Compilers for Parallel Systems LVA 185.A64 SS 2018
Hans Moritsch hans.moritsch@tuwien.ac.at

Loop Interchange

Example 1

```
for i = 1 to 100
   for j = 1 to 100
      S a[i,j+1] = a[i,j]*b[i,j]
   endfor
endfor
```

$S\ \delta_2\ S$

⇒

```
for j = 1 to 100
   for i = 1 to 100
      S a[i,j+1] = a[i,j]*b[i,j]
   endfor
endfor
```

$S\ \delta_1\ S$

⇒

```
for i = 1 to 100
   S a[1:100,j+1] = a[1:100,j]*b[1:100,j]
endfor
```

Example 2

```
for i = 1 to n
   for j = 1 to n
      for k = 1 to n
         S c[i,j] = c[i,j]+a[i,k]*b[k,j]
      endfor
   endfor
endfor
```

$S\ \delta_3\ S$

⇒

```
for k = 1 to n
   for i = 1 to n
      for j = 1 to n
         S c[i,j] = c[i,j]+a[i,k]*b[k,j]
      endfor
   endfor
endfor
```

$S\ \delta_1\ S$

⇒

```
for k = 1 to n
   S c[1:100,1:100] = c[1:100,1:100]+
                    spread(a(1:100,k),2,100)*
                    spread(b(k,1:100),1,100)
      endfor
   endfor
endfor
```

`spread(x,dim,n)` replicates array `x`, in dimension `dim`, `n` times

Example 3

```
for i = 1 to 100
   for j = 1 to 100
      S a[i+1,j] = a[i,j+1]*b[i,j]
   endfor
endfor
```

$S\ \delta_{1,true}^{(<,>)}\ S$

e.g. $S(1,2)\ \delta_{true}\ S(2,1)$, caused by a[2,2]

⇒

```
for j = 1 to 100
   for i = 1 to 100
      S a[i+1,j] = a[i,j+1]*b[i,j]
   endfor
endfor
```

S(2,1) executed before *S(1,2)*

dependence eliminated, loop interchange invalid

Vectorization

* move loops with dependence cycles outwards
* move loops without dependendences inwards
* also: move large loops inwards

```
for i = 1 to 10000
   for i = j to 5
      S a[j,i] = b[j,i]*c[j-1,i+1]
   endfor
endfor
```

Parallelization

* move loops without dependences outwards
* move loops with dependence cycles inwards
* parallelize large loops

Loop interchange is a reordering transformation

* must preserve all dependences

Validity of loop interchange

Perfectly nested loop nest L:
interchange loop at level at level L_c with loop at level L_{c+1}
$\Rightarrow L/c$ with execution order $\ll /c$ and dependence relation δ/c
iteration vector $\mathbf{i}/c = \mathbf{i}$ with components c and $c+1$ swapped

preserve dependence: $S(\mathbf{i})\ \delta\ S'(\mathbf{i}')$ Þ $S(\mathbf{i})\ \delta/c\ S'(\mathbf{i}')$

dependence is destroyed: $S(\mathbf{i}) \ll S'(\mathbf{i}')$ Þ $S'(\mathbf{i}'/c) \ll S(\mathbf{i}/c)$ iff $\theta = (=^{c-1},<,>,*...*)$

Loop interchange at level c is valid iff there is no dependence $S\ \delta\ \theta\ S'$ in L such that $\theta = (=^{c-1},<,>,*...*)$.

Loop independent dependences never prevent (valid) loop interchange

Example 4

```
for i = 1 to 100
   for j = 1 to 100
      S a[i,j+1] = a[i,j]*a[i-1,j+1]
   endfor
endfor
```

$S\ \delta_1^{(=,<)}\ S,\ \ S\ \delta_2^{(<,=)}\ S$
cannot be vectorized

$\Rightarrow$

```
for j = 1 to 100
   for i = 1 to 100
      S a[i,j+1] = a[i,j]*a[i-1,j+1]
   endfor
endfor
```

$S\ \delta/1_1^{(<,=)}\ S,\ \ S\ \delta/1_2^{(=,<)}\ S$
cannot be vectorized

Effect on dependences

* loop carried dependences at levels other than c and $c+1$ are not affected
* loop independent dependences are not affected
* loop carried dependences at level $c+1$ move *outward* to level c
* loop carried dependences at level c
  + *remain* at level c for $dir(\mathbf{i},\mathbf{i}') = (=^{c-1},<,<,*...*)$
  + move *inward* to level $c+1$ for $dir(\mathbf{i},\mathbf{i}') = (=^{c-1},<,=,*...*)$
* all loop carried dependences at level $c+1$ in L/c correspond to loop carried dependences at level c in L

