Compiling for Advanced Architectures

In this lecture, we will concentrate on compilation issues for compiling scientific codes. Typically, scientific codes

- Use arrays as their main data structures.
- Have loops that contain most of the computation in the program.

As a result, advanced optimizing transformations concentrate on loop level optimizations. Most loop level optimizations are source–to–source, i.e., reshape loops at the source level.

Today, we will talk about briefly about

- Dependence analysis
- Vectorization
- Parallelization
- Blocking for cache
- Distributed–memory compilation
Dependence — Overview

dependence relation: Describes all statement-to-statement execution orderings for a sequential program that must be preserved if the meaning of the program is to remain the same. There are two sources of dependences:

**data dependence**

\[ S_1 \quad \text{pi} = 3.14 \]
\[ S_2 \quad r = 5.0 \]
\[ S_3 \quad \text{area} = \text{pi} \times r^{\ast\ast}2 \]

**control dependence**

\[ S_1 \quad \text{if (t .ne. 0.0) then} \]
\[ S_2 \quad a = a/t \]
\[ \text{endif} \]

How to preserve the meaning of these programs? Execute the statements in an order that preserves the original load/store order.
Dependence — Overview

**Definition** — There is a data dependence from statement $S_1$ to statement $S_2$ ($S_1\delta S_2$) if

1. Both statements access the same memory location, and
2. There is a run–time execution path from $S_1$ to $S_2$.

**Data dependence classification**

"$S_2$ depends on $S_1$" — $S_1\delta S_2$

- **true (flow) dependence**
  occurs when $S_1$ writes a memory location that $S_2$ later reads

- **anti dependence**
  occurs when $S_1$ reads a memory location that $S_2$ later writes

- **output dependence**
  occurs when $S_1$ writes a memory location that $S_2$ later writes

- **input dependence**
  occurs when $S_1$ reads a memory location that $S_2$ later reads. Note: Input dependences do not restrict statement (load/store) order!
Dependence — Where do we need it?

We restrict our discussion to data dependence for scalar and subscripted variables (no pointers and no control dependence).

Examples:

\[
\begin{align*}
\text{do } I &= 1, 100 \\
& \text{do } J = 1, 100 \\
& \quad A(I,J) = A(I,J) + 1 \\
& \quad \text{enddo} \\
& \text{enddo}
\end{align*}
\]

\[
\begin{align*}
\text{do } I &= 1, 100 \\
& \text{do } J = 1, 100 \\
& \quad A(I,J) = A(I+1,J) + 1 \\
& \quad \text{enddo} \\
& \text{enddo}
\end{align*}
\]

**vectorization**

\[
\begin{align*}
A(1:100:1,1:100:1) &= A(1:100:1,1:100:1) + 1 \\
A(1:99,1:100) &= A(2:100,1:100) + 1
\end{align*}
\]

**parallelization**

\[
\begin{align*}
\text{doall } I &= 1, 100 \\
& \text{doall } J = 1, 100 \\
& \quad A(I,J) = A(I,J) + 1 \\
& \quad \text{enddo} \\
& \quad \text{implicit barrier sync.} \\
& \text{enddo}
\end{align*}
\]

\[
\begin{align*}
\text{doall } I &= 1, 99 \\
& \text{doall } J = 1, 100 \\
& \quad A(I,J) = A(I+1,J) + 1 \\
& \quad \text{enddo} \\
& \quad \text{implicit barrier sync.} \\
& \text{enddo}
\end{align*}
\]
Dependence Analysis

Question

Do two variable references never/maybe/always access the same memory location?

Benefits

• improves alias analysis
• enables loop transformations

Motivation

• classic optimizations
• instruction scheduling
• data locality (register/cache reuse)
• vectorization, parallelization

Obstacles

• array references
• pointer references
Vectorization vs. Parallelization

**vectorization** — Find parallelism in innermost loops; fine-grain parallelism

**parallelization** — Find parallelism in outermost loops; coarse-grain parallelism

- Parallelization is considered more complex than vectorization, since finding coarse-grain parallelism requires more analysis (e.g., interprocedural analysis).

