Loop Transformation

Dr. Jitendra V. Tembhurne

Contents

- for_all and for_across code conversion
- Synchronisation Primitives
- Transformation Techniques
- Unimodular Transformations

for_all:

If loop independence exists in a loop then it can be converted into a “for_all”

```
for( i = 0;i < N1;i + +)
  for(j = 0;j < N2;j + +)
       a[i][j] = a[i][j − 2];

for_all( i = 0;i < N1;i + +)
  for(j = 0;j < N2;j + +)
       a[i][j] = a[i][j − 2];
```

for_across:

If loop carried dependence exists in a loop then it can be converted into a “for_across” with synchronization.

```
for(i = 2;i <= N;i + +)
  {
    S1 : a[i] = a[i − 1] ∗ b[i] + c[i];
    S2 : c[i] = a[i] + c[i];
    S3 : d[i] = a[i] ∗ 2;
    S4 : e[i] = d[i] + c[i];
  }
```

- **Critical Sections**

```
for_across(i = 2;i <= N;i + +)
{
  begin critical section
  S1 : a[i] = a[i − 1] ∗ b[i] + c[i];
  end critical section
  S2 : c[i] = a[i] + c[i];
  S3 : d[i] = a[i] ∗ 2;
  S4 : e[i] = d[i] + c[i];
}
```

- **Random Synchronisation**

```
for(i = 0;i < N; i + +)
{
   S1 : a[i] = b[i] + 2;
   S2 : d[i] = a[i]/d[i];
   S3 : c[i] = 2 ∗ d[i] + a[i − 1];
}
```

```
for_across(i = 0;i < N;i + +)
{
   S1 : a[i] = b[i] + 2;
   post(ASYNC, i);
   S2 : d[i] = a[i]/d[i];
   wait(ASYNC, i − 1);
   S3 : c[i] = 2 ∗ d[i] + a[i − 1];
}
```

- Pipelining

  In pipelining the ‘for’ loop is divided into segments of code such that source and sink of a dependence lie within the same segment.

```
for(i = 0;i < N;i + +)
{
   S1 : a[i] = b[i] + c[i − 1];
   S2 : c[i] = a[i − 1] + c[i];
   S3 : d[i] = c[i] ∗ 2;
   S4 : e[i] = d[i] + c[i − 1];
   S5 : f[i] = e[i] + f[i];
   S6 : g[i] = f[i] ∗ 2 + d[i];
}
```

```
for(i = 0;i < N;i + +)
{
   Seg1 : a[i] = b[i] + c[i − 1];
   Seg1 : c[i] = a[i − 1] + c[i];

   Seg2 : d[i] = c[i] ∗ 2;
   Seg2 : e[i] = d[i] + c[i − 1];

   Seg3 : f[i] = e[i] + f[i];
   Seg3 : g[i] = f[i] ∗ 2 + d[i];
}
```

- **Barrier synchronisation**

  This synchronisation strategy handles only forward references. In this strategy a loop is divided into segments but all iterations must complete execution of a segment before execution of the next segment is started.

```
for_across(i = 0;i < N; i + +)
{
   S1 : a[i] = b[i] + 2;
   barrier
   S2 : c[i] = a[i − 1] + c[i];
   S3 : d[i] = a[i] + 2;
   barrier
   S4 : e[i] = d[i] + c[i − 1];
   S5 : f[i] = e[i] + f[i];
}
```

Transformation Techniques

- **Loop Coalescing**

  This method transforms several nested concurrent loops into a single concurrent loop. Thus scheduling of the loops is easier.

```
for_all(i = 1;i <= N1;i + +)
  for_all(j = 1;j <= N2;j + +)
      for(k = 1;k <= N3;k + +)
            a[i][j] = a[i][j] + b[i][k] * c[k][j];
```

```
for_all(ij = 1;ij <= N1N2;ij + +)
        for(k = 1;k <= N3;k + +)
                a[ij/N2][ij/N1] = a[ij/N2][ij/N1]
                        + b[ij/N2][k] * c[k][ij/N1];
```

