**Class Information**

**INFORMATION and REMINDERS**

- HW7 deadline: Tuesday, April 18, 11:59pm.
- Project 2 Use `plt-r5rs` - racket version v6.5
- **FINAL EXAM**
  1. **Monday, May 8, 4:00-7:00pm**, College Ave. Campus
  2. cumulative, more than 60% new material; closed book, closed notes
  3. **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.

We will talk about about

- Dependence analysis
- Vectorization
- Parallelization
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**

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

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.

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:

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

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

\[\text{vectorization}\]

\[
A(1:100:1,1:100:1) = A(1:100:1,1:100:1) + 1
\]

\[
A(1:99,1:100) = A(2:100,1:100) + 1
\]

\[\text{parallelization}\]

\[
\begin{array}{l}
\text{doall } I = 1, 100 \\
\text{doall } J = 1, 100 \\
A(I,J) = A(I,J) + 1 \\
\text{enddo}
\end{array}
\]

\[
\begin{array}{l}
\text{do } I = 1, 99 \\
\text{doall } J = 1, 100 \\
A(I,J) = A(I+1,J) + 1 \\
\text{enddo}
\end{array}
\]

\[\text{implicit barrier sync.}\]

\[
\begin{array}{l}
\text{implicit barrier sync.}
\end{array}
\]

\[
\begin{array}{l}
\text{implicit barrier sync.}
\end{array}
\]
Next Lecture

- Dependence analysis
- Automatic vectorization / parallelization