Detecting and Enhancing Loop-Level Parallelism

- Loop-level parallelism is normally analyzed at the source level or close to it, while most analysis of ILP is done once instructions have been generated by the compiler. Loop-level analysis involves determining what dependences exist among the operands in a loop across the iterations of that loop. For now, we will consider only data dependences. The analysis of loop-level parallelism focuses on determining whether data accesses in later iterations are dependent on data values produced in earlier iterations; such dependence is called a loop-carried dependence.

```
for(i=999; i>=0; i=i-1)
        x[i]= x[i]+ s;
```

- In this loop, the two uses of x [i] are dependent, but this dependence is within a single iteration and is not loop carried. There is a loop-carried dependence between successive uses of i in different iterations, but this dependence involves an induction variable that can be easily recognized and eliminated.

Example : Consider a loop like this one:

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

}
```

Assume that A, B, and C are distinct, nonoverlapping arrays. What are the data dependences among the statements Si and S2 in the loop?

Answer There are two different dependences:

1. Si uses a value computed by Si in an earlier iteration, since iteration i com putes A[i+1], which is read in iteration i+1. The same is true of S2 for B[i] and B[i+1].

2. S2 uses the value A[i+1] computed by S1 in the same iteration.

- These two dependences are different and have different effects. To see how they differ, let's assume that only one of these dependences exists at a time. Because the dependence of statement S1 is on an earlier iteration

of S1, this dependence is loop carried. This dependence forces successive iterations of this loop to execute in series.

- The second dependence (S2 depending on S1) 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.

Example: Consider a loop like this one:

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

What are the dependences between Si and S2? Is this loop parallel? If not, show how to make it parallel.

Answer :

- Statement S1 uses the value 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. Two observations are critical to this transformation:
  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.

- These two observations allow us to replace the loop above with the following code sequence:

```
A [0]= A [0]+ B [0] ;

   for (i=0; i<99; i=i+1)

{

   B[i+l]= 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 Dependences:

- Nearly all dependence analysis algorithms work on the assumption that array indices are affine. In simplest terms, a one-dimensional array index is affine if it can be written in the form a x 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. Sparse array accesses, which typically have the form x [y [ii]] , are one of the major examples of non-affine accesses.

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 GCD(a,c) = 2, and d - b =-3

Since 2 does not divide3, no dependence is possible.

Eliminating Dependent Computations:

- As mentioned above, one of the most important forms of dependent computations is a recurrence. A dot product is a perfect example of a recurrence:
  for(i=9999; i>=0; i=i-1)
  sum= sum+ x[i]* y[i];

- This loop is not parallel because it has a loop-carried dependence on the variable sum. We can, however, transform it to a set of loops, one of which is completely parallel and the other that can be partly parallel. The first loop will execute the completely parallel portion of this loop. It looks like:
  for (i=9999; i>=0; i=i-1)
  SUM[i]= x[i]* y[i];
- Notice that sum has been expanded from a scalar into a vector quantity (a transformation called scalar expansion) and that this transformation makes this new loop completely parallel. When we are done, however, we need to do the reduce step, which sums up the elements of the vector. It looks like:
  for (i=9999; i>=0; i=i-1)
  finalsum= finalsum+ sum[i];

- Although this loop is not parallel, it has a very specific structure called a reduction. Reductions are common in linear algebra and, as we shall see in Chapter 6, they are also a key part of the primary parallelism primitive MapReduce used in warehouse-scale computers. In general, any function can be used as a reduction operator, and common cases include operators such as max and min.

- Reductions are sometimes handled by special hardware in a vector and SIMD architecture that allows the reduce step to be done much faster than

it could be done in scalar mode. These work by implementing a technique similar to what can be done in a multiprocessor environment. While the general transformation works with any number of processors, suppose for simplicity we have 10 processors. In the first step of reducing the sum, each processor executes the following (with p as the processor number ranging from 0 to 9):

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

- 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. Similar approaches are used in vector and SIMD processors.