Compilation Considerations for Parallel and Vector Architectures

A Few Sources


Bernstein’s Conditions (1966)

- For a statement S:
  - $\text{IN}(S) = \text{set of variables, registers, or locations used by } S$
  - $\text{OUT}(S) = \text{set written to by } S$
  - $S_1; S_2$ (sequence) is equivalent to $S_1 || S_2$ (parallel) provided that
    - $\text{OUT}(S_1) \cap \text{OUT}(S_2) = \emptyset$
    - $\text{OUT}(S_1) \cap \text{IN}(S_2) = \emptyset$
    - $\text{OUT}(S_2) \cap \text{IN}(S_1) = \emptyset$

Data Dependence

- Expresses constraints on parallel execution, as derived from sequential execution semantics
- Types of Dependence (Kuck, Wolfe, et al.):
  - Flow dependence
  - Anti dependence
  - Output dependence

Flow Dependence

- A variable assigned to in one statement is used in a later one:
  - $A = 5$
  - $B = A \times A$

Anti Dependence

- A variable use in one statement is assigned to in a later one:
  - $B = A \times A$
  - $A = 5$
Output Dependence

- A variable assigned to in one statement is later re-assigned to:
  
  \[
  A = B \times B \\
  A = 5
  \]

Removable Dependences

- Anti Dependence and Output Dependence are **removable**
- They are artifacts of using variables as if memory location, rather than purely for their values.
- Flow Dependence is **not removable**; it expresses essential precedence.
- Clarification of whether location- or value-based dependency is being considered will be left to context.

Notation

- \( S_1 \delta f S_2 \) means \( S_2 \) is flow dependent on \( S_1 \)
- \( S_1 \delta a S_2 \) means \( S_2 \) is anti dependent on \( S_1 \)
- \( S_1 \delta o S_2 \) means \( S_2 \) is output dependent on \( S_1 \)

Dependence Relations determine a Partial Order on Statement Execution

- Statements must be done in sequence
- Statements can be done in either order, or in parallel

Location- vs. Value-Based

- Consider
  
  \[
  \begin{align*}
  A &= 5 \\
  B &= A + 7 \\
  A &= 99 \\
  C &= A \times 2
  \end{align*}
  \]

By using a different variable, the dependency is removed

- Consider
  
  \[
  \begin{align*}
  A &= 5 \\
  B &= A + 7 \\
  AA &= 99 \\
  C &= AA \times 2
  \end{align*}
  \]
Loops add to the Challenge

- Consider for \( K = 1 \) to \( 10 \)
  \[
  S_1(K) \quad A[K] = B[K]
  \]
  - Conclude: All instances \( S_1(K) \) can be done concurrently (since no arrows).

Loops add to the Challenge

- Consider for \( K = 2 \) to \( 10 \)
  \[
  S_1(K) \quad A[K] = A[K-1]
  \]
  - Conclude: All instances \( S_1(K) \) must be done in sequence.

Larger offsets allow more concurrency

- Consider for \( K = 3 \) to \( 10 \)
  \[
  \]
  - \( S_1(3) || S_1(4) \) is possible
  - \( S_1(K) || S_1(K+1) \) is possible, \( K = 3, 5, \ldots \)

“Forward” Offsets

- Consider for \( K = 1 \) to \( 9 \)
  \[
  S_1(K) \quad A[K] = A[K+1]
  \]
  - Conclude: All instances \( S_1(K) \) must be done in sequence (if location-based assumption used).

We can Transform the Previous Example

- For \( K = 1 \) to \( 9 \)
  \[
  S_0(K) \quad B[K] = A[K+1]
  \]
  - \( S_0(1) \)
  - \( S_0(2) \)
  - \( S_0(9) \)

Transformation reduces sequence constraints

- For \( K = 1 \) to \( 9 \)
  \[
  S_0(K) \quad B[K] = A[K+1]
  \]
  - \( S_0(1) \)
  - \( S_0(2) \)
  - \( S_0(9) \)

F90 style:

\[
B(1:9) = A(2:10)
A(2:10) = B(2:10)
\]
The type of transformation just shown can be automated.

This is done routinely in compilers for high-performance machines.

Parallel Execution of Loops Strategy

- Try to issue different instances of a loop body to separate processing elements.
- Generally loops occur nested; try to find appropriate nesting level where different instances can be issued.

Similar issue to Parallelization: Vectorization

- Vector machines:
  - Exploit fine-grain parallel operations (+, *, /) on vector elements
  - Typically done with vector registers
  - Vectorizing concentrates on inner loop
  - Parallelizing concentrates on outer loops (coarser grain)

Example of Loop Vectorization

- do K = 1 to N

Vectorizes to (using F90 notation):

- D(1:N) = A(2:N+1)*5
- A(1:N) = B(1:N) + C(1:N)

Example of Loop Vectorization

- do K = 1 to N

Vectorizes to (using F90 notation):

- D(1:N) = A(2:N+1)*5
- A(1:N) = B(1:N) + C(1:N)

Dependence Distance

- The dependence in the previous example can be summarized:
  \[ S_0(K) \delta_{i} S_1(K) \]

- This essentially says:
  - The \( i \)th iteration of \( S_0 \) must be done before the \( i+1 \)th iteration of \( S_1 \).
Dependence Distance

- In general, there may be a different set of dependence distances for each array:
  
  \[
  \begin{align*}
  S_0(K) & : A[K] = B[K-1] \\
  S_1(K) & : B[K] = A[K]
  \end{align*}
  \]

  Each places a constraint on loop restructuring.

