REMINDERS

• Project 1 grades have been posted. We are looking into “0” points cases.

• Midterm: average 190/250; Project 1: 68/100

• Project 2 has been posted. Due: November 30 (Monday after Thanksgiving). Submission instruction will be posted.

• Homework 7 due December 1.

• Last class: Tuesday, December 8 (December 10 last day of classes)

• Final exam: Thursday, December 17, 4:00pm - 7:00pm, BRR 1095; accumulative, closed books, closed notes. CONFLICTS?
Programming with Concurrency

Why do we care about concurrency?

- Today, concurrency is nearly everywhere (peta-flops supercomputers to high-end smart phones).
- Necessary to keep “Moore’s Law” alive due to power/heat dissipation limits.
- Some form of parallel programming will be required, i.e., automatic tools have not been able to hide all aspects of concurrency.

⇒

Need to understand the basics of parallel programming
Programming with Concurrency

Two ways of thinking about concurrency?

**data-centric view**: partition the data that can be worked on in parallel (data-level parallelism);
⇒ your work is determined by the data that you are assigned to work on.

**task-centric view**: partition the work that can be done concurrently (task-level parallelism);
⇒ your data is determined by the work that you have to do

What tasks have “to travel” to what data (data-centric) or what data has “to travel” to what tasks (task-centric) are symmetric problems.
Programming with Concurrency

Task-level parallelism can be performed at different levels:

1. **Instruction-level** parallelism (ILP) – typically exploited by hardware or compiler

2. **Loop-level parallelism** – single loop iterations are considered individual tasks

3. **Procedure-level** parallelism – different procedures may be executed concurrently

4. **Process-level** parallelism – different programs may be executed concurrently

Will concentrate on loop-level parallelism
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 briefly about

- Dependence analysis
- Vectorization
- Parallelization
OpenMP

- Allows expression of parallelism at different levels: task and loop level
- Parallelization is done through **pragmas**.
- Look at the OpenMP documentation on our class web site.
Parallel Threads Execution Model

Distributed Memory
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**

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

How to preserve the meaning of these programs?

Execute the statements in an order that preserves the original load/store order.
Dependence — 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!
Next Lecture

Things to do:

• Dependence analysis
• OpenMP
• More on automatic vectorization / parallelization