Compiling for Parallelism

PPoPP Tutorial

Ron Cytron
IBM T. J. Watson Research Center

Copyright (c) 1991 IBM Corporation
Hour 1: Introduction and Vectorization

- Parallel Architectural Features
- Programs
- Anatomy of a Restructuring Compiler
- Simple Vectorization
- Control Dependence
- Vectorization with Branching
Parallel Architectural Features

Questions to be addressed:
1. Where do machines exhibit parallelism?
2. Is the parallelism effected automatically or under software control?
3. What are the costs associated with using parallelism?
What is So Hard?

- Automatic detection of parallelism
- Source-parallel languages
- Diversity of architectures
- Conflict with scalar optimization
  (redundancy can be good!)
SEA (Vector Machines)

- Single Execution of Array instructions
- Parallelism effected by:
  - Multiple processing elements driven by single control unit
  - Pipelined (floating point) arithmetic
- Instructions consume and produce:
  - Vector registers
  - Main storage

The architectural taxonomy notation is due to Kuck [38].
A pipeline decomposes a complex operation into successive segments. Consider a complex operation taking $T$ time that can be decomposed into $k$ "equal" segments. The resulting speedup for $N$ results is:

$$Speedup = \frac{NT}{T + (N - 1) \frac{T}{k}}$$

$$= \frac{kN}{N + k - 1}$$

As $N \to \infty$, speedup approaches $k$ (the number of segments).

Language and/or automatic restructuring (vectorization) must allow specification of vector operations.
Chaining [19] together two pipelines of $k_1$ and $k_2$ segments for complex operations taking time $T_1$ and $T_2$, respectively, yields another factor of

$$\frac{k_1 + k_2}{T_1 + T_2} \times \left( \frac{T_1}{k_1} + \frac{T_2}{k_2} \right)$$

in speedup. For $T_1 = T_2$ and $k_1 = k_2$, the chain yields a speedup of 2 over performing the pipelined operations separately.
MES (Multiprocessors)

- *Multiple Execution of Scalar instructions*
- Parallelism effected by:
  - Multiple processing elements each with its own control unit
  - "Wide-word" architectures: multiple functional units driven synchronously.
- Processors synchronize:
  - Through shared memory
  - By message passing
  - Implicitly (synchronous execution)
Programs

• Sequential Programs:
  • Where do programs need speeding up?
  • Where can the programmer-specified sequencing be ignored (without the user knowing we cheated)?

• Parallel Programs:
  • How is parallelism expressed?
  • How can explicit parallelism be matched to architectural features?
Program Restructuring

- Vectorization [8, 39]
- Parallelization of Loops [20, 48]
- General Parallelization [3]
- Trace Scheduling (Wide-Word) [31]

Restructuring involves program analysis and transformations to achieve efficient execution on a target architecture.
Anatomy of a Restructuring Compiler

- Source $\rightarrow$ (Annotated) Source
- Source $\rightarrow$ Object
- Intermediate Language $\rightarrow$ Object

Approach should consider level/extent of user interaction, flexibility of target language processing, and scope of input language.
Example of Source-Source: Parafrase

Parafrase, developed at the University of Illinois at Urbana-Champaign, is driven by a pass list that describes how a program should be restructured. Each transformation is reflected in the source language.

Advantages:
- Transformations easily changed
- Operation easily observed
- Generated programs easily modified
- Target easily changed

References: [27, 33, 39, 40, 43, 44, 49, 54, 55]
Source-Object

- Users maintain a single sequential source program
- Object code can consider cost benefits of vector execution
- Assertions can override compiler decisions
- Analysis is an extension of optimization algorithms

IBM VS/Fortran features outer-loop vectorization and cost analysis for efficacy of vector vs. sequential execution [46].
Example for Tutorial: PTRAN (IBM Research)

- Interprocedural analysis
- Analysis uses low-level structures, typically already computed by optimizing compilers
- Generates source (Parallel Fortran)
- Generates structures used by an optimizing back end

References: [2, 3, 14, 15, 23, 24]
Simple Vectorization

• How is the execution order affected by vectorization?
• Vectorization is achieved by "Loop Distribution".
• What is a dependence graph?
• How is loop distribution done?
Vectorization Changes the Order of Operations

Sequential

\[
\begin{align*}
\text{DO } & i=1 \text{ to } N \\
A(i) & = B(i) + C(i) \\
F(i) & = G(i) * H(i) \\
\end{align*}
\]

ENDDO

Vector

\[
\begin{align*}
A & = \text{Vadd} \ (B, \ C, \ N) \\
F & = \text{Vmult} \ (G, \ H, \ N) \\
\end{align*}
\]

\[
\begin{align*}
A(1) & = B(1) + C(1) \\
A(2) & = \\
A(3) & = \\
F(1) & = G(1) * H(1) \\
F(2) & = \\
F(3) & = \\
\cdots
\end{align*}
\]
Vectorization by Loop Distribution

Original

\[
\begin{align*}
\text{DO } i=1 \text{ to } N \\
\text{DO } j=1 \text{ to } N \\
A(i,j) & = B(i+1,j) \\
B(i+1,j) & = A(i-1,j-4) \\
B(i+1,j+1) & = A(i,j-1)
\end{align*}
\]

Distributed

\[
\begin{align*}
\text{DOPAR } i=1 \text{ to } N \\
\text{DO } j=1 \text{ to } N \\
A(i,j) & = B(i+1,j) \\
B(i+1,j+1) & = A(i,j-1)
\end{align*}
\]

\[
\begin{align*}
\text{ENDDO} \\
\text{ENDDO}
\end{align*}
\]

\[
\begin{align*}
DOPAR \ i=1 \text{ to } N \\
\text{DO } j=1 \text{ to } N \\
B(i+1,j) & = A(i-1,j-4) \\
\text{ENDDO} \\
\text{ENDDO}
\end{align*}
\]
Data Dependence Graphs

- Nodes represent statements that reference (define or use) variables.
- Edges represent flow, anti-, or output dependence.
- Each edge is labelled by a direction vector of length determined by the maximum number of loops that surround the references.
- Each direction vector component shows (for now) whether the dependence exists within an iteration (..., =, ...) or between iterations (..., ≠, ...).

