Announcements

• **Project 3**: Will be posted by tomorrow.

• Exam on Wednesday, May 4, noon to 3:00pm (class time slot). Location to be determined.

• Homework 6 will be posted by tomorrow.
Project: OpenMP and CUDA

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

In the project, you will be given a sequential code that you will need to parallelize using OpenMP and CUDA. You will report experimental results for different problem sizes, data types, and processors/cores used.
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
Vectorization vs. Parallelization

**vectorization** — Find parallelism in innermost loops; fine–grain parallelism

**parallelization** — Find parallelism in outermost loops; coarse–grain parallelism

- Parallelization is considered more complex than vectorization, since finding coarse–grain parallelism requires more analysis (e.g., interprocedural analysis).

- Automatic vectorizers have been very successful
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*
Dependence Testing

Given

\[
\begin{align*}
\text{do } & \ i_1 = L_1, U_1 \\
& \ldots \\
\text{do } & \ i_n = L_n, U_n \\
S_1 & \quad A(f_1(i_1, \ldots, i_n), \ldots, f_m(i_1, \ldots, i_n)) = \ldots \\
S_2 & \quad \ldots = A(g_1(i_1, \ldots, i_n), \ldots, g_m(i_1, \ldots, i_n))
\end{align*}
\]

A dependence between statement \( S_1 \) and \( S_2 \), denoted \( S_1 \delta S_2 \), indicates that \( S_1 \), the source, must be executed before \( S_2 \), the 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.

\[
\text{Does } \exists \ \alpha \leq \beta, \ \text{s.t.} \\
\quad f_k(\alpha) = g_k(\beta) \quad \forall k, \ 1 \leq k \leq m ?
\]
### Iteration Space

\[
\begin{align*}
\text{do } & I = 1, 5 \\
\text{do } & J = I, 6 \\
& \ldots \\
& \text{enddo} \\
& \text{enddo}
\end{align*}
\]

\[1 \leq I \leq 5\]
\[I \leq J \leq 6\]

- lexicographical (sequential) order for the above iteration space is

\[
(1,1), (1,2), \ldots, (1,6) \\
(2,2), (2,3), \ldots (2,6) \\
\ldots \\
(5,5), (5,6)
\]

- given \( I = (i_1, \ldots i_n) \) and \( I' = (i'_1, \ldots i'_n), \)

\[I < I' \text{ iff} \]
\[ (i_1, i_2, \ldots i_k) = (i'_1, i'_2, \ldots i'_k) \ \& \ i_{k+1} < i'_{k+1} \]
Distance & Direction Vectors

\begin{align*}
d & \text{do } I = 1, N \\
& \quad \text{do } J = 1, N \\
&S_1 \ A(I,J) = A(I,J-1) \\
& \quad \text{enddo} \\
& \quad \text{enddo} \\
&S_2 \ A(I,J) = A(I-1,J-1) \\
&S_3 \ B(I,J) = B(I-1,J+1) \\
& \quad \text{enddo} \\
& \quad \text{enddo}
\end{align*}

\begin{itemize}
\item \textbf{Distance Vector} = number of iterations between accesses to the same location
\item \textbf{Direction Vector} = direction in iteration space (=, <, >)
\end{itemize}

\begin{align*}
&S_1 \delta S_1 \\
&S_2 \delta S_2 \\
&S_3 \delta S_3
\end{align*}
Which Loops are Parallel?

\[
\begin{align*}
\text{do } & I = 1, N \\
\text{do } & J = 1, N \\
S_1 & \quad A(I,J) = A(I,J-1) \\
\text{do } & I = 1, N \\
\text{do } & J = 1, N \\
S_2 & \quad A(I,J) = A(I-1,J-1) \\
\text{do } & I = 1, N \\
\text{do } & J = 1, N \\
S_3 & \quad B(I,J) = B(I-1,J+1)
\end{align*}
\]

- A dependence \( D = (d_1, \ldots, d_k) \) is carried at level \( i \), if \( d_i \) is the first nonzero element of the distance/direction vector.

- A loop \( l_i \) is parallel, if \( \forall \) a dependence \( D_j \) carried at level \( i \)

| \( \forall D_j \) | \text{distance vector} \( d_1, \ldots, d_{i-1} > 0 \) | \text{direction vector} \( d_1, \ldots, d_{i-1} = "<" \) |
|武装 | OR | \( d_1, \ldots, d_i = 0 \) | \( d_1, \ldots, d_i = "=" \) |
Approaches to Dependence Testing

• can we solve this problem exactly?

• what is conservative in this framework?

• restrict the problem to consider index and bound expressions that are linear functions

\[ \Rightarrow \text{solving general system of linear equations in integers is NP-hard} \]

Solution Methods

• inexact methods
  ○ Greatest Common Divisor (GCD)
  ○ Banerjee’s inequalities

• cascade of exact, efficient tests
  (fall back on inexact methods if needed)
  ○ Rice (see posted PLDI’91 paper)
  ○ Stanford

• exact general tests (integer programming)
Dependence Testing

SIV - Single Induction Variable Test

1. Single loop nest with constant lower (LB) and upper (UB) bounds, and step 1

\[
\text{for } i = \text{LB}, \text{UB}, 1 \\
\ldots \\
\text{endfor}
\]

The loop bounds define the iteration space for loop induction variable \(i\).

