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

Parallelization

parallel process: execution of one sequential instruction stream
Shared memory systems: no loop carried dependences ⇒ **for-all-loop**

Example 1

```
for i = 1 to 100
   a[i] = b[i]*c[i]+d[i]
   b[i] = c[i]/d[i-1]+a[i]
   if c[i]<0 then c[i] = a[i] * b[i]
endfor
```

no loop carried dependences

⇒

```
for all i = 1 to 100
   a[i] = b[i]*c[i]+d[i]
   b[i] = c[i]/d[i-1]+a[i]
   if c[i]<0 then c[i] = a[i] * b[i]
endfor
```

100 parallel processes, process p_i performs iteration i

Shared memory systems: satisfy loop carried dependences using synchronization ⇒ **for-across-loop**

Example 2

```
for i = 5 to 100
   S  a[i] = b[i]*c[i]+d[i]
   S' b[i] = c[i]/d[i-1]+a[i-4]
endfor
```

$S\ \delta^{(4)}_1\ S$

⇒

```
for across i = 5 to 100
   S  a[i] = b[i]*c[i]+d[i]
      if i<=100-4 then signal(i)
      if i>=5+4   then wait(i-4)
   S' b[i] = c[i]/d[i-1]+a[i-4]
endfor
```

true dependence: defining process signals using process that required data is available in shared memory

4 iterations in parallel

a[5] = b[5]*c[5]+d[5] **signal(5)** b[5] = c[5]/d[4]+a[1]	a[6] = b[6]*c[6]+d[6] signal(6) b[5] = c[6]/d[5]+a[2]	a[7] = b[7]*c[7]+d[7] signal(7) b[7] = c[7]/d[6]+a[3]	a[8] = b[8]*c[8]+d[8] signal(8) b[8] = c[8]/d[7]+a[4]	a[9] = b[9]*c[9]+d[9] signal(9) **wait(5)** b[9] = c[9]/d[8]+a[5]

Distributed memory systems: satisfy true dependence from process *p* to process *p'* by sending value of variable defined in *p* to *p'* (using message passing)

Single Program Multiple Data (SPMD) programming model

Multiple parallel processes execute the same code
may perform different instructions, may access different data at any time

serial section is executed by the first process which reaches the section only, other processes skip it (e.g. initialization, I/O)

parallel section is executed in parallel without synchronization
for-all-loop
iterations are repeatedly assigned to processes - as long as there are iterations left
a process reaching the section after all iterations have been assigned skips the section

replicate section is executed by all processes

barrier statement for synchronization: when reached by a process, it must wait until all other processes have reached it

Example

```
for i = 1 to 3
   S ...
endfor

for j = 1 to 4
   S' ...
endfor
```

no dependences:

⇒

```
for all i = 1 to 3
   S ...
endfor

for all j = 1 to 4
   S' ...
endfor
```

5 processes, e.g.

S(1)	*S(2)*	*S(3)*	*S'(1)*	*S'(2)*
S'(3)		*S'(4)*		

with dependences $S \delta S'$:

⇒

```
for all i = 1 to 3
   S ...
endfor
barrier
for all j = 1 to 4
   S' ...
endfor
```

5 processes:

S(1)	*S(2)*	*S(3)*		

S'(3)		*S'(4)*	*S'(1)*	*S'(2)*

```
for i = 1 to n
   for j = 1 to m
      S ...
   endfor
endfor
```

$S\ \delta_1\ S$, no dependences at level 2:

⇒

```
for i = 1 to 3
   for all j = 1 to 3
      S ...
   endfor
endfor
```

⇔ unrolled: 3 parallel sections:

```
   for all j = 1 to 3
      S ...
   endfor
   for all j = 1 to 3
      S ...
   endfor
   for all j = 1 to 3
      S ...
   endfor
```

5 processes:

S(1,1)	*S(1,2)*	*S(1,3)*	*S(2,1)*	*S(2,2)*
S(2,3)	*S(3,1)*	*S(3,2)*	*S(3,3)*	

```
for i = 1 to 3
   for all j = 1 to 3
      S ...
   endfor
   if i<3 then barrier
endfor
```

5 processes:

S(1,1)	*S(1,2)*	*S(1,3)*		
S(2,1)	*S(2,2)*	*S(2,3)*		
S(2,1)	*S(2,1)*	*S(2,1)*		

dependence between sections ⇒ need barrier synchronization
minimize synchronization by fusing sections