Notation for describing types of data dependence is borrowed from Kuck [38].
Flow Dependence

Scalar
Def-Use Chain

T 1 X =
I
M 2 = X
E

Array
Def-Use Chains

DO i=1 to N
   X(i) =
      = X(i)
 ENDDO

DO i=1 to N
   A(i) =
      = A(i-1)
 ENDDO

i = 1

A(1) =
   = A(0)

i = 2

A(2) =
   = A(1)
Anti-Dependence

Scalar
Use-Def Chain

\[ X = X \]

Array
Use-Def Chains

\[
\begin{align*}
\text{DO } & i=1 \text{ to } N \\
& X(i) = X(i) \\
& \text{ENDDO}
\end{align*}
\]

\[
\begin{align*}
\text{DO } & i=1 \text{ to } N \\
& A(i) = A(i) \\
& \text{ENDDO}
\end{align*}
\]

\[
\begin{align*}
i = 1 & \\
A(1) = A(2) & \quad i = 2 \\
A(2) = A(3) &
\end{align*}
\]
Output Dependence

Scalar
Def-Def Chain

\[ X = \]
\[ X = \]

DO \( i = 1 \) to \( N \)
\[ X(i) = \]
\[ X(i) = \]
ENDDO

Array
Def-Def Chains

DO \( i = 1 \) to \( N \)
\[ A(i) = \]
\[ A(i+1) = \]
ENDDO

\[ i = 1 \]
\[ A(1) = \]
\[ A(2) = \]

\[ i = 2 \]
\[ A(2) = \]
\[ A(3) = \]
When is a Dependence Satisfied?

A dependence can be satisfied by reducing parallelism either with respect to loop iterations or with respect to statements.

Consider two data dependent statements, where loop $L$ is the innermost loop that contains both statements. Statement parallelism can be affected only for those statements contained by loop $L$. Iteration parallelism can be affected only for loop $L$ and its surrounding loops.
Loop-Carried Dependences

To find out which loop is affected, scan the first $L$ components of the direction vector from left to right. The loop contributing the first $\neq$ component is the affected loop [8]. If the associated statements remain in the same loop, that loop must execute sequentially.

The example to the right contains direction vectors:

$$(\neq, =, =, \ast) \text{ and } (=, =, \neq, \ast)$$

This example requires such sequentiality with respect to the $j$ and $L$ loops. The $m$ and $n$ loops can execute concurrently with respect to iterations and statements.

\[
\begin{align*}
\text{DO } j &= 1 \text{ to } N \\
\text{DO } k &= 1 \text{ to } N \\
\text{DO } L &= 1 \text{ to } N \\
\text{DO } m &= 1 \text{ to } N \\
A(j,k,L,m) &= \\
\text{ENDDO} \\
\text{DO } n &= 1 \text{ to } N \\
A(j,k,L,n) &= A(j-1,k,L,n) + A(j,k,L-1,n) \\
\text{ENDDO} \\
\text{ENDDO} \\
\text{ENDDO} \\
\text{ENDDO}
\end{align*}
\]
Loop-Independent Dependences

Again consider two references surrounded by $L$ common loops. A direction vector specifies a loop-independent dependence if none of the $L$ components contains a $\neq$.

Control dependence (described later) identifies how statements must be sequenced to honor such dependences. In the example to the right, the $m$ and $n$ loops cannot execute concurrently with each other, although the iterations of each can execute concurrently.

\[
\begin{align*}
&\text{DO } i=1 \text{ to } N \\
&\quad \text{DO } j=1 \text{ to } N \\
&\quad \text{DO } L=1 \text{ to } N \\
&\quad \quad \text{DO } m=1 \text{ to } N \\
&\quad \quad \quad A(i,j,L,m) = \\
&\quad \quad \quad \text{ENDDO} \\
&\quad \quad \text{DO } n=1 \text{ to } N \\
&\quad \quad \quad = A(i,j,L,n) \\
&\quad \quad \quad \text{ENDDO} \\
&\text{ENDDO} \\
&\text{ENDDO} \\
&\text{ENDDO}
\end{align*}
\]
Setting the Stage for Vectorization

1. Control Flow and Interval Analysis
   - Control Flow Graph [1]
   - Intervals (represent loops) [5, 47]

2. Data Flow Analysis [36]
   - Def-Use Chains
   - Def-Def Chains [22]

3. Constant Propagation and Induction Variable Recognition [21, 52]
   - Look-Aside Tables
Setting the Stage (continued)

1. Control Dependence [24, 30]
2. Data Dependence
   [6, 8-10, 41, 50, 54, 55]
3. Transformations [43]
4. Loop Distribution [38, 45]

These items are covered by this tutorial, but for now assume everything is done except loop distribution.
Inside-Out Loop Distribution Algorithm

1. For loop $vloop$, assume loops $1..(vloop - 1)$ execute sequentially and construct the dependence graph for statements $S$.

2. Form and order the strongly connected components $\Pi$ by the (acyclic) dependence relation induced by the partition to produce $\pi_1, \pi_2, \ldots, \pi_n$.

3. Write loop $vloop$ around each $\pi_i$.

4. Let $S = \Pi$
Vectorization (continued)

An inner loop with a non-recurrent singleton \( \pi \)-block is a vector loop. Of course, hardware must support the vector operation.

Some hardware can support multiple-loop vector operations. Outer loops can be incorporated into vector instructions for such architectures where the singleton \( \pi \)-block remains cycle-free.

