Automatic Loop Parallelization

Analysis for Parallelism

Why is Data Dependence So Important?

Reordering, Concurrency & Atomicity

Data Dependence

Analysis for Parallelism

Reordering, Concurrency & Atomicity

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!

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 data consistently in a concurrent execution environment; i.e., execution is an interleaving of the execution of the individual statements.

Reordered (still sequential)

\[
\begin{align*}
&x_0, y_0, z_0, c_0 \\
&s1: &a = b + c &s2: &c = c + 1 \\
&x_0, y_0, z_0, c_1 \\
&s1: &a = b + c &s2: &c = c + 1 \\
&x_0, y_0, z_0, c_2 \\
&s1: &a = b + c &s2: &c = c + 1 \\
\end{align*}
\]

Concurrent (atomic)

\[
\begin{align*}
&x_0, y_0, z_0, c_0 \\
&s1: &a = b + c &s2: &c = c + 1 \\
&x_0, y_0, z_0, c_1 \\
&s1: &a = b + c &s2: &c = c + 1 \\
&x_0, y_0, z_0, c_2 \\
&s1: &a = b + c &s2: &c = c + 1 \\
\end{align*}
\]

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: \( a \rightarrow b \)
  - Anti-dependence: \( a \leftarrow b \)
  - Output dependence: \( a \leftarrow b \)
  - Input Dependence: \( a \rightarrow b \)

- In General for statement \( s_i \) and \( s_j \) Data dependence exists iff \( s_i \) and \( s_j \) refer to the same variable \( a \) execute before \( s_i \)

Parallelization Goal: DOALL Loops

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

- Example:

```c
DO I = 1 TO N
    A[i] = B[i] + C[i]
ENDDO
```

```c
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:

\[
\begin{align*}
\text{DO I = 4, 12, step 2} & \quad \text{DO I = 1, 5} \\
A(I) = \ldots & \quad A(I^2-2) = \ldots
\end{align*}
\]

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 dependence’s 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.

\[
\begin{align*}
\text{DO I = 0, 5} & \quad \text{DO J = I, 7} \\
A(I,J) = \ldots
\end{align*}
\]

Distance Vectors

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

Distance and Direction Vectors

- Distance Vectors: (infinitely large set)
- Direction Vectors: (realizable if 0 or lexicographically positive)

\[
\begin{align*}
[D, \trianglelefteq, \triangleright, \trianglelefteq, \triangleright, \trianglelefteq]
\end{align*}
\]

More Distance Vectors Examples

- Distance Vectors Examples:

\[
\begin{align*}
\text{DO I = 3, N} & \quad \text{DO J = 2, N} \\
A(I,J) = A(I-1, J+1) + 1 & \quad A(I,J) = A(I+1, J-1) + 1
\end{align*}
\]

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

- Examples:
  
  \[
  \begin{align*}
  \text{DO } j &= 1, N \\
  B[j] &= B[j-1] + 1
  \end{align*}
  \]

- Dependence (Distance and Direction) Vectors:

- Test for parallelization:
  - A loop is parallelizable if for all data dependences \( D \subseteq D, D = 0 \)

Loop-Carried & Loop-Independent Dep.

- A loop-carried dependence occurs between different iteration vectors.
  
  \[
  \begin{align*}
  \text{DO } i &= 1, N \\
  \end{align*}
  \]

- A loop-independent dependence occurs within the same iteration of a loop nest.
  
  \[
  \begin{align*}
  \text{DO } i &= 1, N \\
  \end{align*}
  \]

n-Dimensional Loop Nests

\[
\begin{align*}
\text{DO } i &= 1, N \\
\text{DO } j &= 2, N \\
\end{align*}
\]

- 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

\[
\begin{align*}
(d_1, \ldots, d_i) &> 0 \\
(d_1, \ldots, d_i) &< 0
\end{align*}
\]

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?

\[
\begin{align*}
\text{DO } i &= 11, 20 \\
A[i] &= A[i-1] + 3 \quad &\text{DO } i &= 11, 20 \\
A[i] &= A[i-10] + 1
\end{align*}
\]
Restrict to an Affine Domain

\[
\text{DO } i = 1, N \\
\text{DO } j = 2i+1, 100 \\
\text{... } = A(1, 2i+1, j) \\
\]

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

Non-affine examples:

\[
\text{DO } i = 1, N \\
\text{DO } j = 1, M \\
A(i*j) = A(i*(j-1)) \\
\]

Equivalence to Integer Programming

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

Example:

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

Inequalities:

\[
0 \leq i_1 \leq 100, \quad i_2 \geq 1, \quad \exists i' \leq 100, \quad A \leq i \\
\]

Integer Programming is NP-complete \( \Rightarrow \) Expensive

- \( O(n^2) \)

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)

\[
\text{DO } i \\
\text{DO } j \\
A(i, j) = A(i, j) + C \\
\]

- Separability: whether a given subscript position interacts with other subscripts

\[
\text{DO } i \\
\text{DO } j \\
A(i, j) = A(i, j) + C \\
\]

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