set of tuples that represents the iterations of a loop
- Can visualize the dependences in an iteration space

Distance Vectors

Idea
- Concisely describe dependence relationships between iterations of an iteration space
- For each dimension of an iteration space, the distance is the number of iterations between accesses to the same memory location

Definition
- \( v = |T - T| \)

Example
```
    do i = 1,6
        do j = 1,5
            A(i,j) = A(i-1,j-2)+1
        enddo
    enddo
```

Distance Vector: (2,1)

Protein String Matching Example

\( q = k_{1} \)
\( r = k_{2} \)
\( score[i,j] = 0 \) for the whole array

```
for i=1 to n1-1
    h[i,0] = p[i,0] = 0
    f[i,0] = -q
    for j=1 to n0-1
        f[i,j] = max(f[i,j-1], h[i,j-1]-q) - r
        h[i,j] = p[i,j-1] + max(0, f[i,j], h[i,j]) - r
        EE[i,j] = max(EE[i-1,j], HH[i-1,j], f[i,j], h[i,j])
        score[i,j] = max(score[i,j-1], h[i,j])
    endfor
endfor
return score[n1-1,n0-1]
```

Distance Vectors and Loop Transformations

Idea
- Any transformation we perform on the loop must respect the dependences

Example
```
    do j = 1,5
        do i = 1,6
            A(i,j) = A(i-1,j-2)+1
        enddo
    enddo
```

Can we permute the i and j loops?
### Idea
- Any transformation we perform on the loop must respect the dependences

### Example
```
do i = 1, 6
  do j = 1, 5
    A(i, j) = A(i-1, j-2) + 1
  enddo
enddo
```
Can we permute the `i` and `j` loops?
- Yes

### Distance Vectors and Loop Transformations

### Example
```
do i = 1, 6
  do j = 1, 5
    A(i, j) = A(i-1, j+1) + 1
  enddo
enddo
```
Kind of dependence: Flow
Distance vector: `(1, -1)`

### Exercise
```
do j = 1, 5
  do i = 1, 6
    A(i, j) = A(i-1, j+1) + 1
  enddo
enddo
```
Kind of dependence: Anti
Distance vector: `(1, -1)`

### Example 1: Loop Permutation (reprise)
```
do j = 1, 6
  do i = 1, 5
    A(j, i) = A(j, i) + 1
  enddo
enddo
```
Why is this legal?
- No dependences, so we can arbitrarily change order of iteration execution
Example 2: Parallelization (reprise)

Why can’t this loop be parallelized?

```
do i = 1,100
   A(i) = A(i-1)+1
enddo
```

Why can this loop be parallelized?

```
do i = 1,100
   A(i) = A(i)+1
enddo
```

Distance Vector: (1)

Distance Vector: (0)

Concepts

Improve performance by ...

- improving data locality
- parallelizing the computation

Data Dependences

- iteration space
- distance vectors

Transformation legality

- must respect data dependences

Next Time

Reading

- [Padua86]

Lecture

- Data dependence analysis continued