Vector Loop

\[
\begin{align*}
\text{DO } & \text{i=1 to N} \\
A(i) &= B(i) + C(i) \\
\text{ENDDO}
\end{align*}
\]

Non-Vector Loops

\[
\begin{align*}
\text{DO } & \text{i=1 to N} \\
A(i) &= B(i) + A(i-1) \\
\text{ENDDO}
\end{align*}
\]

\[
\begin{align*}
\text{DO } & \text{i=1 to N} \\
A(i) &= \text{SQRT}(B(i)) \\
\text{ENDDO}
\end{align*}
\]

Chaining can be accomplished by recognizing ordered \( \pi \)-blocks where the contained operations can be chained.
Example of Vectorization

**Dependence Graph**

```
DO i=1 to N
  DO j=1 to N
    (S₁)
    A(i,j) = B(i+1,j)

    (S₂)
    B(i+1,j) = A(i-1,j-4)

    (S₃)
    B(i+1,j+1) = A(i,j-1)
  ENDDO
ENDDO
```
Vectorize Inner Loop

Sequential Outer Loop

DOSEQ i=1 to N
  DO j=1 to N
    (S_1)
    A(i,j) = B(i+1,j)

    (S_2)
    B(i+1,j) = A(i-1,j-4)

    (S_3)
    B(i+1,j+1) = A(i,j-1)
  ENDDO
ENDDDO

Dependence graph reflects only those dependences that must be satisfied by the inner loop.
Example (continued)

Statements S1 and S3 form a strongly connected component. The inner loop is written around those two statements, and the resulting $\pi$ block precedes the block for S2, due to the dependences from $\pi_1$ (S1 and S3) to $\pi_2$ (S2).

This transformation has preserved all dependences. Further, the second $j$ loop is non-recurrent with a single statement, and can therefore be executed in vector mode.

DO i=1 to N
  DOSEQ j=1 to N
    S1
    S3
  ENDDO
  DOPAR j=1 to N
    S2
  ENDDO
ENDDO
Outer Loop Vectorization

DO i=1 to N
    DOSEQ j=1 to N
        S1
        S3
    ENDDO
    DOPAR j=1 to N
        S2
    ENDDO
ENDDO

Outer Loop Parallel?

The dependence graph reflects only those dependences that must be satisfied by the outer loop.
The outer $i$ loop is in both cases parallel. For S2, this means a double-loop vector instruction could be executed. For S1 and S3, no vector instruction is possible unless loops $i$ and $j$ could be interchanged.

```
DOPAR i=1 to N
  DO j=1 to N
    S1
    S3
  ENDDO
ENDDO

DOPAR i=1 to N
  DOPAR j=1 to N
    S2
  ENDDO
ENDDO
```
Control Dependence

• Sequential programs necessarily specify an ordering among operations
• Control dependence determines where such ordering can be relaxed
• Control dependence is useful for parallelization and for vectorizing loops with branches.
Uses of Control Dependence

Control dependence determines that the calls to procedure $P$ and $R$ execute under the same conditions. Where allowed by data dependence, the two calls could be executed concurrently.

The assignment statements also execute under the same conditions. For a wide-word architecture, a compiler might try to schedule the two assignments in one instruction.

Also, the assignment to $A$ occurs with respect to the $DO$ loop. Thus, a vector instruction can be issued for the assignment, controlled by the predicate $c$.

```
DO i=1 to N
  IF (c) THEN
    CALL P
    X = Y + Z
    .
    .
    .
    IF ()
    ENDIF
    .
    .
    A(i) = B(i)+C(i)
    .
    .
  Q = V * S
  CALL R
ENDIF
ENDDO
```
Control Dependence for VLIW

Original Program

\[ x = y + z \]
\[ \text{IF (p) then} \]
\[ w = q * z \]
\[ \text{else} \]
\[ w = y * z \]
\[ \text{endif} \]

\[ r = q * s \]

Trace Scheduled

\[ x = y + z \]
\[ w = y * z \]
\[ r = q * s \]
\[ \text{IF (p) then fix up} \]
\[ \text{Control Dependence} \]
\[ \text{Packed} \]
\[ x = y + z \]
\[ r = q * s \]
\[ \text{IF (p) then} \]
\[ w = q * z \]
\[ \text{else} \]
\[ w = y * z \]
\[ \text{endif} \]

Better first to schedule identically control dependent operations and then pick a trace of the original program [32].
A control flow graph $\text{CFG}(\text{Nodes}, \text{Edges})$ consists of a set of Nodes that represents operations of a program, and a set of Edges that represents transfer of control among the nodes.

Often, straight-line control flow is summarized by nodes that correspond to basic blocks. In data flow parlance, local data flow analysis is computed only within such blocks, but global data flow analysis accounts for flow among basic blocks.
Control Dependence Definition

Let $G$ be a control flow graph, and let $X$ and $Y$ be nodes in $G$. $Y$ is control dependent on $X$ iff

1. there exists a directed path $P$ from $X$ to $Y$ with all $Z$ in $P$ (excluding $X$ and $Y$) post-dominated by $Y$ and
2. $X$ is not post-dominated by $Y$.

In other words, from $X$, there is some edge that definitely causes $Y$ to execute, and there is also some path that avoids executing $Y$. 

- 37 -
Control Dependence Algorithm

Two stages [24, 25]:
1. Construct *immediate control dependence* from the control flow graph and the post-dominator tree.
2. Propagate control dependence using the post-dominator tree.