- \( \delta_f(1) \) for A \( \delta_f(0) \) for B

Direction Vectors

- Less precise than Dependence Distances, but frequently used:
  - \( \delta_f(<) \) used in place of \( \delta_f(n) \) where \( n > 0 \)
  - \( \delta_f(=) \) used in place of \( \delta_f(0) \)
  - \( \delta_f(>) \) used in place of \( \delta_f(n) \) where \( n < 0 \)

  Advantage of using > is that \( n \) might not be fixed, as in:

  \[
  \begin{align*}
  \text{do } K = 2 \text{ to } 10 \\
  C[K] & = A[K]
  \end{align*}
  \]

  Here the dependence distance increases with \( K \).

Example

- \( \delta_f(=) \) or \( \delta_f(<) \)

\[
\begin{align*}
S_1 & : C[K] = A[K+1]
\end{align*}
\]

- \( \delta_f(1) \) or \( \delta_f(<) \)

\[
\begin{align*}
S_1 & : C[K] = A[K+1]
\end{align*}
\]

doacross

- \( \delta_f(=) \)

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

- is optimized to

\[
\begin{align*}
\text{doacross } K = 1 \text{ to } N \\
\end{align*}
\]

doacross Example

- Original loop

\[
\begin{align*}
\text{do } K = 1 \text{ to } N \\
\end{align*}
\]

- is optimized to

\[
\begin{align*}
\text{doacross } K = 1 \text{ to } N \\
\end{align*}
\]
Non-doacross Example

Original loop
\[
\begin{align*}
  & \text{do } K = 2 \text{ to } N \\
  & \quad A[K] = C[K] \\
\end{align*}
\]

\[\delta_{(>)}\]

\[
\begin{align*}
  & \text{cannot be optimized using doacross alone.}
  \\
  & \text{We could provide additional synchronization on the use of } A[K-1] \text{ to do it, but it wouldn't be pure doacross.}
\end{align*}
\]

Loops that “Carry” Dependence

As we saw, loops having only \(\delta_{(>)}\) are optimizable using doacross.

A loop with \(\delta_{(<)}\) or \(\delta_{(>)}\) carries the/a dependence that constrains parallel execution.

Nested Loops

For nested loops, a vector of dependences is used, e.g. \(\delta_{(<)}\) or \(\delta_{(=)}\) with one component per loop nest.

When loops are nested, the outermost loop with a \(\delta_{(=)}\) or \(\delta_{(>)}\) carries the dependence.

Nested Loop Example

\[
\begin{align*}
  & \text{do } K = 2 \text{ to } N \\
  & \quad \text{do } J = 2 \text{ to } N \\
\end{align*}
\]

\[\delta_{(=)} \delta_{(<)}\]

The inner loop carries the dependence for A; no loop carries the dependence for B.

Therefore the outer can be parallelized using doacross.

Nested Loop Example

\[
\begin{align*}
  & \text{do } K = 2 \text{ to } N \\
  & \quad \text{do } J = 2 \text{ to } N \\
\end{align*}
\]

\[\delta_{(=)} \delta_{(<)}\]

Exercise

How to parallelize:

\[
\begin{align*}
  & \text{do } K = 2 \text{ to } N \\
  & \quad \text{do } J = 2 \text{ to } N \\
\end{align*}
\]
Exercise

- How to parallelize:
  
  \[
  \begin{align*}
  & \text{do } K = 2 \text{ to } N \\
  & \hspace{1em} \text{do } J = 2 \text{ to } N \\
  \end{align*}
  \]

- The outer loop carries the dependency

Exercise

- do \(K = 2\) to \(N\)
  
  \[
  \begin{align*}
  & \text{do } J = 2 \text{ to } N \\
  & \hspace{1em} A[K, J] = C[K, J] \\
  & \hspace{1em} B[K, J] = A[K-1, J]
  \end{align*}
  \]

- Parallel:
  
  \[
  \begin{align*}
  & \text{do } K = 2 \text{ to } N \\
  & \hspace{1em} \text{doacross } J = 2 \text{ to } N \\
  \end{align*}
  \]

Loop Interchanging

- do \(K = 1\) to \(N\)
  
  \[
  \begin{align*}
  & \text{do } J = 1 \text{ to } N \\
  \end{align*}
  \]

- \(S_1 \delta^f_{(<, <)} S_1\) implies inner loop cannot be vectorized.
- No dependencies of form \(\delta^f_{(<, >)}\) implies loops can be interchanged

Loop Interchanging

- do \(K = 1\) to \(N\)
  
  \[
  \begin{align*}
  & \text{do } J = 1 \text{ to } N \\
  \end{align*}
  \]

- do \(J = 1\) to \(N\)
  
  \[
  \begin{align*}
  & \text{do } K = 1 \text{ to } N \\
  \end{align*}
  \]

- Now have \(\delta^f_{(<, _)}\)

Loop Interchanging

- do \(J = 1\) to \(N\)
  
  \[
  \begin{align*}
  & \text{do } K = 1 \text{ to } N \\
  \end{align*}
  \]

- Now have \(\delta^f_{(<, _)}\)
- Execute as:
  
  \[
  \begin{align*}
  & \text{do } J = 1 \text{ to } N \\
  \end{align*}
  \]