- Automatic vectorizers have been very successful
Dependence Analysis for Array References

\[
\begin{align*}
\text{do } & I = 1, 100 \\
& A(I) = \\
& = A(I-1) \\
\text{enddo}
\end{align*}
\]

\[
\begin{array}{cccccc}
& & & & & \\
& & & & & \\
1 & 2 & 3 & 4 & 5 & \ldots \\
& & & & & \\
& & & & & \\
& & & & & \\
& & & & & \\
\end{array}
\]

\[
\text{do } I = 1, 100 \\
A(I) = \\
= A(I) \\
\text{enddo}
\]

\[
\begin{array}{cccccc}
& & & & & \\
& & & & & \\
1 & 2 & 3 & 4 & 5 & \ldots \\
& & & & & \\
& & & & & \\
& & & & & \\
& & & & & \\
\end{array}
\]

A \textbf{loop-independent} dependence exists regardless of the loop structure. The source and sink of the dependence occur on the same loop iteration.

A \textbf{loop-carried} dependence is induced by the iterations of a loop. The source and sink of the dependence occur on different loop iterations.

\textit{Loop-carried dependences can inhibit parallelization and loop transformations}
Dependence Testing

Given

\[ \text{do } i_1 = L_1, U_1 \]
\[ \ldots \]
\[ \text{do } i_n = L_n, U_n \]

\[ S_1 = A(f_1(i_1, \ldots, i_n), \ldots, f_m(i_1, \ldots, i_n)) = \ldots \]
\[ S_2 = \ldots = A(g_1(i_1, \ldots, i_n), \ldots, g_m(i_1, \ldots, i_n)) \]

A *dependence* between statement \( S_1 \) and \( S_2 \), denoted \( S_1 \delta S_2 \), indicates that \( S_1 \), the *source*, must be executed before \( S_2 \), the *sink* on some iteration of the nest.

Let \( \alpha \& \beta \) be a vector of \( n \) integers within the ranges of the lower and upper bounds of the \( n \) loops.

Does \( \exists \alpha \leq \beta \), s.t.

\[ f_k(\alpha) = g_k(\beta) \quad \forall k, 1 \leq k \leq m \]
Iteration Space

\begin{verbatim}
  do I = 1, 5
    do J = I, 6
      ...
    enddo
  enddo

1 \leq I \leq 5
I \leq J \leq 6

\end{verbatim}

- lexicographical (sequential) order for the above iteration space is

\[(1, 1), (1, 2), \ldots, (1, 6)
(2, 2), (2, 3), \ldots, (2, 6)
\ldots
(5, 5), (5, 6)\]

- given \(I = (i_1, \ldots, i_n)\) and \(I' = (i'_1, \ldots, i'_n)\),

\[I < I' \text{ iff}
(i_1, i_2, \ldots, i_k) = (i'_1, i'_2, \ldots, i'_k) \& i_{k+1} < i'_{k+1}\]
Distance & Direction Vectors

\begin{align*}
\text{do } & I = 1, N \\
\text{do } & J = 1, N \\
S_1 & A(I,J) = A(I,J-1) \\
\text{endo} \\
\text{endo} \\
\text{do } & I = 1, N \\
\text{do } & J = 1, N \\
S_2 & A(I,J) = A(I-1,J-1) \\
S_3 & B(I,J) = B(I-1,J+1) \\
\text{endo} \\
\text{endo} \\
\end{align*}

Distance Vector = number of iterations between accesses to the same location

Direction Vector = direction in iteration space (\(=, <, >\))

\begin{align*}
S_1 & \delta S_1 \\
S_2 & \delta S_2 \\
S_3 & \delta S_3
\end{align*}
Which Loops are Parallel?

\[
\begin{align*}
&\text{do } I = 1, N \\
&\quad \text{do } J = 1, N \\
&S_1 \quad A(I,J) = A(I,J-1)
\end{align*}
\]

\[
\begin{align*}
&\text{do } I = 1, N \\
&\quad \text{do } J = 1, N \\
&S_2 \quad A(I,J) = A(I-1,J-1)
\end{align*}
\]

\[
\begin{align*}
&\text{do } I = 1, N \\
&\quad \text{do } J = 1, N \\
&S_3 \quad B(I,J) = B(I-1,J+1)
\end{align*}
\]