---

A node N **post-dominates** a node M if all paths from M to **Exit** must pass through N. (Executing M implies execution of N.)
Immediate Control Dependence

IF (a)
THEN x = y + z
ELSE
    p = m - s
    IF (b)
    THEN w = y - z
    ELSE q = r * s
    ENDIF
    s = m * n
ENDIF
r = u * v

Stmt Depends on
-----------
IF (a) Entry
x = a
p = a
w = b
q = b
s =
r =
IF (b)

X is immediately control dependent on A iff control can branch from A directly to X but X does not post-dominate A.
A control dependence propagates up the post-dominator tree until a node is reached that post-dominates the source of the control dependence.
Vectorization with Branching (No Exits)

1. Construct a dependence graph with the union of data dependence edges and control dependence edges.

2. Form SCC as with normal loop distribution.

3. Control dependence edges between the resulting Π blocks necessitate control vectors for vector execution.

4. IF-statements remain, where necessary, for control between vector operations.
Example

DO i=1 to N
  IF (p) then
    q =
    DO j=1 to N
      IF (q) then
        A(i,j) =
        ENDIF
      B(i,j) =
      ENDDO
    p =
    ENDIF
  ENDDO

\[
\begin{align*}
S1 & \rightarrow S2 \\
S2 & \rightarrow S3 \\
S3 & \rightarrow S4 \\
S4 & \rightarrow S5 \\
S5 & \rightarrow S6
\end{align*}
\]
Loop Distribution

The inner loop contains a branch that must be translated into a control vector for the assignment to $A$. The assignment to $B$ post-dominates the IF and so is unaffected.

The outer loop contains a recurrence on $p$, preventing loop distribution.

DO i=1 to N
    IF (p) then
        q =
        DO j=1 to N
            m(j) = if (q) then 1 else 0
        ENDDO
    ENDDO

DOPAR j=1 to N
    A(i,j) = WHERE m(j)
ENDDO
DOPAR j=1 to N
    B(i,j) =
ENDDO
p =
ENDDO
ENDDO
Hour 2: Data Dependence

- Sample Transformation [7, 53, 55]
- Interchanging Vectors [55]
- Dependence Equations
- Context
- Decision Algorithms
- Interprocedural Information

[4, 11-14, 18, 51]
Why Loop Interchange?

- May need to bring a parallel loop inside to obtain vector execution.
- May want to bring a better loop inside (longer length, better stride, etc.).
- Beneficial for scalar processors (increase locality).
- Like many transformations, loop interchange is based on direction vectors.
Iteration Space (2-d)

DO i=1 to 3
   DO j=1 to 3
      1(1,1)  2(1,2)  3(1,3)
      4(2,1)  5(2,2)  6(2,3)
   END DO
END DO
    7(3,1)  8(3,2)  9(3,3)

Loop interchanging must preserve dependences.
OK to Loop Interchange

DO i=1 to 3
  DO j=1 to 3
    A(i,j) = A(i,j-1)
  ENDDO
ENDDO

1 → 2 → 3
4 → 5 → 6
7 → 8 → 9

Rule: Must hit tail of arrow before hitting head
Also OK to Loop Interchange

DO i=1 to 3
   DO j=1 to 3
      A(i,j) =
          = A(i-1,j)
   ENDDO
ENDDO
Still O.K to Loop Interchange

\[
\begin{align*}
\text{DO } i = 1 \text{ to } 3 \\
\text{DO } j = 1 \text{ to } 3 \\
A(i,j) &= A(i-1,j-1) \\
\text{ENDDO} \\
\text{ENDDO}
\end{align*}
\]
Not OK to Loop Interchange

DO i=1 to 3
  DO j=1 to 3
    A(i,j) =
    = A(i-1,j+1)
  ENDDO
ENDDO

But, you could reverse the inner loop and then interchange!
Direction Vectors

The earlier distinction of $\neq$ and $=$ is too crude to test for loop interchanging (and other transformations).

Direction vectors characterize the temporal nature of a dependence with respect to a set of loops (intervals).

In testing a pair of references, a given loop $i$ is modelled by two values, $x$ and $y$, that represent (potentially different) values of $i$ at each reference.
Direction Vector Components

\[(z_1, z_2, \ldots, z_d)\]

Each component \(z_k\) represents the relation of instances \(x_k\) and \(y_k\) of an iteration variable \(i_k\).

1. If \(z_k = "="\), then \(x_k = y_k\) and the dependence holds within the same iteration of loop \(k\).

---

Recall that a given component reflects activity only with respect to the associated loop.
Components (continued)

2. If $z_k = "<"$, then the dependence holds from some iteration of loop $k$ to a subsequent iteration.

3. If $z_k = ">)$, then the dependence holds from some iteration of loop $k$ to an earlier iteration.

For convenience, let $z_k = (*)$ represent the union of $(<)$, $(=)$, and $(>)$. In such cases, $x_k$ and $y_k$ could assume any value in the iteration space.
Single-Loop Direction Vectors

Let $x$ and $y$ represent instances of a single loop variable index $i$. In comparing these instances, the space is divided into regions where the instance of $x$ precedes $y$ ($x < y$), where the dependence holds within an iteration ($x = y$), and where the instance of $x$ occurs after $y$ ($x > y$).

In testing from a definition to a use, a dependence in the $<$ space is a flow dependence (the definition occurs before the use. A dependence in the $>$ space is an anti-dependence (the definition occurs after the use).

A direction vector of (*) specifies no ordering.
Loop Interchange Revisited

The flow dependence from the definition to use of $A$ has a direction vector of $(<,>)$.

When constructing a direction vector from a definition to a use,
- Loop-carried flow dependences always have their left-most non-$=$ dependence as $<$.  
- Loop-carried anti-dependences always have their left-most non-$=$ dependence as $>$. For loop-independent dependences, the topological order of the def and use determines flow or anti-dependence.

\[
\begin{align*}
\text{DO } & i = 1 \text{ to } 3 \\
\text{DO } & j = 1 \text{ to } 3 \\
A(i,j) & = A(i-1,j+1) \\
\text{ENDDO} \\
\text{ENDDO}
\end{align*}
\]
When is Loop Interchange Legal?

- Exchanging two loops swaps their direction vector components.
- Reversing a loop changes the "sign" of the component: $< \rightarrow >$ or $> \rightarrow <$.
- In the transformed program, the order of writes before reads and other writes must be preserved.

Simple test: Don't exchange loops $k$ and $k + 1$ if there is a component where $(\ldots, z_k, z_{k+1}, \ldots)$ is $(\ldots, <, >, \ldots)$. 
More Loop Interchanging

The flow dependence \((<, <, >)\) remains a flow dependence even if the two inner loops are exchanged.