- **Loop Distribution**

  Nested loops can be vectorised by distributing the outermost loop and vectorising each of the resulting loops or loop nests. Thus this loop transformation method is very much suitable for vector machines.

```
for(i = 1;i <= N;i + +)
{
   b[i][1] = 0;
   for(j = 1;i <= M;j + +)
        a[i] = a[i] + b[i][j] * c[i][j];
        g[i] = f[i] + a[i];
}
```

```
b[1 : N][1] = 0;

for(i = 1;i <= N;i + +)
        a[i] = a[i] + b[i][1 : M] * c[i][1 : M];

g[1 : N] = f[1 : N] + a[1 : N];
```

```
for(i = 0;i < N;i + +)
{
        S1 : a[i + 1] = b[i − 1] + c[i];
        S2 : b[i] = a[i] ∗ k;
        S3 : c[i] = b[i] − 1;
}
```

The dependencies for this loop are $S_1 \delta^f S_2$, $S_2 \delta^f S_1$, $S_1 \delta^a S_3$ and $S_2 \delta^f S_3$.

```
for(i = 0;i < N;i + +)
{
        S1 : a[i + 1] = b[i − 1] + c[i];
        S2 : b[i] = a[i] ∗ k;
}
c[0 . . . N − 1] = b[0 . . . N − 1] − 1;
```

- **Node splitting**

  The data dependence cycle in a loop which prevents loop vectorisation can some-times be broken by node splitting.

```
for(i = 1;i <= N; i + +)
{
        S1 : d[i] = a[i − 1] + a[i + 1];
        S2 : a[i] = b[i] + c[i];
}
```

$S_2 \delta^f S_1$, $S_1 \delta^a S_2$

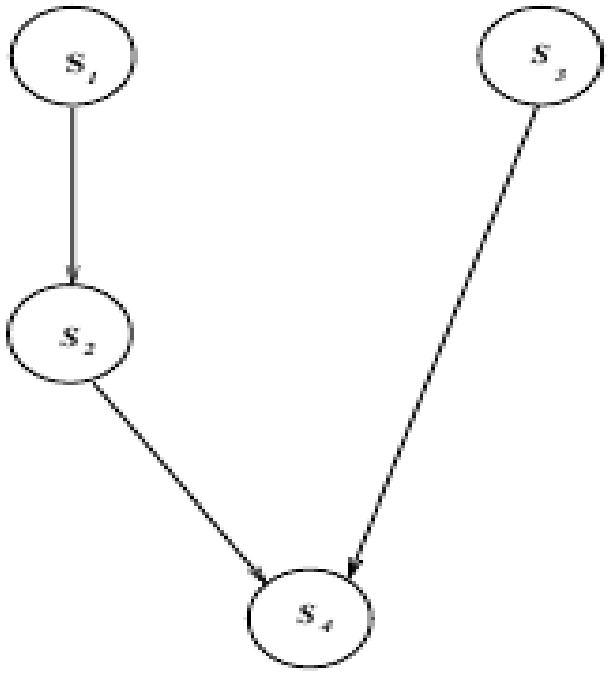
```
for(i = 1;i <= N;i + +)
{
        S1a : x[i] = a[i + 1];
        S2 : a[i] = b[i] + c[i];
        S1b : d[i] = a[i − 1] + x[i];
}
```

- **Statement substitution**

  If the dependence graph for a loop has multi-statement cycles, then statement reordering cannot be used to vectorise the loop. In this case statement substitution can be used to reduce the dependence cycle to a single statement cycle.

```
S1 : a = 1;
S2 : b = a + 2;
S3 : c = 2;
S4 : d = b + c;
```