- a dependence \( D = (d_1, \ldots, d_k) \) is carried at level \( i \), if \( d_i \) is the first nonzero element of the distance/direction vector

- a loop \( l_i \) is parallel, if \( \forall \) a dependence \( D_j \) carried at level \( i \)

\[
\begin{array}{c|c|c}
& \text{distance vector} & \text{direction vector} \\
\hline
\forall D_j & d_1, \ldots, d_{i-1} > 0 & d_1, \ldots, d_{i-1} = "<"
\\
\text{OR} & d_1, \ldots, d_i = 0 & d_1, \ldots, d_i = "="
\end{array}
\]
Approaches to Dependence Testing

- can we solve this problem exactly?
- what is conservative in this framework?
- restrict the problem to consider index and bound expressions that are linear functions

⇒ solving general system of linear equations in integers is NP-hard

Solution Methods

- inexact methods
  - Greatest Common Divisor (GCD)
  - Banerjee’s inequalities

- cascade of exact, efficient tests (fall back on inexact methods if needed)
  - Rice
  - Stanford

- exact general tests (integer programming)
Loop Transformations

Goal

• modify execution order of loop iterations
• preserve data dependence constraints

Motivation

• data locality
  (increase reuse of registers, cache)
• parallelism
  (eliminate loop-carried deps, incr granularity)

Taxonomy

• loop interchange
  (change order of loops in nest)
• loop fusion
  (merge bodies of adjacent loops)
• loop distribution
  (split body of loop into adjacent loops)
• strip-mine and interchange (tiling, blocking)
  (split loop into nested loops, then interchange)
Loop Interchange

\[
\begin{align*}
d & \text{do } I = 1, N \\
d & \text{do } J = 1, N \\
S_1 & \quad A(I, J) = A(I, J-1) \\
S_2 & \quad B(I, J) = B(I-1, J-1) \\
\text{endo} \\
\text{endo}
\end{align*}
\]

⇒ loop interchange ⇒

\[
\begin{align*}
d & \text{do } J = 1, N \\
d & \text{do } I = 1, N \\
S_1 & \quad A(I, J) = A(I, J-1) \\
S_2 & \quad B(I, J) = B(I-1, J-1) \\
\text{endo} \\
\text{endo}
\end{align*}
\]

Loop interchange is safe \textit{iff}

- it does not create a lexicographically negative direction vector \((1,-1) \rightarrow (-1,1)\)

⇒ Benefits

- may expose parallel loops, incr granularity
- reordering iterations may improve reuse
Loop Fusion

\[
\begin{align*}
\text{do } i &= 2, N \\
S_1 & \quad A(i) = B(i) \\
\text{do } i &= 2, N \\
S_2 & \quad B(i) = A(i-1)
\end{align*}
\]

\[\implies \text{loop fusion} \implies\]

\[
\begin{align*}
\text{do } i &= 2, N \\
S_1 & \quad A(i) = B(i) \\
S_2 & \quad B(i) = A(i-1)
\end{align*}
\]

Loop fusion is safe \textit{iff}

- no loop-independent dependence between nests is converted to a backward loop-carried dep

(\text{would fusion be safe if } S_2 \text{ referenced } a(i+1) ?)

\[\Rightarrow \text{Benefits}\]

- reduces loop overhead
- improves reuse between loop nests
- increases granularity of parallel loop
Loop Distribution

\[
\begin{align*}
  \text{do} & \quad \text{i} = 2, N \\
  S_1 & \quad A(i) = B(i) \\
  S_2 & \quad B(i) = A(i-1)
\end{align*}
\]

\[\implies\text{loop distribution} \implies\]

\[
\begin{align*}
  \text{do} & \quad \text{i} = 2, N \\
  S_1 & \quad A(i) = B(i) \\
  \text{do} & \quad \text{i} = 2, N \\
  S_2 & \quad B(i) = A(i-1)
\end{align*}
\]