The resulting direction vector is \((<, >, <)\).
Computing Data Dependence

1. Construct a *dependence equation* that models a dependence of interest.
2. Establish a *context* in which the dependence equation should be tested.
3. Invoke a *decision algorithm* to test the equation for solutions in the provided context.
Dependence Equations

Data flow analysis already identified def-use and def-def pairs of interest.

Dependence analysis can then be viewed as an augmentation of the def-use or def-def arcs.

Issues in forming dependence equations:
• Scope of analysis
• Behavior of references
We want to model two independent references: one at the definition and one at the use.

Thus, we are interested in two values for $i$, and we let these be represented by $x$ at the definition and $y$ at the use.

The dependence then models when the two subscripts could refer to the same element of $A$:

\[ 3x + 5 = 10y - 3 \]
\[ 3x - 10y = -8 \]

```plaintext
DO i=1 to 10
   A(3*i+5) = A(10*i-3)
ENDDO
```
Forming Dependence Equations

For linear subscripts, the instances can be specified as:

\[ f(x_1, x_2, \ldots, x_d) = \sum_{k=1}^{d} a_k x_k + a_0 \]

and

\[ g(y_1, y_2, \ldots, y_d) = \sum_{k=1}^{d} b_k y_k + b_0, \]

The dependence equation is then:

\[ \sum_{k=1}^{d} (a_k x_k - b_k y_k) = b_0 - a_0 \]

\[ \text{DO } i_1 = 1 \text{ TO } N_1 \]
\[ \ldots \]
\[ \text{DO } i_d = 1 \text{ TO } N_d \]
\[ A(f(i_1, \ldots, i_d)) = \ldots \]
\[ = A(g(i_1, \ldots, i_d)) \]
\[ \text{ENDDDO} \]
\[ \ldots \]
\[ \text{ENDDDO} \]

\[ a_k, b_k, a_0, \text{ and } b_0 \text{ are integer constants.} \]
Multiple Subscripts

Approaches:

- Linearize the subscripts and form a single dependence equation.
- Form a system of equations, one for each subscript.
- Both of the above.

There are various trade-offs associated with these approaches. This will be discussed later.
We desire only relevant solutions to a dependence equation:

1. Solutions must be integer-valued.
2. Solutions must assign values to the free variables that lie within the range of their associated iteration variables.
3. Solutions must assign values that respect a given direction vector.

\[
\begin{align*}
\text{DO } i=1 \text{ to } 10 \\
A(3i+5) &= A(10i-3) \\
\text{ENDDO}
\end{align*}
\]

Although the dependence equation \(3x - 10y = -8\) has infinite solutions, relevant solutions (considering points 1 and 2 above) assign \(x\) and \(y\) integer values in the range \((1,2,\ldots,10)\).
A dependence equation is first tested at the most general level (*,*,...,*), where no temporal ordering is asserted between instances of any surrounding loop iteration variable.

The decision algorithm(s) are conservative, but if any can prove independence with respect to a node of the direction vector tree, then testing need not proceed to the children of that node [14].

Where independence cannot be shown, the decision algorithms are applied to increasingly refined direction vectors.

Eventually, either independence can be shown or a direction vector cannot be further refined. The most prevalent decision algorithms are the GCD [37] and Banerjee-Wolfe [10, 55]
GCD Decision Algorithm

The GCD test determines if a dependence equation has any integer solution(s). Such solutions may fail to respect a given direction vector or lie within the iteration space.

In the example shown to the right, the GCD test finds that the dependence equation:

\[ 2x - 2y = 1 \]

has no integer solutions \( x \) and \( y \).

DO i=1 to N
   A(2i) =
   = A(2i+1)
ENDDO
Factoring Context into GCD

Note that for the "=" direction vector, \( x_k = y_k \) and the coefficients can be combined into a single term of the dependence equation. In refining direction vectors, the GCD test can be reapplied where an element of a direction vector becomes "=". For example, the dependence equation:

\[
3x + 4y = 8
\]

apparently has solutions for \( x < y \) and \( x > y \) (and therefore has solutions for "*"), but for \( x = y \), the dependence equation becomes:

\[
7x = 8
\]

which has no integer solutions.

\[
\text{DO } i=1 \text{ to } N \\
A(3i+12) = A(-4i+20)
\]

\text{ENDDO}
GCD Test

Input:

A dependence equation.
A direction vector \( Z = (z_1, z_2, ... , z_d) \) as the context in which the dependence equation should be tested.

Algorithm:

for \( k = 1 \) to \( d \) do

if \( z_k = "=" \)

then \( t_k = a_k - b_k \)

else \( t_k = \gcd(a_k, b_k) \)

end for

if \( \gcd(t_1, t_2, ..., t_d) | (b_0 - a_0) \)

then return DEPENDENCE

else return INDEPENDENCE

The probability of \( \gcd(i_1, i_2) = 1 \) is \( \frac{6}{\pi^2} \) or roughly 60\% for \( i_1 \) and \( i_2 \) chosen at random \([17]\)
Geometric Decision Algorithm

• Decides if the dependence equation has (real) solutions in the iteration space that satisfy the supplied direction vector.

• This test can be modified to test for “funny” iteration spaces.

• The test is exponential in the depth of interval nesting.

• The test is very easily understood and simple to program.
1. Generate all points that could be of interest (for a single loop, there are 8 such points).