Parallelization: generalize results from vectorization: find large code regions to be executed in parallel (instead of single statements)

recap

c is a *level* of a path in a dependence graph:
* there is at least one edge which has a dependence of level c *and*
* all edges have a maximal level ≥ *c*

when can we execute instances of a statement in a loop in parallel?
consider statement *S* at loop level *n*, and another level *c*, $c \leq n$:

* S is *serial at level c* iff there is a cycle including S, such that c is a level of this cycle
* $c > n$ (n is a more outer level than c): S is *scalar at level c (*if a statement is at a more outer loop level than c, there are no instances of S at level c at all, which could be subject to parallel execution)
* otherwise S is *concurrent at level c* $\Rightarrow$ instances of S in L_c can be executed in parallel

```
L1 for i = 2 to 4
   L2 for j = 1 to 4
      S a[i,j] = ... a[i-1,j]
   endfor
endfor
```

$S\ \delta_1\ S$

S is serial at level 1, concurrent at level 2

instances of S in j-loop can be executed in parallel:

$\Rightarrow$

```
for i = 2 to 4
   a[i,1:4] = a[i-1,1:4]
end do
```

```
a[2,1] = ... a[1,1]
a[2,2] = ... a[1,2]
a[2,3] = ... a[1,3]
a[2,4] = ... a[1,4]
a[3,1] = ... a[2,1]
a[3,2] = ... a[2,2]
a[3,3] = ... a[2,3]
a[3,4] = ... a[2,4]
a[4,1] = ... a[3,1]
a[4,2] = ... a[3,2]
a[4,3] = ... a[3,3]
a[4,4] = ... a[3,4]
```

```
L1 for i = 1 to 4
   L2 for j = 2 to 4
      S a[i,j] = ... a[i,j-1]
   endfor
endfor
```

$S\ \delta_2\ S \Rightarrow$ no vectorization

S is concurrent at level 1, serial at level 2

instances of S in i-loop can be executed in parallel

```
a[1,2] = ... a[1,1]
a[1,3] = ... a[1,2]
a[1,4] = ... a[1,3]
a[2,2] = ... a[2,1]
a[2,3] = ... a[2,2]
a[2,4] = ... a[2,3]
a[3,2] = ... a[3,1]
a[3,3] = ... a[3,2]
a[3,4] = ... a[3,3]
a[4,2] = ... a[4,1]
a[4,3] = ... a[4,2]
a[4,4] = ... a[4,3]
```

this parallelism cannot be exploited by vectorization

region R_c *at level c* : node of the acyclic condensation of the level *c* dependence graph (= SCC)

R_c is *scalar*, iff it contains a statement which is scalar at level c
R_c is *serial* iff it contains a statement which is serial at level c
R_c is *parallel* iff it is not a serial region

scalar ⇒ serial

if R_c is serial, all statements of R_c are serial at level c
* R_c is an SCC with a cycle at level *c*,
  or a statement not involved in a cycle (if R_c is scalar)

if R_c is parallel, all statements of R_c are concurrent at level c
* R_c is an SCC with cycles at levels > c,
  or a statement not involved in a cycle

R_c is either serial or parallel
R_c is parallel if it is not scalar, and has no edge at level c

<u>all iterations of a parallel region can be executed in parallel without synchronization</u>
* region as a whole can be executed in parallel (no need for loop distribution)

```
L1 for i=1 to 3
    L2 for j=1 to 3
        L3 for k=1 to 3
               S1
           L4 for l=1 to 3
                  S2
                    …
                  S3
              endfor
          endfor
      endfor
  endfor
```

	c=1	c=2	c=3	c=4
S_1 n=3	serial	concurrent	serial	scalar
S_2 n=4	serial	concurrent	serial	concurrent
S_3 n=4	serial	concurrent	serial	concurrent

R_1 = {S_1, S_2, S_3 } ... serial
R_2 = {S_1, S_2, S_3 } ... parallel
R_3 = {S_1, S_2, S_3 } ... serial
$R_{4,1}$ = {S_1} ... scalar
$R_{4,2}$ = {S_2} ... parallel
$R_{4,3}$ = {S_3} ... parallel

Loop Fusion

combine two adjacent for-loops at level *c* into one
requires identical lower and upper bounds

