Automatic Loop Parallelization

Copyright 2015, Pedro C. Diniz, all rights reserved.
Students enrolled in the Compilers class at the University of Southern California have explicit permission to make copies of these materials for their personal use.

Acknowledgement: Some of the material in this lecture is based on class notes from CSCI595-Spring 2004 class at USC graciously provided by Dr. Jacqueline Chame and Dr. Mary Hall.
Analysis for Parallelism

- Find Data Dependences Across Loop Iterations
- Unlike Data-Flow Analysis Identify Individual Accesses, not aggregate Effects
- Abstraction: For Affine Array Index Functions
  - Linear Inequalities
  - Techniques: Linear Algebra and Integer Programming
Analysis for Parallelism

- Find Data Dependences Across Loop Iterations
- Unlike Data-Flow Analysis Identify Individual Accesses, not aggregate Effects
- Abstraction: For Affine Array Index Functions
  - Linear Inequalities
  - Techniques: Linear Algebra and Integer Programming
Data Dependence

- **Definition:** Two memory accesses are involved in a data dependence if they may refer to the same memory location and one of the references is a write (not quite complete…).

- **Note:** A Data Dependence can either be between two distinct program statements or two different dynamic executions of the same program statement.

- Two important uses of Data Dependence information:
  - **Parallelization:** if there is not a data dependence between two computations, they may execute safely in parallel.
  - **Locality:** the absence of data dependences eliminates sequential ordering constraints, allowing freedom to reorder for better data locality, also suggests “reuse”
Why is Data Dependence So Important?

- **Basic**: Need to preserve program behavior…

- **Sequential Semantics**: Each Statement Modifies the State of the Execution

- **Goal of Parallelization (reordering)**: Reach the same final state - faster!

  $\begin{align*}
  s1: & \quad a = b + c \\
  s2: & \quad c = 1
  \end{align*}$

  $\begin{align*}
  a: & \quad a_0 \quad b:b_0 \quad c:c_0 \\
  a: & \quad a_0 + c_0 \quad b:b_0 \quad c:c_0 \\
  a: & \quad a_0 + 1 \quad b:b_0 \quad c:1
  \end{align*}$
Reordering, Concurrency & Atomicity

**Reordering:** execution is sequential but order is changed

**Concurrency:** Statement execution independent at different times

**Atomicity:** Ensures each statement changes *State* consistently in a concurrent execution environment, i.e., execution is an interleaving of the execution of the individual statements.

\[
\begin{align*}
\text{s1: } & a = b + c \\
\text{s2: } & c = c + 1 \\
\end{align*}
\]

\[
\begin{align*}
& a: a_0 \quad b: b_0 \quad c: c_0 \\
& a: b_0 + c_0 \quad b: b_0 \quad c: c_0 \\
& a: b_0 + c_0 \quad b: b_0 \quad c: c_0 + 1
\end{align*}
\]
Reordering, Concurrency & Atomicity

• **Reordering**: execution is sequential but order is changed

• **Concurrency**: Statement execution independent at different times

• **Atomicity**: Ensures each statement changes State consistently in a concurrent execution environment, i.e., execution is an interleaving of the execution of the individual statements.

Reordered
*(still sequential)*

\[
\begin{align*}
\text{s1: } & \quad a = b + c \\
& \quad a: b_0 + c_0 \quad b: b_0 \quad c: c_0 \\
\text{s2: } & \quad c = c + 1 \\
& \quad a: b_0 + c_0 \quad b: b_0 \quad c: c_0
\end{align*}
\]

\[
\begin{align*}
\text{s1: } & \quad a = b + c \\
& \quad a: b_0 + c_0 \quad b: b_0 \quad c: c_0 + 1 \\
\text{s2: } & \quad c = c + 1 \\
& \quad a: b_0 + c_0 + 1 \quad b: b_0 \quad c: c_0 + 1
\end{align*}
\]
Reordering, Concurrency & Atomicity