2. Eliminate points that do not respect the direction vector (point number 3 does not respect $x < y$).

3. Evaluate the dependence equation at each remaining point.

4. If the resulting evaluations are all of the same sign, then the equation is outside of the specified region and independence is shown. Otherwise, dependence must be assumed [10, 55].
Consider a term:

\[(a_k x_k - b_k y_k)\]

Now, minimize and maximize this term w.r.t. the iteration space and direction vector:

\[LB_k \leq (a_k x_k - b_k y_k) \leq UB_k\]

Doing this for each term and summing yields:
\[ \sum \text{LB}_k \leq \sum (a_k x_k - b_k y_k) \leq \sum \text{UB}_k \]

From the dependence equation, the sum of the variable terms must equal the constant term, so the inequality becomes:

\[ \sum \text{LB}_k \leq (b_0 - a_0) \leq \sum \text{UB}_k \]

Where this inequality does not hold, the dependence equation does not pass through the iteration space, and independence is shown. Otherwise, dependence must be assumed.
Example of Banerjee-Wolfe Inexact

$x$ and $y$ are both instances of $i$.

$(x + y)$ is minimized for $x = y = 1$

$(x + y)$ is maximized for $x = y = 100$

Inequality:

\[ 2 \leq -5 \leq 200 \]

Fails! Independence is shown

DO i=1 to 100
  A(i+10) =
  = A(-i+5)
ENDDO

Dependence Equation:

\[ x + y = -5 \]
Minimizing Terms

Consider a term $ax - by$, where $x$ and $y$ represent instances of an iteration variable that ranges from 1 to $N$. Due to linearity, the term obtains its minimum at extremes of the iteration range. The example on the right reaches a minimum for $x = N$ and $y = 1$.

However, where $x$ must maintain some relation to $y$, values between 1 and $N$ are of interest. For $x < y$, the minimum could occur at $(x = 1, y = 2)$ or $(x = N - 1, y = N)$.

$$\text{DO } i=1 \text{ to } N$$
$$A(-3i+5) =$$
$$= A(-8i-20)$$

**EndDO**

Dependence Equation

$$-3x + 8y = -25$$

The choice depends on the relative values of the coefficients $a$ and $b$. 

- 73 -
Each of the inexact tests (GCD, Geometric, and Banerjee-Wolfe) fails to capture all aspects of context. An exact test determines if integer solutions exist for a dependence equation, where any solution both lies in the iteration space and respects the supplied direction vector. An example where the coupling of inexact tests fails is shown on the right. The equation has integer solutions, but all solutions lie outside the region of interest.

More often, a lack of compile-time information causes spurious dependences than does imprecision of decision algorithms.
Difficulties

The sign of $K$ determines flow or anti-dependence.

$\text{DO } i=1 \text{ to } N$
$\text{T}(i) = T(i+K)$
$\text{ENDDO}$

Upper bound not constant, yet no anti-dependence exists. Symbolic dependence test helps.

$\text{DO } i=1 \text{ to } N$
$\text{T}(i) = T(i+N)$
$\text{ENDDO}$

Terms cancel in dependence equation. Symbolic arithmetic helps.

$\text{DO } i=1 \text{ to } N$
$\text{T}(M-i) = T(M-i-1)$
$\text{ENDDO}$
Interprocedural Information

- Forward propagation of constants and alias information
- Backward propagation of MOD and USE information

Without interprocedural analysis, worst-case assumptions prevail as to the effects of FOO on A and B. Suppose FOO references only \( A(y), y \leq i \). If FOO does not modify B, then no cycle exists and the loop can be distributed.

The effects of a procedure on variables invisible at a call site must also be considered.

```
DO i=1 to N
  A(i) = B(i)
  CALL FOO (A, B, i)
ENDDO
```

```
DO i=1 to N
  A(i) = random()
ENDDO
```
Hour 3: Multiprocessing

- Nested Parallelism
- Privatization
- Synchronization
- Memory Hierarchies
Why Multiprocess?

- Vector speedup is limited by the ability to decompose an operation into pipeline segments.
- Multiprocessors are more general-purpose: multiprogramming can soak up excess MIPS.
- Programs have parallelism in outer loops, between loops and statements, across procedure calls, etc.

We focus here on how to parallelize operations, not whether it's economically a good idea.
DOALL Loops

The iterations of a DOALL loop can execute in any order. No synchronization is required, except for control of the iteration variable.

In terms of allowable dependences, a DOALL loop may not carry any dependence. Thus, a dependence must either be satisfied by an outer sequential loop or be loop-independent.

```
DO i=1 to N
  DOALL j=1 to N
    A(j) = B(j) + C(j)
    C(j) = A(j) + D(j)
    F(i,j) =
    F(i-1,j)
  ENDDOALL
ENDDO
```
Loop Distribution to Obtain DOALL Loops

DO i=1 to N
    DO j=1 to N
        A(i,j) =
        = A(i-1,j)
        = A(i,j)
    ENDDO
ENDDO
ENDDO

DO i=1 to N
    DOALL j=1 to N
        A(i,j) =
        = A(i,j)
    ENDDO
ENDDO
ENDDO
ENDDO

Just like distribution for vectorization, except loops are distributed around a (=,=,...,=) dependence. Ideally, DOALL loops should be interchanged to the outside.
DO ALL Loop Distribution (continued)

DO i=1 to N
   DO ALL j=1 to N
      A(i,j) = A(i,j)
   ENDDO

DO ALL j=1 to N
      A(i,j) = A(i-1,j)
 ENDDO
 ENDDO

Distribution with fusion and loop interchanging to minimize barrier synchronization is NP-Complete [16].
Multiprocessing Can be More Efficient

SIMD

DOPAR i=1 to N
    v statements
ENDDO

DO i=1 to N
    S−v statements
ENDDO

T = v + N(S−v)

using N processors

MIMD

DOPIPE i=1 to N

    Seg 1    V_1   12345678
    Seg 2    V_2   1234567
    .        .     12345
    Seg v    V_v   12

Seg v+1    S−v   1
ENDDO

T = S + (N − 1)(S−v)
T = N(S−v) + v

using v+1 processors!

The pipeline has better processor utilization yet achieves the same speedup [29].
How to Pipeline?

1. Like vectorization, form Π-blocks of dependence graph.
2. Each Π-block becomes a segment of the pipeline.
The execution time is then determined by the "longest"-executing segment.