Scalar Expansion

Example 5

```
for i = 1 to 100
   S1 a = b[i]*c[i]
   S2 d[i] = a+1
   S3 e[i] = a*(d[i]-2)
endfor

⇒
float $a[1:100]
for i = 1 to 100
   S1' $a[i] = b[i]*c[i]
   S2' d[i] = $a[i]+1
   S3' e[i] = $a[i]*(d[i]-2)
endfor
```

Example 5

```
a=f[0,0]

for i = 1 to 10
   for j = 1 to 20
      b[i,j] = a*b[i+1,j-1]
      a = f[i,j]
   endfor
endfor

⇒
float $a[1:10,0:20]

a=f[0,0]
$a[1,0] = a

for i = 1 to 10
   for j = 1 to 20
      b[i,j] = $a[i,j-1]*b[i+1,j-1]
      $a[i,j] = f[i,j]
   endfor
   if i<10 $a[i+1,0] = $a[i,20]
endfor
a = $a[10,20]
```

Variable Copying

Example 7

```
for i = 1 to n
   S  a[i] = b[i]*c[i]
   S' d[i] = a[i]+a[i+1]
endfor
```

⇒

```
float $a2[1:n]

$a2[1:n] = a[2:n+1]
for i = 1 to n
   S  a[i] = b[i]*c[i]
   S' d[i] = a[i]+$a2[i]
endfor
```

⇒

```
$a2[1:n] = a[2:n+1]
a[1:n] = b[1:n]*c[1:n]
d[1:n] = a[1:n]+$a2[1:n]
```

Example 8

```
for i = 1 to n
   a[i] = a[i+2]+1
endfor
```

⇒

```
a[1:n] = a[3:n+2]+1
```

Fetch-before-store semantics of vector statements has effect of variable copying ⇒ vectorization in case of a single statement antidependence cycle is possible

Index set splitting

Example 9

```
for i = 1 to 200
   S  b[i] = a[201-i]+c[i]
   S' a[i] = c[i-1]*2
endfor
```

⇒

```
for i = 1 to 100
   S  b[i] = a[201-i]+c[i]
   S' a[i] = c[i-1]*2
endfor

for i = 101 to 200
   S  b[i] = a[201-i]+c[i]
   S' a[i] = c[i-1]*2
endfor
```

⇒

```
b[1:100] = a[200:101:-1] + c[1:100]
a[1:100] = c[0:99]*2
b[101:200] = a[100:1:-1] + c[101:200]
a[101:200] = c[100:199]*2
```

Node Splitting

Example 10

```
L for i = 1 to n
   S  b[i] = a[i]+c[i]*d[i]
   S' a[i+1] = b[i]*(d[i]-c[i])
endfor
```

⇒

```
L' for i = 1 to n
   S1  $t1[i] = c[i]*d[i]
   S2  $t2[i] = d[i]-c[i]
   S3  b[i] = a[i]+$t1[i]
   S4  a[i+1] = b[i]*$t2[i]
endfor
```

⇒

```
$t1[1:n] = c[1:n]*d[1:n]
$t2[1:n] = d[1:n]-c[1:n]
for i = 1 to n
   S3 b[i] = a[i] + $t1[i]
   S4 a[i+1] = b[i] + * $t2[i]
endfor
```

Loop Peeling

Example 11

```
for j = 1 to n
   for i = 1 to n
      S1 f[i,j] = g[i,j]+3*g[i-1,j]
      if j<2 goto S4
      if j<n goto S3
      S2 g[i,j] = f[i,j]
      S3 g[i-1,j] = f_old[i,j]
      S4 f_old[i,j] = f[i,j]
   endfor
endfor
```

⇒

```
for i = 1 to n
   f[i,1] = g[i,1]+3*g[i-1,1]
   f_old[i,1] = f[i,1]
   endfor
endfor

for j = 2 to n-1
   for i = 1 to n
      f[i,j] = g[i,j]+3*g[i-1,j]
      g[i-1,j] = f_old[i,j]
      f_old[i,j] = f[i,j]
   endfor
endfor

for i = 1 to n
   f[i,n] = g[i,n]+3*g[i-1,n]
   g[i,n] = f[i,n]
   g[i-1,n] = f_old[i,n]
   f_old[i,n] = f[i,n]
endfor
```

⇒

```
f[1:n] = g[1:n1] + 3* g[0:n-1,1]
f_old[1:n,1] = f[1:n,1]
f[1:n,2:n-2] = g[1:n,2:n-1]+3*[g[0:n-1,2:n-1]
g[0:n-1,2:n-1] = f_old[1:n,2:n-1]
f_old[1:n,2:n-1] = f[1:n,2:n-1]
for i = 1 to n
   f[i,n] = g[i,n]+3*g[i-1,n]
   g[i,n] = f[i,n]
endfor
g[0:n-1,n] = f_old[1:n,n]
f_old[1:n,n] = f[1:n,n]
```

