Detecting Loop-Level Parallelism

Prof. Naga Kandasamy
ECE Department, Drexel University

The analysis of loop-level parallelism focuses on determining whether data accesses in later iterations are dependent on data values produced in earlier iterations—the so-called *loop-carried dependence*. Consider the following simple example:

```
for (i = 0; i < 1000; i ++)
    x[i] = x[i] + s;
```

In the above loop, the uses of `x[i]` are dependent, but this dependence is within a single iteration of the loop and is not loop carried. Consider a slightly more complex example:

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

Here `A`, `B`, and `C` are distinct, non-overlapping arrays. There are two different dependencies:

1. Statement `S1` uses a value computed by `S1` in an earlier iteration since iteration i computes `A[i + 1]` which is read in iteration $i+1$. The same is true for statement `S2` for `B[i]` and `B[i + 1]`. This dependence is loop carried, and forces successive iterations of this loop to execute in series.

2. Statement `S2` uses the value `A[i + 1]` computed by `S1` in the same iteration. This dependence is within an iteration and is not loop carried. Thus, if this were the only dependence, multiple iterations of the loop could execute in parallel, as long as each pair of statements in an iteration were kept in order.

Another example is one that calculates the first n Fibonacci series.

```
fib[0] = fib[1] 1;
for (i = 2; i < n; i++)
    fib[i] = fib[i - 1] + fib[i - 2];
```

We can try parallelizing the above loop with a `parallel for` directive.

```
fib[0] = fib[1] = 1;
#pragma omp parallel for num_threads(thread_count)\
                         private(i) shared(fib)
for (i = 2; i < n; i++)
    fib[i] = fib[i - 1] + fib[i - 2];
```

The compiler will create an executable without complaint. However, the results will be incorrect due to the loop carried dependencies. We note two important points:

- OpenMP compilers don't check for dependencies among iterations in a loop that's been parallelized with a `parallel for` directive.
- A loop in which the results of one or more iterations depend on other iterations cannot, in general, be correctly parallelized by OpenMP.

It is also possible to have a loop-carried dependence that does not prevent parallelism, as the following example shows.

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

Statement `S1` uses the value of `B[i]` assigned in the previous iteration by statement `S2`, so there is a loop-carried dependence between `S2` and `S1`.

Despite this loop-carried dependence, this loop can be made parallel. Unlike the earlier loop, this dependence is not circular; neither statement depends on itself, and although `S1` depends on `S2`, `S2` does not depend on `S1`. A loop is parallel if it can be written without a cycle in the dependences, since the absence of a cycle means that the dependences give a partial ordering on the statements. Although there are no circular dependences in the above loop, it must be transformed to conform to the partial ordering and expose the parallelism.

1. There is no dependence from `S1` to `S2`. If there were, then there would be a cycle in the dependences and the loop would not be parallel. Since this other dependence is absent, interchanging the two statements will not affect the execution of `S2`.
2. On the first iteration of the loop, statement `S2` depends on the value of `B[0]` computed prior to initiating the loop.

Building on the above two observations, we can rewrite the original loop as follows:

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

The dependence between the two statements is no longer loop carried, so that iterations of the loop may be overlapped, provided the statements in each iteration are kept in order.

Finding the dependences in a program is important both to determine which loops might contain parallelism and to eliminate name dependences. Most dependence analysis algorithms work on the assumption that array indices are *affine*. A one-dimensional array index is affine if it can be written in the form $a \times i + b$, where a and b are constants and i is the loop index variable. The index of a multidimensional array is affine if the index in each dimension is affine.

```
for (i = 0; i < n; i++) {
    A[a*i + b] = ...;          /* Writing to array location */
    ... = A[c*i + d];          /* Reading from array location */
}
```

Determining whether there is a dependence between two references to the same array in a loop is thus equivalent to determining whether two affine functions can have the same value for different indices between the bounds of the loop. In the above code snippet, suppose we have stored to an array element with index value $a \times i + b$ and loaded from the same array with index value $c \times i + d$, where $0 \leq i < n$ is the loop index variable. A dependence exists if two conditions hold:

1. There are two iteration indices,j and k, that are both within the limits of the `for` loop. That is $0 \leq j < n$ and $0 \leq k < n$, and $j \neq k$.
2. The loop stores into an array element indexed by $a \times j + b$ and later fetches from that same array element when it is indexed by $c \times k + d$. That is, $a \times j + b = c \times k + d$.

