⑩ Development of Predicated Data-flow Analysis

Motoyasu TAKABATAKE[1, 2]

[1] APC Technology Group [2] Hitachi, Ltd.

Contents

1. Background

- Parallelization of loops is very effective in scientific applications, because most of those loops consume much execution time.
- So, advanced data-flow analysis, which extracts much parallelism from those loops, is very important.

2.1 Problem and Purpose

- Features of conventional data-flow analysis
  - The reference region of each array is calculated.
  - The result is used to find the parallelizability of each loop.
- Problem
  - When an “IF” statement exists in a loop, the data-flow analysis approximates the region referenced in the loop; the compiler may serialize the loop which can be executed in parallel actually.
- Purpose
  - To parallelize the loop including “IF” statements that affect the parallelizability of the loop.

2.2 Problem statement

```
DO I=1,N
   IF (X>5) THEN
      A(I+1) = A(I)...
   ENDIF
   ...= A(I)
ENDDO
```

approximation:
= This is always modified

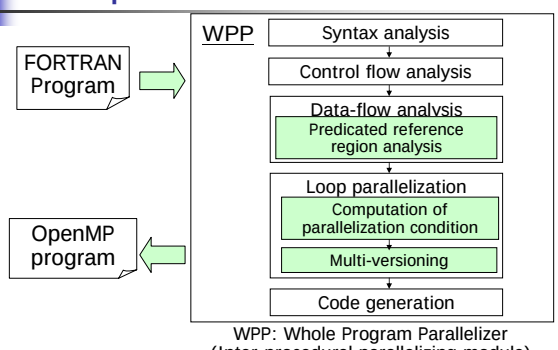
- Sample program:
  if X<=5, the loop is parallelizable, because array A is used only.

- Conventional data-flow analysis
  - Reference regions for Array A in this loop are approximated.
    - Mod：A(2:N+1)
    - Use：A(1:N)
  - This loop has loop-carried dependence.
    So, this loop is serialized.

3. Predicated Data-Flow Analysis

- Features
  - Array reference regions includes “IF” conditions.
  - Parallelization condition is computed using predicated reference regions.
  - Multi-versioned code, which chooses one of parallel or sequential loops at runtime, are generated using those condition.

4. Implementation



WPP: Whole Program Parallelizer
(Inter-procedural parallelizing module)

4.1 Predicated reference region analysis

```
DO I=1,N
   IF (X>5) THEN
      A(I+1) = A(I) ...
   ENDIF
   ...= A(I)
ENDDO
```

Predicate:X>5 is added to reference region:A(I+1)

- Kind of reference regions:

USE	region of arrays that may be used
EUSE	region of arrays that may have upward-exposed use
MOD	region of arrays that may be defined
KILL	region of arrays that must be defined

- Predicated reference region for the array A at I-th iteration:

```
USE:  A[I]
ESUE: A[I]
MOD:  `X>5`,A[I+1]
KILL: `X>5`,A[I+1]
```


4.2 Computation of parallelization condition (definition)

- Conditional expressions for (simple) loop parallelization
  - MOD(1:I-1) ∩ EUSE(I) = ϕ &
    USE(1:I-1) ∩ MOD(I) = ϕ &
    MOD(1:I-1) ∩ MOD(I) = ϕ
- A conditional expression for loop parallelization using privatization
  - MOD(1:I-1) ∩ EUSE(I) = ϕ

$$MOD(1:I-1) := \bigcup_{i \in [1:I-1]} MOD[i]$$

4.2 Computation of parallelization condition (example)

- Conditional expression for MOD(1:I-1) ∩ EUSE(I) = ϕ

I-th iteration:
ESUE(I): A[I]
MOD(I): `X>5`,A[I+1]

1~(I-1)-th iteration:
MOD(1:I-1): `X>5`,A[2:I]

MOD(1:I-1) ∩ EUSE(I)
= `X>5`,A[2:I] ∩ A[I]
= `X>5`,A[I] = ϕ
⇒ `X<=5` → Parallelization condition：X<=5

4.3 Multi-versioning (pattern)

- Generated code chooses one loop from the following three loops at runtime.
  - A parallel loop without private variables,
  - A parallel loop with private variables,
  - A sequential loop.

4.3 Multi-versioning (example)

```
DO I=1,N
   IF (X>5) THEN
      A(I+1) = A(I)...
   ENDIF
   ...= A(I)
ENDDO
```

Input program

Parallelization condition

```
IF (X<=5) THEN
!$OMP PARALLEL DO
   DO I=1,N
      IF (X>5) THEN
         A(I+1) = A(I)...
      ENDIF
      ...= A(I)
   ENDDO
!$OMP END PARALLEL DO
ELSE
   DO I=1,N
      IF (X>5) THEN
         A(I+1) = A(I)...
      ENDIF
      ...= A(I)
   ENDDO
ENDIF
```

Parallel loop

Sequential loop

Output program

5. Evaluation

```
program main
real a(n), b(n)
call init(a,b,n)
read(*,*) c
do i=1,n
   call sub(a,b,i,n,c)
enddo
stop
end
```

Target loop

```
subroutine sub(a,b,i,n,c)
real a(n), b(n)
...
if (c<1) then
   b(i) = b(i) + a(i)
else
   b(i) = b(i-1)+a(i)
endif
return
end
```

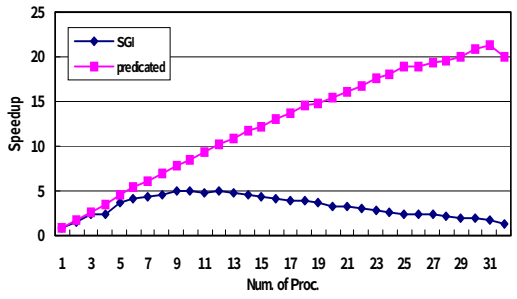
Target loop calls to subroutine “sub” involving “IF” statement, whose branch target has loop-carried dependence.

Input program

```
program main
Real a(n), b(n)
call init(a,b,n)
read(*,*) c
If (c<1) then
!$omp parallel do private(i)
   do i=1,n
      call sub(a,b,i,n,c)
   enddo
else
   do i=1,n
      call sub(a,b,I,n,c)
   enddo
stop
end
```

Output program

Execution on Ogirin® 2000



Result of Execution

The program to which our method is applied

- ran 21 times faster than the sequential one on 31 processors and
- ran 4.3 times faster than the program compiled by SGI® parallelizing compiler.

6. Conclusion

- We have implemented the predicated data-flow analysis in our module.
- Initial evaluation shows the effectiveness of the predicated data-flow analysis.