COSC4201
Loop Level Parallelism

Prof. Mokhtar Aboelaze

Based on Slides by

Prof. L. Bhuyan (UCR)

Prof. M. Shaaban (RIT)

Loop Level Parallelism LLP

° **Loop-Level Parallelism (LLP) analysis focuses on whether data accesses in later iterations of a loop are data dependent on data values produced in earlier iterations and possibly making loop iterations independent.**

° **e.g. in** **for (i=1; i<=1000; i++)**

x[i] = x[i] + s;

the computation in each iteration is independent of the previous iterations and the loop is thus parallel. The use of X[i] twice is within a single iteration.

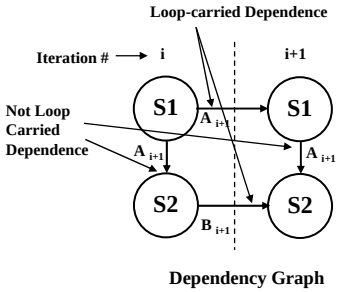
⇒ Thus loop iterations are parallel (or independent from each other).

Loop Level Parallelism LLP

- Loop-carried Dependence: **A data dependence between different loop iterations (data produced in earlier iteration used in a later one).**
- **LLP analysis is important in software optimizations such as loop unrolling since it usually requires loop iterations to be independent.**
- **LLP analysis is normally done at the source code level or close to it since assembly language and target machine code generation introduces loop-carried name dependence in the registers used for addressing and incrementing.**
- **Instruction level parallelism (ILP) analysis, on the other hand, is usually done when instructions are generated by the compiler**

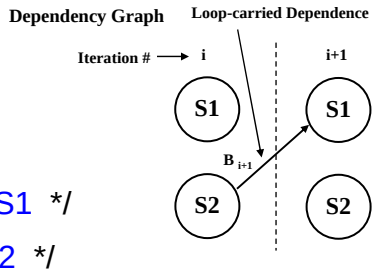
Loop Level Parallelism LLP

```
for (i=1; i<=100; i=i+1) {
        A[i+1] = A[i] + C[i];  /* S1 */
        B[i+1] = B[i] + A[i+1];} /* S2 */
}
```



Dependency Graph

- S2 uses the value A[i+1], computed by S1 in the same iteration. This data dependence is within the same iteration (not a loop-carried dependence).
  - ⇒ does not prevent loop iteration parallelism.
- S1 uses a value computed by S1 in an earlier iteration, since iteration i computes A[i+1] read in iteration i+1 (loop-carried dependence, prevents parallelism). The same applies for S2 for B[i] and B[i+1]
  - ⇒ These two dependencies are loop-carried spanning more than one iteration preventing loop parallelism.




```
for (i=1; i<=100; i=i+1) {
      A[i] = A[i] + B[i];        /* S1 */
     B[i+1] = C[i] + D[i];       /* S2 */
      }
```

- **S1 uses the value B[i] computed by S2 in the previous iteration (loop-carried dependence)**
- **This dependence is not circular:**
  - S1 depends on S2 but S2 does not depend on S1.

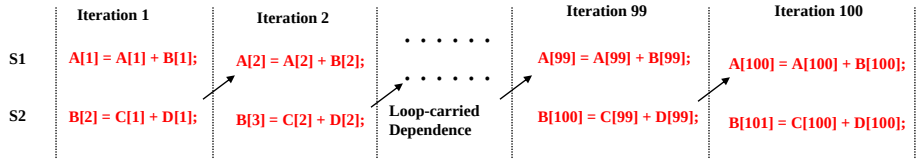
LLP Analysis Example 2

```
A[1] = A[1] + B[1];
  for (i=1; i<=99; i=i+1)  {
        B[i+1] = C[i] + D[i];
        A[i+1] = A[i+1] + B[i+1];
   }
   B[101] = C[100] + D[100];
```

LLP Analysis Example 2

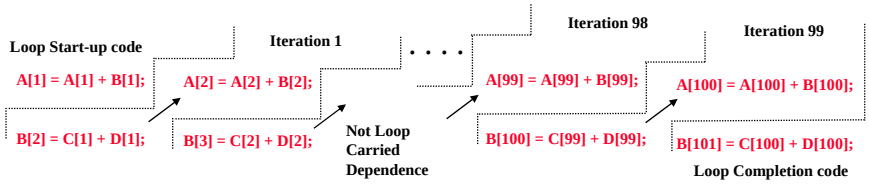
Original Loop:

```
for (i=1; i<=100; i=i+1) {
    A[i] = A[i] + B[i];      /* S1 */
    B[i+1] = C[i] + D[i];    /* S2 */
}
```



Modified Parallel Loop:

```
A[1] = A[1] + B[1];
for (i=1; i<=99; i=i+1) {
    B[i+1] = C[i] + D[i];
    A[i+1] = A[i+1] + B[i+1];
}
B[101] = C[100] + D[100];
```



LLP

```
for(i=2;i<=100;i++) {
    y[i]=y[i-1]+y[i]
}
```

```
for(i=2;i<=100;i++) {
    y[i]=y[i-5]+y[i]
}
```

Finding Dependences

- Finding dependences in the program is very important for renaming and executing instructions in parallel.
- Arrays and pointers makes finding dependences very difficult.
- Assume array indices are *affine*, which means on the form `a x i+b` where `a` and `b` are constant.
- GCD test can be used to detect dependences.

GCD Test

- Assume we stored an array with index value of $a \times i + b$ and loaded an array with an index value of $c \times i + d$
- Are they pointing to the same location?
- Assume the loop limit is *m,n*
- Are there

$$j,k \quad m \le j,k \le n \text{ such that } a \times j + b = c \times k + d$$

GCD Test

- A simple and sufficient test for absence can be found.
- If a loop dependence exists, then

$$GCD(c,a) \text{ must divides } (d-b)$$

- If that test fails, there is no guarantee there is dependence (loop bound)

GCD Test

```
for(i=1; i<=100; i=i+1) {
        x[2*i+3] = x[2*i] * 5.0;
}
```

a = 2 b = 3 c = 2 d = 0

GCD(a, c) = 2

d - b = -3

2 does not divide -3 ⇒ No dependence is not possible.

5,7,9,11,13,15,17,19,21,23,….

4,6,8,10,12,14,16,18,20,22,…..

Dependence Analysis -- Difficulties

- Dependence analysis is a very important tool for exploiting LLP, it can not be used in these situations
- Objects are referenced using pointers
- Array indexing using another array A[b[I]]
- Dependence may exist for some values of input, but in reality the input never takes these values.
- When we want to more than the possibility of dependence (which write causes it?)
- Dependence analysis across procedure boundaries

Dependence Analysis -- Difficulties

- Sometimes, *points-to* analysis might help.
- We might be able to answer *simpler* questions, or get some hints.
- Do 2 pointers point to the same list?
- Type information
- Information derived when the object was allocated
- Pointer assignments