- The dependence graph for the above example is shown below



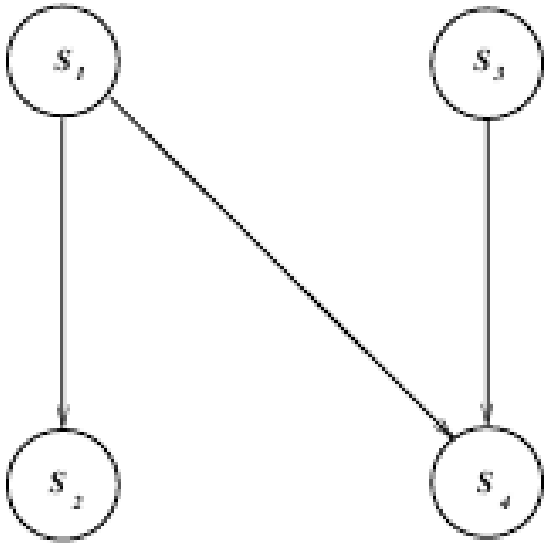
S1 : a = 1;

S2 : b = a + 2;

S3 : c = 2;

S4 : d = (a + 2) + c;

- The modified dependence graph is



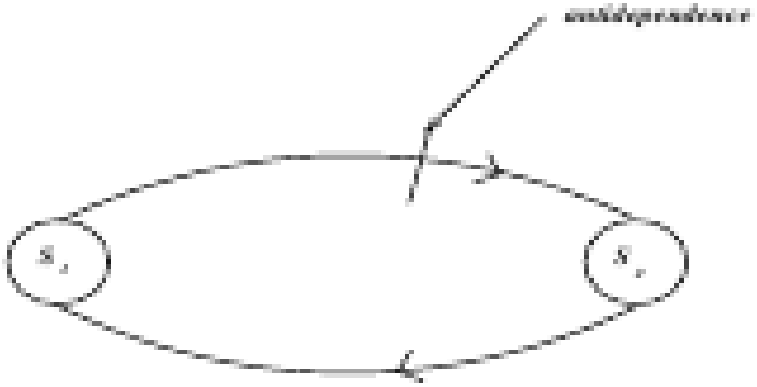
```
for(i = 0;i < N; i + +)
{
        S1 : a[i] = c[i] + b[i];
        S2 : c[i] = e[i];
        S3 : b[i + 1] = a[i] + 2;
}
```

The above loop has a cycle $S_1 \delta S_3 \delta S_1$

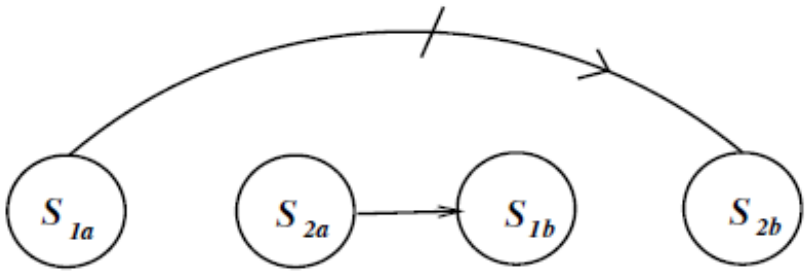
```
for(i = 0;i < N;i + +)
{
        S1 : a[i] = c[i] + b[i];
        S3 : b[i + 1] = (c[i] + b[i]) + 2;
        S2 : c[i] = e[i];
}
```

- **Index set splitting**
  When a cycle of data flow dependence and data anti dependence exists, index set splitting can be applied.

```
for(i = 1;i <= 100;i+)
{
        S1 : a[i] = b[101 − i] + c;
        S2 : b[i] = e[i];
}
```



```
for(i = 1;i <= 50;i + +)
{
      S1a : a[i] = b[101 − i] + c;
      S2a : b[i] = e[i];
}
for(i = 51;i <= 100;i + +)
{
      S1b : a[i] = b[101 − i] + c;
      S2b : b[i] = e[i];
}
```