- **Reordering**: execution is sequential but order is changed

- **Concurrency**: Statement execution independent at different times

- **Atomicity**: Ensures each statement changes *State* consistently in a concurrent execution environment, i.e., execution is an interleaving of the execution of the individual statements.

```
Reordering, Concurrency & Atomicity

Reordering: execution is sequential but order is changed

Concurrency: Statement execution independent at different times

Atomicity: Ensures each statement changes State consistently in a concurrent execution environment, i.e., execution is an interleaving of the execution of the individual statements.

Concurrent (atomic)
```

```
s1: a = b + c
    a: a_0 b: b_0 c: c_0
    a: a_0 b: b_0 + c c_0
    a: b_0 + c_0 b: b_0 c: c_0
    a: b_0 + c_0 b: b_0 + c c_0 + 1

s2: c = c + 1
    a: a_0 b: b_0 c: c_0
    a: a_0 b: b_0 + c c_0 + 1
    a: b_0 + c_0 b: b_0 c: c_0 + 1
```
Reordering, Concurrency & Atomicity

- **Reordering**: execution is sequential but order is changed

- **Concurrency**: Statement execution independent at different times

- **Atomicity**: Ensures each statement changes State consistently in a concurrent execution environment, i.e, execution is an interleaving of the execution of the individual statements.

```
Concurrent
(atomic)

s1:    a = b + c
  a: a_0  b:b_0  c:c_0

s2:    c = c + 1
  a: b_0 + c_0  b:b_0  c:c_0
  a: b_0 + c_0  b:b_0  c:c_0 + 1
```

```
s1:    a = b + c
  a: a_0  b:b_0  c:c_0

s2:    c = c + 1
  a: b_0 + c_0 + 1  b:b_0  c:c_0 + 1
```
Data Dependence for Scalars

• (A More Comprehensive) Definition: Two memory accesses are involved in a data dependence if they may refer to the same memory location.

• Types of Dependence:
  True dependence
    \[ a = \ldots \]
    \[ \ldots = a \]
  Anti-dependence
    \[ \ldots = a \]
    \[ a = \ldots \]
  Output dependence
    \[ a = \ldots \]
    \[ a = \ldots \]
  Input Dependence
    \[ \ldots = a \]
    \[ \ldots = a \]

• In General for statement \( s_i \) and \( s_j \) a Data dependence exists iff \( s_i \) and \( s_j \) refer to the same variable \( s_i \) executes before \( s_j \)
Parallelization Goal: DOALL Loops

- **DOALL Loops**: Loops whose iterations can execute concurrently (hence in any order)
  - No data dependences
  - Control and Synchronization are trivial

- Example:

  ```plaintext
  DO I = 1 TO N
      A(i) = B(i) + C(i)
  ENDDO
  ```

  ```plaintext
  DO I = 1 TO N
      spawn task({A(i)=B(i)+C(i)})
  ENDDO
  wait();
  ```
Parallelization Goal: DOALL Loops

- **DOALL Loops**: Loops whose iterations can execute concurrently (hence in any order)
  - No data dependences
  - Control and Synchronization are trivial

- Example:
  
  ```
  DO I = 1 TO N
  A(i) = B(i)+C(i)
  ENDDO
  ```

  ```
  DO I = 1 TO N
  spawn task({A(i)=B(i)+C(i)})
  ENDDO
  wait();
  ```
Preliminaries: Loop Normalization

- Normalization allows “base” framework reference for analysis
- Assumes loop iteration counts begin at “1” and step by “1”
- Loops can be normalized to ensure this property:

```
DO I = 4, 12, step 2
  A(I) = …
```

```
DO I = 1, 5
  A(I*2+2) = ...
```
Definitions about Reordering

• Definitions:
  – Two computations are equivalent if, on the same inputs,
    • they produce identical outputs
    • the outputs are executed in the same order
  – A reordering transformation changes the order of statement execution without adding or deleting any statement executions.
  – A reordering transformation preserves a dependence if it preserves the relative execution order of the dependences’ source and sink.

• Theorem:
  – Any reordering transformation that preserves every dependence in a program preserves the meaning of that program.
Iteration Space

- $n$-dimensional discrete Cartesian space for $n$ deep loops
- Iteration is represented as coordinates in iteration space
- Sequential execution order of iterations: Lexicographic order $[0,0], [0,1], ..., [0,6],[0,7], [1,1], [1,2], ..., [1,6], ...
- Iteration $I$ (a vector) is lexicographically less than $I'$, $I < I'$, iff there exists $c$ $(i_1, ..., i_{c-1}) = (i'_1, ..., i'_{c-1})$ and $i_c < i'_c$. 

```
DO I = 0, 5
    DO J = I, 7
        ...
```

```
0 \leq i 
0 \leq j 
i \leq 5 
j \leq 7
```
Distance Vectors

\[
\begin{align*}
\text{DO } & \ I = 2, \ N \\
\text{DO } & \ J = 2, \ N \\
A(I,J) & = A(I-1,J-1)+1
\end{align*}
\]

- Distance Vector = [1,1]
- A loop has a Distance Vector (DV) if there exists data dependence from a node I to a node I’, and DV = I’ - I.
- Since I’ > I, D >= 0. (D is lexicographically greater than or equal to 0).
Distance and Direction Vectors

• Distance Vectors: (infinitely large set)

\[
\begin{pmatrix}
0 & 0 & 0 & \cdots & 0 \\
0 & 1 & 2 & \cdots & n
\end{pmatrix} \begin{pmatrix}
1 & \cdots & 1 \\
-n & 0 & \cdots & n
\end{pmatrix} \cdots \begin{pmatrix}
n & \cdots & n \\
-n & 0 & \cdots & n
\end{pmatrix}
\]

• Direction Vectors: (realizable if 0 or lexicographically positive)

\([=,=],[=,<],[<,>],[<,=],[<,<]\)

• Common notation:

0 =
+
->
+/- *
More Distance Vectors Examples

DO I = 2, N
  DO J = 2, N
    A(I,J) = A(I-1,J+1)+1

DO I = 2, N
  DO J = 2, N
    A(I,J) = A(I+1,J-1)+1

DO I = 2, N
  DO J = 2, N
    A(I,J) = A(I,J+1)+1

**Question:** Which are lexicographically positive?
Parallelization: 1-Dimensional Loop

• Examples:

\[
\begin{align*}
\text{DO } J &= 1, N \\
A(J) &= A(J) + 1
\end{align*}
\]

\[
\begin{align*}
\text{DO } J &= 2, N \\
B(J) &= B(J-1) + 1
\end{align*}
\]

• Dependence (Distance and Direction) Vectors:

• Test for parallelization:

  – A loop is parallelizable if for all data dependences \( D \in D \), \( D \geq 0 \)
Loop-Carried & Loop-Independent Dep.

• A loop-carried dependence occurs between different iteration vectors.
  
  DO I = 1, N  
  A(I+1) = A(I) ...

• A loop-independent dependence occurs within the same iteration of a loop nest.
  
  DO I = 1, N  
  A(I+1) = A(I) ...
n-Dimensional Loop Nests

DO I = 1, N
    DO J = 2, N
        A(I,J) = A(I,J-1) + 1
    END DO
END DO

DO I = 2, N
    DO J = 2, N
        A(I,J) = A(I-1,J-1) + 1
    END DO
END DO

• Definition:
  \( D = (d_1, \ldots, d_n) \) is loop-carried at level \( i \) if \( d_i \) is the first nonzero element.
Test for Parallelization

The $i$th loop of an $n$-dimensional loop is parallelizable if there does not exist any level $i$ data dependences.

The $i$th loop is parallelizable if for all dependences $D = (d_1, \ldots, d_n)$, either

$$(d_1, \ldots, d_{i-1}) > 0$$

or

$$(d_1, \ldots, d_i) = 0$$
Parallelization Algorithm

- For each pair of array references within the current loop:
  - Determine if there exists a dependence between that pair

- Key points:
  - $n^2$ tests for $n$ accesses in loop!
  - a single access is compared with itself
  - includes accesses in all loops within a nest

- Requires: Good and Quick Dependence Testing Procedure
Dependence Testing

- Question so far:
  - What is the distance/direction (in the iteration space) between two accesses to the same memory location?

- Simpler question:
  - Can two data accesses ever refer to the same memory location?

```
DO I = 11, 20
  A(I) = A(I-1) + 3

DO I = 11, 20
  A(I) = A(I-10) + 1
```
Restrict to an Affine Domain

DO i = 1, N
   DO j = 2*i, 100
      A(i+2*j+3, 4*i+2*j, 3*i) = ...  
      ... = A(1, 2*i+1, j)

• Only use loop bounds and array indices which are integer linear functions of loop variables.

• Non-affine examples:
  DO i= 1, N
     DO j = 1, M
        A(i*j) = A(i*(j-1))
  DO i= 1, N
     A(i) = B(C(i))
Equivalence to Integer Programming

• Need to determine if $F(i) = G(i')$, where $i$ and $i'$ are iteration vectors, with constraints $i, i' \geq L$, $U \geq i, i'$

• Example:

\[
\text{DO I = 2, 100} \\
\quad A(I) = A(I-1)
\]

• Inequalities:

\[
0 \leq i_1 \leq 100, \quad i_2 = i_1 - 1, \quad i_2 \leq 100 \\
\text{integer vector } I, \quad AI \leq b
\]

• Integer Programming is NP-complete => Expensive
  - $O(\text{size of the coefficients})$
  - $O(n^n)$
Dependence Testing in the 80s

- Historically, simplify with inexact tests that are more efficient
- Examples: GCD test, Banerjee’s test
- 2 outcomes
  - no dependence
  - maybe a dependence
- Typically, apply a series of more powerful, inexact tests whenever a “maybe” answer is given
- May sacrifice parallelism
Modern Dependence Testing (1991)

- Derive a collection of specific, exact tests that are very efficient

- **Exact tests give two possible answers:** no dependence or definitely a dependence

- Only use inexact tests when exact tests not applicable

- Advantages:
  - exact tests are applicable most of the time
  - avoids cascading of dependence testing when dependence exists

- Example Systems: SUIF (Stanford), PFC/ParaScope (Rice)
Some Dependence Testing Terms

- **Complexity**: Number of loop indices in a subscript position (ZIV, SIV, MIV)
  
  ```
  DO I
      DO J
         DO K
         A(5, I+1, J) = A(N, I, K) + C
  ```

- **Separability**: whether a given subscript position interacts with other subscripts
  
  ```
  DO I
      DO J
         DO K
         A(I, I, J) = A(I, K, J) + C
  ```
Utility of Separability

• Can independently examine each subscript position
• No precision is lost by simply merging the independent components of a direction vector.
• Subscript positions that are not separable are called coupled.
Example of a Simple, Exact Test

- Strong SIV: An SIV subscript for loop index $I$ is strong if it has the form $<aI + c1, aI' + c2>$

- Dependence distance can be calculated exactly as follows:
  
  $d = I' - I = (c1 - c2) / a$
Dependence Testing Overview

• Partition the subscripts into separable and minimal coupled groups.

• Classify each subscript as ZIV, SIV, or MIV.

• For each separable subscript, apply appropriate dependence test. If independence is proved, DONE! Otherwise, produce a set of direction vectors.

• For each coupled group, apply a multiple subscript test and derive direction vectors.

• If any test yields independence, DONE! Otherwise, merge all direction vectors.
Effectiveness of Automatic Parallelization

- Fortran Applications *Automatically* Parallelized by the Stanford SUIF Compiler
- Yielded 50% Higher Specfp95 ratio than previously reported
Summary

• Data dependence is a fundamental concept in compilers for high-performance computing (HPC).

• Data dependence can be used to determine the safety of reordering transformations
  - preserving dependences = preserving “meaning”

• Iteration vectors, distance and direction vectors are abstractions for understanding whether reordering transformations preserve dependences

• Dependence testing has been shown to be equivalent to integer programming
  - can start with simple exact tests
  - can use integer programming techniques
  - can approximate with inexact tests