INFORMATION and REMINDERS

• Homework 7 is due Friday, December 1 at 11:59pm.

• New second programming project deadline: Monday, December 4, 11:59pm.

• Midterm sample solutions are available.

  Midterm grade challenge deadline:
  December 11.

• Final exam: Wednesday, December 20, 4:00-7:00pm, in our classroom.

DO YOU HAVE A CONFLICT?

http://sasundergrad.rutgers.edu/forms/final-exam-conflict

  – More than two (2) final exams on one calendar day
  – More than two (2) final exams scheduled in consecutive periods
  – Two final exams scheduled for the same exam period.
Review: 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.

We will talk about

- Dependence analysis
- Vectorization
- Parallelization
Review: Project and OpenMP

Two important issues while specifying the parallel execution of a `for` loops:

- **safety** – parallel execution has to preserve all dependences

- **profitability** – benefits of parallel execution have to compensate for the overhead penalty
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**

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

**control dependence**

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

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

Theorem

Any reordering transformation that preserves every dependence (i.e., visits first the source, and then the sink of the dependence) in a program preserves the meaning of that program.

\[ \square \]

Note: Dependence starts with the notion of a sequential execution, i.e., starts with a sequential program.
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:

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

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

\textbf{vectorization}

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

\textbf{parallelization}

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

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

Things to do:

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