Dependence Analysis

Michael O'Boyle

February, 2014

Course Structure

- 5 lectures on high level restructuring for parallelism and memory
- Dependence Analysis
- Program Transformation
- Automatic vectorisation
- