CS 516 Compilers and Programming Languages II
Parallel Computing-2
Goal of parallelization / concurrent execution

Starting point: A sequential application

Goal: Identify “independent pieces of work” that can be executed concurrently without changing the semantics of the application, i.e., do not change application input/output behavior (safety of parallelization). Performance impact \( \rightarrow \) reduction in the length of the critical path

Outcome: A (partially) parallelized application where parts of an application are identified as executable in parallel. Examples: An application may contain parallel regions, parallel loops, fork-join, VLIW, …

Key Challenge: How to decide whether parallel executions of parts of the application are safe? \( \rightarrow \) notion of data and control dependence
Dependence relation: Describes all program regions (e.g.: statements) 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 and control dependencies**:

**Example:**

**data dependence**

(statement level)

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

- \(S_3\) cannot be executed before \(S_1\) or \(S_2\)

**control dependence**

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

- \(S_2\) cannot be executed before \(S_1\)
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:

(1) Dependence starts with the notion of a sequential execution, i.e., starts with a sequential program.

(2) Control dependencies can be converted to data dependencies. From now on, we only look at data dependencies,
We will concentrate on compilation issues for compiling scientific codes on shared-memory parallel/vector architectures. 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
- Automatic vectorization
- Automatic parallelization
- Heterogenous parallel architectures

```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;
    }
}
```
Definition
There is a data dependence from statement $S_1$ to statement $S_2$ ($S_2$ depends on $S_1$) if:
1. Both statements access the same memory location and at least one of them stores/writes into it, and
2. There is a feasible 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 (RAW hazard)
$S_1$ writes a memory location that $S_2$ later reads

anti dependence (WAR hazard)
$S_1$ reads a memory location that $S_2$ later writes

output dependence (WAW hazard)
$S_1$ writes a memory location that $S_2$ later writes

input dependence
$S_1$ reads a memory location that $S_2$ later reads.

Note: Input dependences do not restrict statement (load/store) order!
We restrict our discussion to data dependence for scalar and subscripted variables (no pointers and no control dependence).

**Sequential source code**

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

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

**vectorization**

```
I:100:1, J:100:1 = I:100:1, J:100:1 + 1
A(I:100:1, J:100:1) = A(I:100:1, J:100:1) + 1
A(I:99, J:100) = A(I:99, J:100) + 1
```

**parallelization**

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

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

Lecture 11  cs516, spring 20
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
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
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; together with loop-independent dependencies they can limit possible transformations.
Homework #2 extension until Tuesday, March 3

Next class

- Iteration space
- Distance and direction vectors
- Valid transformations
- Automatic vectorization