Loop distribution is safe iff

- statements involved in a cycle of true deps (recurrence) remain in the same loop, and
- if \(\exists\) a dependence between two statements placed in different loops, it must be forward

\[\Rightarrow\text{Benefits}\]

- necessary for vectorization
- may enable partial/full parallelization
- may enable other loop transformations
- may reduce register/cache pressure
**Data Locality**

**Why locality?**
- memory accesses are expensive
- exploit higher levels of memory hierarchy by reusing registers, cache lines, TLB, etc.
- locality of reference ⇔ reuse

**Locality**
- temporal locality: reuse of a specific location
- spatial locality: reuse of adjacent locations (cache lines, pages)

**What locality/reuse occurs in this loop nest?**

```plaintext
do i = 1, N  
do j = 1, M  
   A(i) = A(i) + B(j)
```

![Diagram of A and B with arrows indicating operations and data flow]
Strip-Mine and Interchange (Tiling)

\[
\begin{align*}
&\text{do } i = 1, N \\
&\quad \text{do } j = 1, M \\
&\quad \quad A(i) = A(i) + B(j) \\
\Rightarrow &\text{ Strip Mine} \Rightarrow \\
&\text{do } i = 1, N \\
&\quad \text{do } jj = 1, M, T \\
&\quad \quad \text{do } j = jj, jj+T-1 \\
&\quad \quad \quad A(i) = A(i) + B(j) \\
\Rightarrow &\text{ Interchange} \Rightarrow \\
&\text{do } jj = 1, M, T \\
&\quad \text{do } i = 1, N \\
&\quad \quad \text{do } j = jj, jj+T-1 \\
&\quad \quad \quad A(i) = A(i) + B(j)
\end{align*}
\]

Strip mining is always safe, with interchange it

- changes shape of iteration space
- can exploit reuse for multiple loops
Loop Transformations to Improve Reuse

Assumptions

cache architecture (simple):
– one word cache lines,
– LRU replacement policy,
– fully associative cache,
– $M >$ cache size

Analysis

Original loop nest

A

B

A(i) — N cache misses, one for each outer iteration (cold misses); reuse for inner iterations
B(j) — N*M cache misses due to LRU policy (capacity misses)
Loop Transformations to Improve Reuse

Transformed loop nest — strip mining and interchange

$A(i) - N \times M / T$ cache misses (conservative)

$B(j) - M$ cache misses; once element is in cache, it stays in cache until all computation using it is done

Comparison

$N + N \times M$ misses vs. $M + \frac{N \times M}{T}$ misses

$\Rightarrow$ tradeoff decision

Note:

- strip–mine and interchange is similar to unroll–and–jam
- typically, cache architectures are more complex
- scalar replacement transformation for registers
Compiling for Distributed–Memory Multiprocessors

Assumptions

• Data parallelism has been specified via data layout;
  Example
  \textit{align} A(i) \textit{with} B(i)
  \textit{distribute} A(\text{block}) \textit{on} 4 \textit{procs}

• Compiler generates SPMD code (single program, multiple data)

• Compiler uses \textit{owner–computes} rule

Compiler challenges

• Has to manage local name spaces (there is no global name space!)

• Insert necessary communication

• \textit{Goal}: Minimize cost of communication while maximizing parallelism
Compiling for Distributed-Memory Multiprocessors

Example

real A(100), B(100)
do i = 1, 99
   A(i) = A(i+1) + B(i+1)
endo

real A(26), B(26) // 1 element overlap to the right
if my$proc == 3 then my$up=24 else my$up=25 endif
if my$proc > 0 then
   send( A(1), my$proc - 1) //send to left
   send( B(1), my$proc - 1) //send to left
endif
if my$proc < 3 then
   receive( A(26), my$proc + 1) //receive from right
   receive( B(26), my$proc + 1) //receive from right
endif
do i = 1, my$up
   A(i) = A(i+1) + B(i+1)
endo