Class Information

INFORMATION and REMINDERS

• HW7 deadline extension? Friday, April 21, 11:59pm.

• Project 2
  – Deadline extension? Monday, April 24, 11:59pm
  – Use plt-r5rs - racket version v6.5 (command line interpreter); do not include #lang racket from DrRacket.

• FINAL EXAM
  1. Monday, May 8, 4:00-7:00pm, College Ave. Campus
  2. CONFLICTS? Need to know as soon as possible; there are fixed rules to resolve conflicts
Loop-level Parallelism

We will concentrate on compilation issues for compiling scientific codes. Some of the basic ideas can be applied to other application domains as well. 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.
**Dependence — Definitions**

**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:

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

\[
\begin{align*}
\text{do } & I = 1, 99 \\
\text{do } & J = 1, 100 \\
& A(I,J) = A(I+1,J) + 1 \\
\text{enddo} \\
\text{enddo}
\end{align*}
\]

**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**

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

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

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

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

Loop-carried dependences can inhibit parallelization and loop transformations.
Dependence Testing

Given

\[
\begin{align*}
&\text{do } i_1 = L_1, U_1 \\
&\quad \ldots \\
&\quad \text{do } i_n = L_n, U_n \\
&S_1 \quad A(f_1(i_1, \ldots, i_n), \ldots, f_m(i_1, \ldots, i_n)) = \ldots \\
&S_2 \quad \ldots = A(g_1(i_1, \ldots, i_n), \ldots, g_m(i_1, \ldots, i_n))
\end{align*}
\]

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.

\[
\text{Does } \exists \alpha \leq \beta, \text{ s.t. } f_k(\alpha) = g_k(\beta) \quad \forall k, 1 \leq k \leq m ?
\]
Iteration Space

\[
\begin{align*}
&\text{do } I = 1, 5 \\
&\quad \text{do } J = I, 6 \\
&\quad \ldots \\
&\quad \text{enddo} \\
&\text{enddo} \\
&1 \leq I \leq 5 \\
&I \leq J \leq 6
\end{align*}
\]

- 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 \( I' = (i'_1, \ldots, i'_n) \),

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

\[\text{Distance Vector} = \text{number of iterations between accesses to the same location}\]

\[\text{Direction Vector} = \text{direction in iteration space (}=, <, >)\]

- Distance Vector: \(S_1\delta S_1\)
- Direction Vector: \(S_2\delta S_2\)
- Distance Vector: \(S_3\delta S_3\)
Next Lecture

Things to do:

- Dependence analysis
- OpenMP tutorial on our website
- More on automatic vectorization / parallelization