Example 1

valid loop fusion

```
L for i = 1 to n
     S a[i] = a[i-3]+c[i]
  endfor

L' for i = 1 to n
     S' d[i] = a[i-1]*d[i-2]
  endfor
```

⇒

```
L" for i = 1 to n
     S  a[i] = a[i-3]+c[i]
     S' d[i] = a[i-1]*d[i-2]
  endfor
```

$S\ \delta_\infty\ S' \Rightarrow S\ \delta_1\ S'$

consider `a[3]`
before: *S(3)* << *S'(4)*
after: *S(3)* << *S'(4)*

Example 2

invalid loop fusion

```
L for i = 1 to n
     S a[i] = a[i-3]+c[i]
  endfor

L' for i = 1 to n
     S' d[i] = a[i+1]*d[i-2]
  endfor
```

⇒

```
L" for i = 1 to n
     S  a[i] = a[i-3]+c[i]
     S' d[i] = a[i+1]*d[i-2]
  endfor
```

consider `a[5]`
before: *S(5)* << *S'(4)*
after: *S(5)* >> *S'(4)* !

a dependence δ_∞ prevents *serial* loop fusion iff $S(\mathbf{i})\ \delta_\infty\ S'(\mathbf{i}')$ and $i_c > i'_c$
* fusion: iteration vectors may be different

loop fusion
* reduces loop overhead
* improves locality

Fusion of for-all-loops

```
L for all i = 1 to n
     stmts
  endfor
```

```
L′ for all i = 1 to n
     stmts′
  endfor
```

⇒

```
L″ for all i = 1 to n
     stmts
     stmts′
  endfor
```

a dependence δ_∞ prevents parallel fusion iff $S(\mathbf{i})\ \delta_\infty\ S'(\mathbf{i}')$ and $i_c \neq i'_c$
* fusion: iteration vectors must be equal

Code generation

generate serial or parallel code for a region at level c
parallel code for **outermost** parallel region of a nest, serial code for the rest of levels
goal: combine regions to minimize synchronisation (number of barriers) ⇒ build ***clusters***:
sequential loops are fused as much as possible
parallel loops are fused as much as possible; if impossible, generate barrier inbetween
minimize number of clusters ⇒ minimize synchronisation

barrier-free edge (vs. *barrier edge*) between two regions
* both regions are serial, or
* both regions are parallel and can be fused

Example 3

```
for i = 1 to n
   for j = 1 to n
      S1 a[i,j] = a[i,j-1]
      S2 b[i,j] = a[i,j-2]+2
   endfor
endfor
```

```
for i = 1 to n
   for j = 1 to n
      for k = 1 to ub
         S3 c[i,j,k] = a[i,j]*(c[i-1,j,k] + c[i,j-2,k] + c[i,j,k-3])/3
      endfor
   endfor
endfor
```

```
for i = 1 to n
   for j = 1 to n
      S4 d[i,j] = d[i-3,j] + a[i,j]*2
   endfor
endfor
```

		c=1	c=2	c=3
S_1 n=2	G_1	parallel	serial	
S_2 n=2	G_2	parallel	parallel	
S_3 n=3	G_3	serial	serial	serial
S_4 n=2	G_4	serial	parallel	

c=1:

(G_1,G_2) is a barrier-free edge (fusable parallel regions)
(G_1,G_3) is a barrier edge
(G_1,G_4) is a barrier edge

clusters: {G_1,G_2} and {G_3,G_4}

cluster consists of segments
segment: connected subgraph of the cluster
all regions in a segment are either *scalar*, or *serial* (not scalar), or *parallel*
if not scalar: all level *c* loops in the segment can be fused

{G_1,G_2}: 1 (parallel) segment
{G_3,G_4}: 2 (serial) segments

parallelization:
⇒

```
for all i = 1 to n
   for j = 1 to n
      S1 a[i,j] = a[i,j-1]
      S2 b[i,j] = a[i,j-2]+2
   endfor
endfor

barrier

for i = 1 to n
   for j = 1 to n
      for k = 1 to ub
         S3 c[i,j,k] = a[i,j]*(c[i-1,j,k] + c[i,j-2,k] + c[i,j,k-3])/3
      endfor
   endfor
endfor

for i = 1 to n
   for all j = 1 to n
      S4 d[i,j] = d[i-3,j] + a[i,j]*2
   endfor
   if i<n then barrier
endfor
```