Loop Unrolling

Example 12

```
for i = 1 to 1000
   a[i] = b[i+2]*c[i-1]
endfor
```

⇒

```
for i = 1 to 999 step 2
   a[i] = b[i+2]*c[i-1]
   a[i+1] = b[i+3]*c[i]
endfor
```

can improve data locality

Example 12a

```
for j = 1 to n
   for i = 1 to 64
      S c[i] = c[i] + a[i,j] * b[j]
   endfor
endfor

c[1:64] = c[1:64] + a[1:64,j] * b[j]
```

vector code:

```
for j = 1 to n
   v0 = c[1:64]
   v1 = a[1:64,j]
   r0 = b[j]
   v1 = v1 * r0
   v0 = v0 + v1
   c[1:64] = v0
endfor
```

unrolling:

for j = 1 to n-1 step 2

```
   for i = 1 to 64
      S′ c[i] = c[i] + a[i,j] * b[j] + a[i,j+1] * b[i,j+1]
   endfor
endfor

for i = 1 to n
   v0 = c[1:64]
   v1 = a[1:64,j]
   r0 = b[j]
   v1 = v1 * r0
   v0 = v0 + v1
   v2 = a[1:64,j+1]
   r1 = b[j+1]
   v2 = v2 * r1
   v0 = v0 + v2
   c[1:64] = v0
```

```
endfor
```

`v0` used twice!

Loop Rerolling

Example 13

```
q = 0.0
for k = 1 to 996 step 5
   q = q+z(k)*x[k]+z(k+1)*x[k+1]+z(k+2)*x[k+2]+z(k+3)*x[k+3]
endfor
```

⇒

```
q = 0.0
for k = 1 to 1000
   q = q+z(k)*x[k]
endfor
```

⇒

```
q = dotproduct(z,x)
```

Idiom Recognition

reduction function “reduces” array to a scalar

automatic detection of reductions

- sum(a)
- product(a)
- dotproduct(a,b)
- max/minval(a)
- max/minloc(a)minloc(a)
- ...

If-conversion

forward branch

```
for i = ...
   ...
   S  goto S'
   ...
   S' ...

   ...
endfor
```

backward branch

```
for i = ...
   ...
   S  ...
   ...
   S' if k<0 goto S
   ...
endfor
```

exit branches

```
for i = ...
   S1 ...
   for j = ...
      ...
      S2 if k<eps goto S1
      ...
      S3 if l>ub goto S4
      ...
   endfor
endfor
S4 ...
```

transform conditional statements and branches into statements with *masks:*

Example 14

```
for i = 1 to n
   S1 a[i] = a[i] + b[i]                 true
   S2 if a[i]==0                         true
         then S3 goto S7                 a[i]==0
   S4 if a[i]>c[i]                       !(a[i]==0)
         then S5 a[i] = a[i-2]           !(a[i]==0) && a[i]>c[i]
         else S6 a[i] = a[i]+1           !(a[i]==0) && !(a[i]>c[i])
   S7 d[i]=a[i]*2                        true
endfor
```

introduce a variable for each conditional expression

⇒
```
for i = 1 to n
   S1 a[i] = a[i] + b[i]                    true
   S2'$c2 = (a[i]==0)                       true
   S2 if $c2                                true
         then S3 goto S7                    $c2
   S4'$c4 = (a[i]>c[i])                     !$c2
   S4 if $c4                                !$c2
         then S5 a[i] = a[i-2]              !$c2 && $c4
         else S6 a[i] = a[i]+1              !$c2 && !$c4
   S7 d[i]=a[i]*2                           true
endfor
```

eliminate (original) conditional statements and branches ⇒ sequence of masked statements

⇒
```
for i = 1 to n
   a[i] = a[i] + b[i]
   $c2 = (a[i]==0)
   if !$c2                                  $c4 = (a[i]>c[i])
   if !$c2 && $c4                           a[i] = a[i-2]
   if !$c2 && !$c4                          a[i] = a[i]+1
   d[i]=a[i]*2
endfor
```

scalar expansion of condition variables, vectorization, generation of where-statements