Optimization: try to combine segments to minimize number of segments without increasing execution time (NP-Complete).
Nested Parallelism

1. Maximal concurrency is determined by control dependence:
   - All loops assumed DOALL
   - All identically forward control dependent nodes execute concurrently

2. Correct concurrency is determined by data dependence.

3. Effective concurrency is formed by partitioning.


From IBM's Parallel Fortran: statements within a Case execute sequentially, but Cases execute concurrently [35].
Example: Original Program

FORWARD CONTROL DEPENDENCE GRAPH

DO i = 1 to N
  x = f00(y, z)
  IF (Q .EQ. 5.0) THEN
    A(i) = x + y
    B(i) = c(i) + d(i)
  ENDIF
ENDDO

The effects of back-edges are ignored for statement concurrency.
Forward Control Dependence

Computed by the normal control dependence algorithm [23], using a modified control flow graph:

- Each interval is augmented with an $All$Exits node.
- An edge is added from $All$Exits to the target of each exit edge from the interval.
- Any back edge to the interval header is redirected to the $All$Exits node.

For structured control flow graphs, the corresponding forward control dependence graph is a tree; otherwise, a DAG obtains.
Example: Maximal Concurrency

DOALL i=1 to N
Parallel Cases
  Case
  X = FOO(Y, Z)
  Case
  IF (Q .EQ. 5.0) THEN
    Parallel Cases
    Case
    A(i) = X + Y
    Case
    B(i) = C(i) + D(i)
    End Cases
  ENDIF
End Cases
ENDDO
End Cases

The processes shown above assume the IF-THEN branch is taken.
Eliminating Dependences

• Some dependences are due to storage use rather than to actual flow of values.

• By increasing storage, some dependences can be eliminated:
  • Scalar expansion turns scalars into arrays.
  • Privatization causes processes to allocate storage for instances of variables.

Self anti-, output, and even some flow dependences are satisfied by pipeline execution. This can be incorporated into context.
Expansion vs. Privatization

\[
x(0) = x
\]

\[
\text{DOVEC } i=1 \text{ to } N
\]
\[
x(i) = B(i)
\]
\[
A(i) = x(i)
\]
\[
\text{ENDDO}
\]

\[
\text{DOALL } i=1 \text{ to } N
\]
\[
\underline{\text{Private } x}
\]
\[
\underline{\text{Copy IN/OUT}}
\]
\[
x = B(i)
\]
\[
A(i) = x
\]
\[
\text{ENDDOALL}
\]

The number of private instances of \(x\) depends on the number of processes executing the DOALL. Similarly, scalar expansion can be effected through vector registers.
Satisfying Dependences

Dependence can be satisfied:

- *Explicitly:* synchronization is inserted so that the source and sink of a dependences are properly coordinated.

- *Implicitly:* concurrency is reduced so that the involved statements cannot execute concurrently.

---

These two methods are actually endpoints of a continuum.
All loop-independent dependences between the subtrees are satisfied by implicit synchronization, but the critical path length may be increased.
Sequencing to Satisfy Dependences

- If a dependence is loop-carried by loop $i$, then the iterations of loop $i$ must execute sequentially.
- If the dependence is loop-independent (could be between loops), then [34]:
  1. Find the least common ancestor (LCA) in the control dependence graph of the nodes involved in the dependence.
  2. Identically control dependent children of the LCA node must execute sequentially if they are on a path to the data dependent nodes.
If $X$ is privatized with respect to the loop, then the only remaining dependence is the loop-independent flow dependence. The LCA of the definition and use of $X$ that participate in this flow dependence is the DO node. If the assignment to $X$ and the IF statement are sequenced, then no explicit synchronization is necessary.
Storage Hierarchies

- Dependences dictate how data should be *logically* shared among processes.
- The trend toward large-scale multiprocessors has exposed physically local memories that must be managed by software.

The goal is to maximize use of fast local memories while retaining program correctness.
Example: Programmable Caches

References to certain addresses cause data to reside in cache. Cache management instructions:

Post: Data associated with an address is copied back to global memory. The processor's cache retains its copy of the data.

Invalidate: Data associated with an address is marked invalid. Global memory is unaffected.

Flush: Combined Post and Invalidate
Problem Statement

- Initially, assume ALL data is temporarily cacheable
- Use compile-time analysis to place cache management instructions
- Examine the profitability (potential uses of cached data), marking some data cacheable and some data non-cacheable.

Basic idea: analyze program to optimize placement of data movement instructions [26].
Processor-Crossing Dependences

When the source and sink of a flow dependence can execute in different processors, then cache actions are necessary.

Dependence is loop carried by $i$ loop: serializing that loop satisfies the dependence.

Synchronization alone is insufficient!

\[
\begin{align*}
\text{DO } & i=1 \text{ to } N \\
\text{DOALL } & j=1 \text{ to } N \\
A(i,j) & = \quad = A(i-1,j) \\
\text{ENDDOALL} \\
\text{END}
\end{align*}
\]
If P2 fails to define X locally, then it should use P1's value for X.
Solved as a very busy expressions problem for $X$, where a single use is imagined at the synchronization point.