In general, we cannot determine whether dependence exists at compile time. For example, the values of a, b, c, and d may not be known (they could be values in other arrays), making it impossible to tell if a dependence exists. In other cases, the dependence testing may be very expensive but decidable at compile time; for example, the accesses may depend on the iteration indices of multiple nested loops. Many programs, however, contain primarily simple indices where a, b, c, and d are all constants. For these cases, we can devise reasonable compile time tests for dependence.

As an example, a simple and sufficient test for the absence of a dependence is the *greatest common divisor* (*GCD*) test. It is based on the observation that if a loop-carried dependence exists, then $\mathrm{GCD}(c, a)$ must divide $(d - b)$. Recall that an integer, x, divides another integer, y, if we get an integer quotient when we do the division y/x and there is no remainder.

For example, we can use the above GCD test to determine whether dependences exist in the following loop:

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

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

The following is another example:

```
for(i = 0;i < 100;i++) {
    a[2*i] = b[i];         /* Statement S1 */
    c[i] = a[4*i + 1];     /* Statement S2 */
}
```

We have $\text{GCD}(2, 4) = 2$ and the dividend is 1. As 2 can not divide 1 while leaving a reminder of 0, there is no dependency between statements S1 and S2. The loop can be parallelized.

The GCD test is sufficient to guarantee that no dependence exists; however, there are cases where the GCD test succeeds but no dependence exists. This can arise, for example, because the GCD test does not consider the loop bounds. In general, determining whether a dependence actually exists is NP-complete.

Let's discuss one final example regarding parallelizing `for` loops.

$$\pi = 4\left[1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots\right] = 4\sum_{k=0}^{\infty}\frac{(-1)^k}{2k+1}$$

The serial code implementing the above formula is as follows.

```
double plus_minus = 1.0;
double sum = 0.0;

for (k = 0; k < n; k++) {
    sum += plus_minus/(2 * k + 1);
    plus_minus = -plus_minus;
}
approx_pi = 4 * sum;
```

Simply parallelizing the loop as follows will not help.

```
1  int plus_minus = 1;
2  double sum = 0.0;
3
4  #pragma omp parallel for num_threads(thread_count)\
5                           private(k) reduction(+:sum)
6  for (k = 0; k < n; k++) {
7      sum += plus_minus/(2 * k + 1);
8      plus_minus = -plus_minus;
9  }
10 approx_pi = 4 * sum;
```

This is because of the loop carried dependence of the *plus_minus* variable in line 8. If loop iterations i and $i + 1$ are distributed to two different threads, the resulting calculation will be incorrect — since both threads will use the value of 1 for *plus_minus*. Examining the alternating series of pluses and minuses in the infinite series, however, we note that one thread must use the value of +1 for

plus_minus whereas the other must use -1. The value to use depends on the value of the loop index k. Each thread must independently determine the values of *plus_minus* based on the loop iterations allocated to it. The corrected code is as follows.

```
int plus_minus;
double sum = 0.0;

#pragma omp parallel for num_threads(thread_count)\
                         private(k, plus_minus) reduction(+:sum)
for (k = 0; k < n; k++) {
    if (k % 2 == 0) /* Better to use k & 1 == 0 for speed */
        plus_minus = 1;
    else
        plus_minus = -1;
    sum += plus_minus/(2 * k + 1);
}
approx_pi = 4 * sum;
```