

Loop-Level Parallelism

Loops are the **primary source of parallelism** in programs. Compiler technology can discover and exploit this parallelism through careful analysis of data dependencies between loop iterations.

Loop-Carried Dependence

Occurs when data accesses in later iterations depend on values produced in earlier iterations. This is the key factor that determines whether a loop can be parallelized.

Induction Variables

Variables like loop counters that follow a predictable pattern can often be recognized and eliminated, enabling parallelization even when apparent dependencies exist.

Intra-Loop Dependence

Dependencies within a single iteration don't prevent parallelism across iterations, as long as statements within each iteration maintain their order.

Finding and manipulating loop-level parallelism is critical for exploiting both Data-Level Parallelism (DLP) and Thread-Level Parallelism (TLP), as well as more aggressive static Instruction-Level Parallelism (ILP) approaches.

Analyzing Loop Dependencies

```
for (i=0; i<100; i=i+1) {  
    A[i+1] = A[i] + C[i];    /* S1 */  
    B[i+1] = B[i] + A[i+1]; /* S2 */  
}
```

- This loop contains two different dependencies:
 - S1 uses A[i] computed in the previous iteration - a loop-carried dependence that forces sequential execution
 - S2 uses A[i+1] computed by S1 in the same iteration - not loop-carried, allowing parallel execution if statements remain ordered
- The compiler must identify these dependencies to determine how much parallelism can be exploited.

Transforming Loops for Parallelism

Not all loop-carried dependencies prevent parallelism. Consider this example:

```
for (i=0; i<100; i=i+1) {  
    A[i] = A[i] + B[i];    /* S1 */  
    B[i+1] = C[i] + D[i]; /* S2 */  
}
```

S1 depends on the value of B[i] assigned by S2 in the previous iteration. However, this dependence isn't circular - S2 doesn't depend on S1.

We can transform the loop to expose parallelism:

```
A[0] = A[0] + B[0];  
for (i=0; i<99; i=i+1) {  
    B[i+1] = C[i] + D[i];  
    A[i+1] = A[i+1] + B[i+1];  
}  
B[100] = C[99] + D[99];
```

Now the dependence is no longer loop-carried, allowing iterations to overlap if statements remain ordered within each iteration.

Finding Dependencies in Array Accesses



Affine Array Indices

Most dependence analysis algorithms work on array indices in the form $a \times i + b$, where a and b are constants and i is the loop index variable.



Mathematical Tests

Determining dependence requires checking if two affine functions can have the same value for different indices within loop bounds.



GCD Test

A simple test: if a loop-carried dependence exists between array accesses $a \times i + b$ and $c \times i + d$, then $\text{GCD}(c, a)$ must divide $(d - b)$.

While determining whether a dependence exists is **generally NP-complete**, many **common cases** can be analyzed precisely at low cost. Modern compilers use a hierarchy of exact tests increasing in generality and cost to efficiently analyze dependencies.

Dependence analysis is critical for exploiting parallelism, but it has limitations - it works best with affine array indices and struggles with pointer-based accesses and cross-procedure analysis.

Example

- Use the GCD test to determine whether dependences exist in the following loop:

```
for (i=0; i<100; i=i+1) {  
    X[2*i+3] = X[2*i] * 5.0;  
}
```

- Answer
 - Given the values $a = 2$, $b = 3$, $c = 2$, and $d = 0$, then $\text{GCD}(a,c) = 2$, and $d - b = -3$.
 - Since 2 does not divide -3 , no dependence is possible.

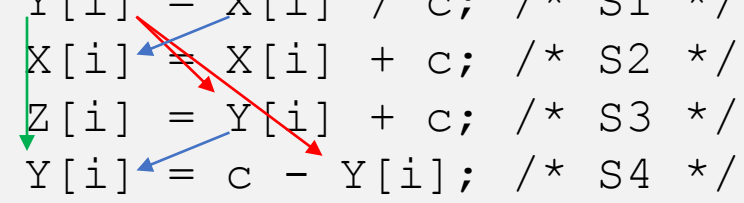
Example 2

- The following loop has multiple types of dependences. Find all the true dependences, output dependences, and antidependences, and eliminate the output dependences and antidependences by renaming.

```
for (i=0; i<100; i=i+1) {  
    Y[i] = X[i] / c; /* S1 */  
    X[i] = X[i] + c; /* S2 */  
    Z[i] = Y[i] + c; /* S3 */  
    Y[i] = c - Y[i]; /* S4 */  
}
```

Example 2

```
for (i=0; i<100; i=i+1) {  
    Y[i] = X[i] / c; /* S1 */  
    X[i] = X[i] + c; /* S2 */  
    Z[i] = Y[i] + c; /* S3 */  
    Y[i] = c - Y[i]; /* S4 */  
}
```



■ Answer

- True dependences

- From S1 to S3 and from S1 to S4 because of Y[i]
- These are not loop carried, so they do not prevent the loop from being considered parallel
- These dependences will force S3 and S4 to wait for S1 to complete

- Antidependences

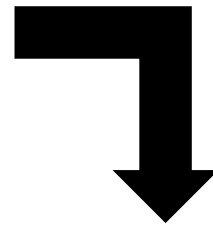
- From S1 to S2, based on X[i], and from S3 to S4 for Y[i]

- Output dependence

- From S1 to S4, based on Y[i]

Example 2

```
for (i=0; i<100; i=i+1) {  
    Y[i] = X[i] / c; /* S1 */  
    X[i] = X[i] + c; /* S2 */  
    Z[i] = Y[i] + c; /* S3 */  
    Y[i] = c - Y[i]; /* S4 */  
}
```



```
for (i=0; i<100; i=i+1) {  
    T[i] = X[i] / c; /* Y renamed to T to remove output dependence */  
    X1[i] = X[i] + c; /* X renamed to X1 to remove antidependence */  
    Z[i] = T[i] + c; /* Y renamed to T to remove antidependence */  
    Y[i] = c - T[i];  
}
```


Handling Recurrences and Reductions

```
for (i=9999; i>=0; i=i-1)
    sum = sum + x[i] * y[i];
```



Make the loop parallel, but need a reduction.
Reductions are common in linear algebra.

```
for (i=9999; i>=0; i=i-1)
    sum[i] = x[i] * y[i];
for (i=9999; i>=0; i=i-1)
    finalsum = finalsum + sum[i];
```



This loop, which sums up 1000 elements on each of the ten processors, is completely parallel.
A simple scalar loop can then complete the summation of the last ten sums.

```
for (i=999; i>=0; i=i-1)
    finalsum[p] = finalsum[p] + sum[i+1000*p];
```