Class Information

- Homework 7 due today. Project 2 due this Sunday.
- Midterm grades released.
  Average: 172.5; Median: 178.5; 25%: 200.5; 75%: 145.75.
- Changed policy to calculate exam grade.
  Take the better of the two – midterm grade or final grade.
  Detailed distribution for midterm is listed below:

```
Histogram of Midterm [250 pts total]
```

![Histogram of Midterm Grades](image)
Why do we care about concurrency?

► Today, concurrency is nearly everywhere (peta-flops supercomputers to 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
Classic von Neumann model of computing has single thread control, however, parallel programs have more than one.

How to decompose a task into multiple independent sub-tasks?

**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
Flynn’s Classical Taxonomy

- One of the most widely used classifications, in use since 1966.
- Single Instruction, Single Data (SISD) – Non-parallel
- Single Instruction, Multiple Data (SIMD)
- Multiple Instruction, Single Data (MISD)
- Multiple Instruction, Multiple Data (MIMD)
Dependence and Parallelization

Dependence analysis is fundamental to task decomposition.

- Divide a task into indivisible sequential units of computation
- A task decomposition can be modeled as a directed graph such that if $A \rightarrow B$ the result of task $A$ is required for the processing of task $B$.

Example:

\begin{align*}
C &= 4; \\
D &= 2; \\
B &= 5 + D; \\
A &= B + C;
\end{align*}
Loop-level Parallelism

We will use loop analysis as example to describe dependence analysis and parallelization.

Typically, scientific codes:

- Have loops that contain most of the computation in the program. Tens of lines of code might comprise up to 90% computation.
- Use arrays as their main data structures.

As a result, many optimizing transformations concentrate on loop level optimizations. Most loop level optimizations are source–to–source, i.e., reshape loops at the source level.
Loop Level Parallelism

Dependence relation: 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.

<table>
<thead>
<tr>
<th>data dependence</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S_1 ) ( \pi = 3.14 )</td>
</tr>
<tr>
<td>( S_2 ) ( r = 5.0 )</td>
</tr>
<tr>
<td>( S_3 ) ( \text{area} = \pi \times r^{**2} )</td>
</tr>
</tbody>
</table>

How to preserve the meaning of these programs during parallelization?

**Semantics is preserved if dependence is preserved**
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: We start with the notion of a sequential execution, i.e., starts with a sequential program.
Bernstein’s Condition: — 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. One of them is a write.
3. 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?

Assume we only have scalar and subscripted variables (no pointers and no control dependence) for data dependence analysis.

Exercise: which loop is parallelizable?

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

Zheng Zhang 12 eddy.zhengzhang@cs.rutgers.edu
Dependence — Where do we need it?

Assume we only have scalar and subscripted variables (no pointers and no control dependence) for data dependence analysis.

Exercise: which loop is parallelizable?

```plaintext
do I = 1, 100
  do J = 1, 100
    A(I,J) = A(I,J) + 1
  enddo
enddo
```

```plaintext
do I = 1, 99
  do J = 1, 100
    A(I,J) = A(I+1,J) + 1
  enddo
enddo
```

`parallelization`

```plaintext
doall I = 1, 100
  doall J = 1, 100
    A(I,J) = A(I,J) + 1
  enddo
  implicit barrier sync.
enddo
```

```plaintext
doall I = 1, 99
  doall J = 1, 100
    A(I,J) = A(I+1,J) + 1
  enddo
  implicit barrier sync.
enddo
```

Zheng Zhang  eddy.zhengzhang@cs.rutgers.edu
Dependence Analysis

Question

Do two variable references never/maybe/always access the same memory location and one of them is a write?

Benefits

▶ improves alias analysis
▶ enables loop transformations
▶ enables parallelization (especially automatic parallelization)

Motivation

▶ classic optimizations
▶ instruction scheduling
▶ data locality (register/cache reuse)
▶ vectorization, parallelization
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*.

do I = 1, 100
    A(I) = ...
    ...= A(I-1)
enddo

\[ \begin{array}{cccc}
1 & 2 & 3 & 4 \\
& I & & \\
\end{array} \]

\[ \begin{array}{cccc}
1 & 2 & 3 & 4 \\
& I & & \\
\end{array} \]

\[ \begin{array}{cccc}
1 & 2 & 3 & 4 \\
& I & & \\
\end{array} \]

\[ \begin{array}{cccc}
1 & 2 & 3 & 4 \\
& I & & \\
\end{array} \]

\[ \begin{array}{cccc}
1 & 2 & 3 & 4 \\
& I & & \\
\end{array} \]

\[ \begin{array}{cccc}
1 & 2 & 3 & 4 \\
& I & & \\
\end{array} \]

\[ \begin{array}{cccc}
1 & 2 & 3 & 4 \\
& I & & \\
\end{array} \]

\[ \begin{array}{cccc}
1 & 2 & 3 & 4 \\
& I & & \\
\end{array} \]

\[ \begin{array}{cccc}
1 & 2 & 3 & 4 \\
& I & & \\
\end{array} \]

\[ \begin{array}{cccc}
1 & 2 & 3 & 4 \\
& I & & \\
\end{array} \]

\[ \begin{array}{cccc}
1 & 2 & 3 & 4 \\
& I & & \\
\end{array} \]

\[ \begin{array}{cccc}
1 & 2 & 3 & 4 \\
& I & & \\
\end{array} \]

\[ \begin{array}{cccc}
1 & 2 & 3 & 4 \\
& I & & \\
\end{array} \]

\[ \begin{array}{cccc}
1 & 2 & 3 & 4 \\
& I & & \\
\end{array} \]

\[ \begin{array}{cccc}
1 & 2 & 3 & 4 \\
& I & & \\
\end{array} \]
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 \textit{dependence} between statement $S_1$ and $S_2$, denoted $S_1 \_resume S_2$, indicates that $S_1$, the \textit{source}, must be executed before $S_2$, the \textit{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.

Does $\exists \alpha \leq \beta$, s.t.

\[
f_k(\alpha) = g_k(\beta) \quad \forall k, 1 \leq k \leq m?
\]
Next time:

- More on dependence testing and parallelization
- Reading: ALSU Chapter 11.1 – 11.3