⇒
```
a[1:n] = a[1:n] + b[1:n]
$c2[1:n] = (a[1:n]==0)
where (!$c2[1:n]))                          $c4[1:n] = (a[1:n]>c[1:n])
where (!$c2[1:n]) && $c4[1:n])              a[1:n] = [1:n]-2
where (!$c2[1:n]) && !$c4[1:n])             a[1:n] = [1:n]+1
d[1:n] = a[1:n]*2
```

backward and exit branches: introduce control variable

Example 15

```
for i = 1 to n
   a[i] = b[i] + (c[i-1]+1)
   for j = 1 to m
      d[i,j] = d[i,j-1]/a[i]
      if d[i,j] < 0 then goto S
   endfor
   b[i] = b[i]/c[i+1]
 S a[i] = a[i]*b[i]
endfor
```

⇒

```
for i = 1 to n
   a[i] = b[i] + (c[i-1]+1)
   $in_j_loop = true
   for j = 1 to m
      if $in_j_loop then d[i,j] = d[i,j-1]/a[i]
      if $in_j_loop then $in_j_loop = !(d[i,j] < 0)
   endfor
   if !$in_j_loop then goto S   ... forward branch
   b[i] = b[i]/c[i+1]
 S a[i] = a[i]*b[i]
endfor
```

backward branches are usually loops

Detection of loops in flow graphs

Flow graph *G = (N, E, s)*

(N,E) is a directed graph

s is the initial node: there is a path from *s* to every node of *G*

Program (flow) graph: a node may represent

- single statement of source program,
- single statement of an intermediate representation (e.g. triples, quadrupels),
- ***basic block***: sequence of statements with **one** entry and **one** exit point,

Initial node corresponds to program/procedure statement

```
bb 1
S0 program
S1 read(l)
S2 n = 0
S3 k = 0
S4 m = 1

bb 2
S5 k = k + m
S6 c = k > l
S7 if c then goto S11

bb 3
S8 n = n + 1
S9 m = m + 2
S10 goto S5

bb 4
S11 write(n)
S12 end
```

Dominance Relation ≤

reflexive partial order over nodes of a flow graph

node *n* **dominates** node *n'* $(n \leq n')$ ⇔ each path from s to *n'* contains *n*

n **properly** dominates *n'* $(n < n')$ ⇔ $n \leq n'$ and $n \neq n'$

n **directly** dominates *n'* $(n <_d n')$ ⇔ $n < n'$ and there exists no n'' : $n < n'' < n'$

DOM(*n*) ... set of **dominators** of *n*

$1 \leq 1$, $1 \leq 2$, $1 \leq 3$, $1 \leq 4$, $2 \leq 2$, $2 \leq 3$, $2 \leq 4$, $3 \leq 3$, $4 \leq 4$

$1 <_d 2$, $2 <_d 3$, $2 <_d 4$

$DOM(1) = \{1\}$, $DOM(2) = \{1,2\}$, $DOM(3) = \{1,2,3\}$, $DOM(4) = \{1,2,4\}$

$n < n' \Rightarrow n$ is executed before n'

direct dominator of *n* is unique

direct dominator of *n* needs not to be immediate predecessor of *n*

dominator tree: each node dominates only its descendents

Natural loop

1. single entry point (*header*), dominates all nodes in the loop
2. at least one path back to the header

look for edges (n,d) such that $d < n$ ... *back edges*

Natural loop of a back edge

d and all nodes that can reach *n* without going through *d*

d ... loop header

Loop sectioning, Strip mining

```
for i = 1 to n
   a[i] = b[i] + c[i]
endfor

a[1:n] = b[1:n] + c[1:n]
```

maximum vector length of 64

*n = k*64 + r*

outer *sectioning* loop and inner *strip* loop
always legal
changes dependence distance

⇒

```
for $i1 = 1 to k+1
   for $i2 = 1 to min(64,n-($i1-1)*64)
      i = ($i1-1)*64 + $i2
      a[i] = b[i] + c[i]
   endfor
endfor
```

⇒

```
for $i1 = 1 to k
   a[($i1-1)*64+1 : $i1*64] = b[($i1-1)*64+1 : $i1*64]
endfor
if r>0 then a[$i1*64+1 : k*64+r] = b[$i1*64+1 : k*64+r] ...cleanup
```

alternative formulation:

⇒

```
for $is = 1 to n step 64
   for i = $is to min(n,$is+64-1)
      a[i] = b[i] + c[i]
   endfor
endfor
```

Loop Tiling

strip mining for nested loops

```
for i = 1 to m
   for j = 1 to n
      a[i,j] = a[i,j] + 1
   endfor
endfor
```

⇒

```
for $is = 1 to n step 64
   for $js = 1 to n step 64
      for i = $is to min(n,$is+64-1)
         for j = $is to min(n,$js+64-1)
            a[i,j] = a[i,j] + 1
         endfor
      endfor
   endfor
endfor
```