Reduction in network traffic over hardware schemes.
Invalidate

In general, invalidation instructions could be moved from a use site to somewhere after the synchronization point for the dependence causing the invalidation, if the following conditions hold:

1. A processor executes the invalidation instruction if the use site is reached.
2. The address(es) referenced at the use site can be generated at the invalidation point.
Invalidation Example

INVALID (X)

if () then X = ...
else
endif

.
.
.
= X

if () then X = ...
else INVALID (X)
endif

.
.
.
= X

Would like to put invalidation instructions exactly where needed
Data Flow Solution

The data flow values assigned at a given point $E$ are:

**VALID** All paths from the start of P2 to $E$ contain a killing definition or invalidation of $X$.

**NODEF** There are no definitions of $X$ on any path from P2 to $E$.

**PRES** Some path from the start of P2 to $E$ defines, yet fails to kill, $X$.

If $IN$ is the data flow value on entry to node $N$ and $N$ can either $KILL(X)$, $PRESERVE(X)$, or $NODEF(X)$, then

$$OUT(N) = f(IN, \text{Action}(N))$$

follows:

$$f(\text{VALID}, \text{Action}(N)) = \text{VALID}$$
$$f(\text{PRES}, \text{KILL}(X)) = \text{VALID}$$
$$f(\text{PRES}, \text{PRESERVES}(X)) = \text{PRES}$$
$$f(\text{PRES}, \text{NODEF}(X)) = \text{PRES}$$
$$f(\text{NODEF}, \text{KILL}(X)) = \text{VALID}$$
$$f(\text{NODEF}, \text{PRESERVES}(X)) = \text{PRES}$$
$$f(\text{NODEF}, \text{NODEF}(X)) = \text{NODEF}$$
Meet Lattice

Path 2

<table>
<thead>
<tr>
<th>Meet</th>
<th>VALID</th>
<th>NODEF</th>
<th>PRES</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>VALID</td>
<td>VALID</td>
<td>PRES</td>
</tr>
<tr>
<td>a</td>
<td>NODEF</td>
<td>VALID</td>
<td>NODEF</td>
</tr>
<tr>
<td>t</td>
<td>PRES</td>
<td>PRES</td>
<td>PRES</td>
</tr>
</tbody>
</table>

Where VALID meets NODEF to produce VALID, an invalidation instruction is placed on the NODEF edge.
if ()
then $X =$
else INVALID($X$)
endif

if ()
then ....
else $X =$
endif

$= X$
Look what the cat dragged in

\[
\text{INVALID}(X) = Y = X
\]

If \( X \) and \( Y \) are in the same cache line, then the reference to \( Y \) brings in a copy of \( X \) as well.
The instruction streams shown to the right execute asynchronously, so the fetches and stores of the shared variables $X$ and $Y$ may interleave in any manner.

Assuming $X$ and $Y$ are initially "0", the values fetched for $X$ by P2 could be either "0" or "1". However, if P1's fetch of $Y$ happens to precede P2's store of $Y$, then P2 should see "1" when it fetches $X$. If $X$ remains in cache from P2's first reference, then the value of "0" would be seen.

An example of such a program is Lamport's exclusion algorithm [42].
Analysis can determine where caching shared variables violates correctness of the interleaving semantics [28]: A graph is constructed with undirected arcs for cross-process references to shared variables and directed arcs for program flow within a process. Cycles in this graph indicate where program flow must be reflected by storage accesses. The solution is shown to the right. P2 must invalidate X after the first fetch. Y must be written prior to the second fetch of X.

Without such analysis, only one shared variable could be cached at a time (in registers, cache, or any form of local memory).
The run-time flexibility of traditional cache protocols can be augmented to support parallel programs [28]. Suppose each of P2's statements is conditionally executed. Correctness is maintained by invalidating X after the first fetch. Y is marked dirty, but not posted, if Y is stored. If the final fetch of X occurs, then if Y is dirty, it must first be written to shared memory.

\[ \text{P2} \]

```plaintext
if () then
    = X
    _INVALID(X)
endif

if () then
    Y = 1
    _DIRTY(Y)
endif

if () then
    POST-IF-DIRTY()
    = X
```
Summary

• Architectural concurrency is still evolving
• Programming solution is probably a combination of source and restructured concurrency
• Many techniques are easily incorporated into existing compilers
• Architectural support required for effective use of concurrency
Bibliography


William Harrison. An Overview of the Structure of Parallel Processing.

AUG 1987

Reilly Gupta and Mary Lou Soffa. A Recognizable LIW Architecture

June 1984

The ACM Symposium on Compiler Construction. 37 - 47, Montreal,


Languages and Systems, 319-349, July 1987.

J. Ferenceti, K. Ohelstein, and J. Werner. The Program Dependence

M. S. Thesis

James Russell Beckman. Parallel Loop Constructs for


Caching Correctness and Optimization. SIGPLAN. 1986. Submitted

Ron Cytron, Dennis Shasha, and Marc Snir. Software Controlled

Reconfiguring Compilers on Program Performance for High-Speed


Ron Cytron, David J. Kuck, and Alex W. Vedernikov. The effect of

Management of Programmable Caches, (Exhausted Abstraction).

Ron Cytron, Steve Katoz, and Kevin P. Macauliffe. Automatic

Programming Languages. Proc. 25th Annual ACM Symposium on Principles of


Ron Cytron and Jeannine Ferenceti. An Improved Control Dependence

Renaming for Parallelism Detection and Storage Allocation.

1986 International Conference on Parallel

Proceedings of the 1986 International Conference on Parallel


Ron Cytron and Jeannine Ferenceti. What's in a Name? Or the Value of


24.

25.

26.

27.

28.

29.

30.

31.

32.

33.
February 1971. Report 7-1-424

Bruce R. Leasure. Compiling Serial Languages for Parallel Machines.

No. 504 PR-85-5. UTL-EQ-85-8005

Bruce Leasure. The Parareuse Project's Fortran Analyzer Major Module


Lamprion: A Fast Mutual Exclusion Algorithm. ACM Transactions


L. Lamport. The Parallel Execution of DIO Loops. Communications of


Compumac 80 (Fourth International Computer Software and Applications


Ken Kennedy, A Survey of Data Flow Analysis Techniques. In Seven


IBM Parallel Fortran Language and Library Reference. International


Wilson C. Hieh. Extacting Parallelism from Sequential Programs.

55.


54.


53.


52.


51.


50.


49.


48.

B. Simons and A. Munshi, Scheduling Loops on Processors: A Design for Optimizations of the Bitvectoring Class, New York University, September 1979, Computer Science Department Report No. 17.

47.

J. T. Schwartz and M. Sharir, A Design for Optimizations of the