2. Two array references with array subscript (index) expressions of the form (true dependence)

\[
\text{for } i = \text{LB}, \text{UB}, 1 \\
R1: \quad X(a*i + c1) = \ldots \quad \backslash\backslash \text{write} \\
R2: \quad \ldots X(a*i + c2) \ldots \quad \backslash\backslash \text{read} \\
\text{endfor}
\]

where \(a\), \(c1\), and \(c2\) are integer constants, \(R1\) and \(R2\) are references to the same array, \(i\) is the loop induction variable, and \(a \neq 0\).

Question:

Is there a true dependence between \(R1\) and \(R2\)?
Dependence Testing

There is a dependence between R1 and R2 iff

$$\exists i, i' : i \leq i' \text{ and } (a \ast i + c_1) = (a \ast i' + c_2)$$

where $i$ and $i'$ are two iterations in the iteration space of the loop. This means that in both iterations, the same element of array $X$ would be accessed.

So let’s just solve the equation:

$$(a \ast i + c_1) = (a \ast i' + c_2) \iff$$

$$\frac{c_1 - c_2}{a} = i' - i = \Delta d$$

There is a dependence with distance $\Delta d$ iff

1. $\Delta d$ is an integer value and

2. $UB - LB \geq \Delta d \geq 0$
Dependence Testing Examples

1. for $i = LB, UB, 1$
   
   \text{R1: } X(i) = \ldots \quad \backslash \text{ write}
   
   \text{R2: } \ldots X(i - 2) \ldots \quad \backslash \text{ read}
   
   \text{endfor}

   $a=1, c_1=0, c_2=-2 \Rightarrow \Delta d = 2$ \textit{(dependence)}

2. for $i = LB, UB, 1$
   
   \text{R1: } X(2*i) = \ldots \quad \backslash \text{ write}
   
   \text{R2: } \ldots X(2*i - 1) \ldots \quad \backslash \text{ read}
   
   \text{endfor}

   $a=2, c_1=0, c_2=-1 \Rightarrow \Delta d = \frac{1}{2}$ \textit{(no dependence)}

Assume R1 executes before R2.

\underline{Classification of dependences:}:

- R1 is write, R2 is read $\Rightarrow$ \textbf{true} dependence
- R1 is read, R2 is write $\Rightarrow$ \textbf{anti} dependence
- R1 is write, R2 is write $\Rightarrow$ \textbf{output} dependence
Dependence Testing

ZIV - Zero Induction Variable Test

Two array references with array subscript (index) expressions of the form of a constant:

for \( i = \text{LB}, \text{UB}, 1 \)

\begin{align*}
R1: & \quad X(c1) = \ldots \quad \text{\\ write} \\
R2: & \quad \ldots X(c2) \ldots \quad \text{\\ read} \\
\end{align*}

endfor

where \( c_1 \), and \( c_2 \) are integer constants, and R1 and R2 are references to the same array.

There is a dependence between R1 and R2 \textit{iff}

\[ c_1 = c_2 = c. \]

What is the dependence distance \( \Delta d \)?

Since every iteration \( i \) writes \( X(c) \), and every iteration \( i' \) reads \( X(c) \), there is no fixed distance \( \Delta d \). In fact, both references have true, anti, and output dependences:

\[
\Delta d \in \{0, \ldots UB - LB\} \text{ for true} \\
\Delta d \in \{1, \ldots UB - LB\} \text{ for anti and output}
\]
Sample code:

```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.
Sample code:

```c
#define CHUNK_SIZE 2
int chunk = CHUNK_SIZE
#pragma omp parallel for \
    schedule (dynamic, chunk) \
    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, with CHUNK_SIZE iterations scheduled as a group; default chunk size=1
- three basic scheduling strategies:
  - static, dynamic, or guided

There are many more options of specifying how to execute for loops in parallel (see the online OpenMP tutorial)
Project and CUDA

Please see other set of slides: CUDA programming
A Simple Vectorizing Compiler

How to vectorize the following loops?

for (i=2; i<100; i++) {
    S1:  a[i] = b[i+1] + 1;
    S2:  b[i] = a[i] + 5;
}

for (i=2; i<100; i++) {
    S1:  a[i] = b[i-1] + a[i-1] + 3;
    S2:  b[i] = a[i+1] + 5;
}

Simple vectorizer assumptions:

1. singly-nested loops
2. constant upper and lower bounds, step is always 1
3. body is sequence of assignment statements to array
   variables
4. simple array index expressions of induction variable
   (i +/- c or c); can use ZIV or SIV test
5. no function calls
A Simple Vectorizing Source-to-Source Compiler

SKETCH OF BASIC ALGORITHM

Here is a basic vectorization algorithm based on a statement-level dependence graph:

1. Construct statement-level dependence graph considering true, anti, and output dependences; in the final dependence graph, the type of the dependence is not important any more

2. Detect strongly connected components (SCC) over the dependence graph; represent SCC as summary nodes; walk resulting graph in topological order; For each visited node do

    (a) if SCC has more than one statement in it, distribute loop with statements of SCC as its body, and keep the code sequential

    (b) if SCC is a single statement and has no dependence cycle, distribute loop around it and generate vector code; otherwise, mark distributed loop sequential.
Next Lecture

Things to do:

• Project review
• First order predicate calculus
• Logic programming in Prolog