Announcements

- **Project 2 deadline**: today at 11:59pm
- **Roadmap**: Parallel programming; type systems, logic programming (Prolog)
- **Final exam period**: May 4 to May 10. Exam on Wednesday, May 4, noon to 3:00pm (class time slot).
Why Do We Care About Concurrency?

CMOS (Complementary Metal-Oxide-Semiconductor) technology:

\[
\text{Power}_{CPU} = \text{capacitance} \times \text{voltage}^2 \times \text{frequency}
\]

Moore’s Law for **single** CPU speed:
Increase in CPU speed by 2× every 18-24 months.

How to keep Moore’s law going?

- **Increase frequency**: Power goes up.
- **Reduce feature size**: Dennard’s Scaling Rule:
  Power is proportional to the area of the transistor.

  Problem: Dennard’s Scaling ignores “leakage current” and “threshold voltage” ⇒ as transistors get smaller, power density increases

- **Better designs**: That works, but it does not give you exponential improvements (e.g.: deeper pipeline, better branch prediction, bigger caches, etc.)

  ⇒ We hit the “**Power Wall**”.
Power Wall Answer: Parallel Architectures!

Instead of a single CPU, use many CPUs to keep Moore’s law going.

Power Density

Once you are beyond 100W/cm$^2$ or so, aircooling alone will not be sufficient.

Frequency Scaling Wall
Programming with Concurrency

Concurrency is here to stay!

- Today, concurrency is nearly everywhere (peta-flops supercomputers to high-end smart phones).
- Some form of parallel programming will be required, i.e., automatic tools have not been able to hide all aspects of concurrency.
- Several different parallel programming models / architecture designs: shared memory, distributed memory, heterogeneous computing with accelerators (GPUs, FPGAs)

⇒ Need to understand

1. the basics of parallel programming
2. the specifics for each architecture.
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.
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

- Dependence analysis
- Vectorization
- Parallelization
- Heterogenous parallel architectures
Shared Memory Programming with OpenMP

- MIMD architecture (multiple instructions, multiple data)
- Allows expression of parallelism at different levels: task and loop level. Parallelization through pragmas.
- Basic fork/join thread execution model with barrier synchronization between parallel regions.
Shared Memory Programming with OpenMP

OpenMP program example:

```c
#pragma omp parallel for private(i, hash)
    for (j = 0; j < num_hf; j++) {
        for (i = 0; i < wl_size; i++) {
            hash = hf[j] (get_word(wl, i));
            hash %= bv_size;
            bv[hash] = 1;
        }
    }
```

This specifies:

- outermost (j-loop) is parallel
- each thread will get its own copy of variables i and hash, eliminating loop carried anti and output dependences.
Distributed-Memory Programming with MPI

Distributed Memory

MPI (message passing interface)

- MIMD architecture (multiple instructions, multiple data). SPMD programming model (single program, multiple data).
- No global shared memory. Communication through explicit send/receive operations. Receives are blocking, sends may or may not be blocking.
- MPI defines an abstract processor topology (e.g.: 3-dim grid) to allow “virtual” addressing of processors (e.g.: North, South, West, East in a 2-dim grid)
Heterogenous Computing and CUDA

CUDA: **Compute Unified Device Architecture**

- Host and device (GPU) programs. Program consists of parallel kernels that are executed in sequence.
- GPUs have been designed for speed for graphic applications (e.g.: real-time gaming) as a graphical co-processor. Bare metal design approach.
- Explicit movement of objects between host and device (GPU) memory.
- GPU optimized for streaming computation with limited temporal locality
- GPU implements SIMT (single instruction multiple threads) model.
Heterogenous Computing and CUDA

CPU systems vs. GPU systems comparison

From a “raw capability” point of view, GPUs win big time.
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**

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

**control dependence**

\[
\begin{align*}
S_1 & \quad \text{if} (t \neq 0.0) \text{ then} \\
S_2 & \quad a = a/t \\
& \quad \text{endif}
\end{align*}
\]

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:

\[
\begin{align*}
&\text{do } I = 1, 100 \\
&\hspace{1cm} \text{do } J = 1, 100 \\
&\hspace{2cm} A(I,J) = A(I,J) + 1 \\
&\hspace{1cm} \text{enddo} \\
&\text{enddo}
\end{align*}
\]

\[
\begin{align*}
&\text{do } I = 1, 99 \\
&\hspace{1cm} \text{do } J = 1, 100 \\
&\hspace{2cm} A(I,J) = A(I+1,J) + 1 \\
&\hspace{1cm} \text{enddo} \\
&\text{enddo}
\end{align*}
\]

**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**

\[
\begin{align*}
&\text{doall } I = 1, 100 \\
&\hspace{1cm} \text{doall } J = 1, 100 \\
&\hspace{2cm} A(I,J) = A(I,J) + 1 \\
&\hspace{1cm} \text{enddo} \\
&\hspace{1cm} \text{implicit barrier sync.} \\
&\text{enddo}
\end{align*}
\]

\[
\begin{align*}
&\text{doall } I = 1, 99 \\
&\hspace{1cm} \text{doall } J = 1, 100 \\
&\hspace{2cm} A(I,J) = A(I+1,J) + 1 \\
&\hspace{1cm} \text{enddo} \\
&\hspace{1cm} \text{implicit barrier sync.} \\
&\text{enddo}
\end{align*}
\]

198:515, Spring 2016