S_{1a}
S_{2a}
S_{1b}
S_{2b}

- **Loop fusion**
  Loop fusion merges two loops into a single loop

```
for_all(i = 0;i < N;i + +)
        d[i] = e[i] + f[i] + x[i];
for_all(j = 0;j < N;j + +)
        e[j] = d[j] + f[j];

for_all(i = 0; i < N; i + +)
        d[i] = e[i] + f[i] + x[i];
        e[i] = d[i] + f[i];
```

- **Loop interchange**

  Vectorisation is performed in the inner loop rather than in the outer loop. Some-times it is possible to perform loop interchange so as to enable vectorisation.

```
for(i = 1;i <= N;i + +)
   for(j = 1;j <= N;j + +)
        S1 : a[i][j] = a[i][j − 1] + a[i][j + 1];
```

$S_1 \delta^f S_1$ with distance vector [0 1] and $S_1 \delta^a S_1$ with distance [0 1].

```
for(i = 1;i <= N;i + +)
  for(j = 1;j <= N;j + +)
      a[i][j] = a[i − 1][j] + a[i − 2][j];
```

Interchange the loop for “for_all”

Unimodular Transformations

- **Unimodular matrix**

A square integral matrix U is unimodular if $|U| = \pm 1$.

- **Loop interchange**

If d =[d1,d2] is the dependence vector in loop interchange then after loop interchanging we get the transformed dependence vector

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} d_1 \\ d_2 \end{bmatrix} = \begin{bmatrix} d_2 \\ d_1 \end{bmatrix}$$

- **Loop reversal**

  Reversal of the i^{th} loop in a loop nest is represented by the identity matrix with i^{th} diagonal element set to -1.

$$U = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

```
for(i = 1;i <= N;i + +)
  for(j = 1;j <= N;j + +)
       a[i][j] = a[i − 2][j + 1];
```

After loop revarsal:

```
for(i = 1;i <= N;i + +)
  for(j = −1;j <= −N;j + +)
       a[i][−j] = a[i − 2][−j + 1];
```

- **Loop skewing**
  Skewing loop ij by an integer factor f with respect to loop ii maps the iteration (i1, i2,i3, . . . i_{i-1} ,i_i, i_{j+1}, . . . , i_{j-1}, i_j, i_{j+1}, . . .,i_n) to (i_1, i_2,i_3, . . .,i_{i-1}, i_i, i_{i+1}, . . ., ij−1, i_j+ fi_i,i_{j+1},. . . ,i_n).

```
for(i1 = 0;i1 <= 6;i1 + +)
   for(i2 = 0;i2 <= 8;i2 + +)
         a[i1][i2] = a[i1][i2 − 1] + a[i1 − 1][i2] + a[i1 − 1][i2 + 1] ;
```

We get D , U={1,0;1,1} and new D'

```
for(i1 = 0;i1 <= 6;i1 + +)
for(j2 = j1;j2 <= j1 + 8;j2 + +)
   a[j1][j2 − j1] = a[j1][j2 − j1 − 1] + a[j1 − 1][j2 − j1−1]
                    + a[j1 − 1][j2 − j1];
```

- **Loop permutation**

  A permutation σ on a loop nest transforms iteration (i_1, i_2, i_3,. . ., i_n) to ($i_{\sigma 1}$, $i_{\sigma 2}$, $i_{\sigma 3}$, . . ., $i_{\sigma n}$). The transformation can be expressed in a matrix form as a unimodular transformation matrix U derived from the identity matrix I with rows permuted by σ .

We get D, U={0,1;1,0}, D'

```
for(i = 1;i <= N; i + +)
  for(j = 1;j <= N; j + +)
       a[i][j] = a[i − 1][j − 1] + a[i][j − 1];
```

```
for(i = 1;i <= N; i + +)
  for(j = 1;j <= N; j + +)
      a[i][j] = a[i − 1][j − 1] + a[i –1][j];
```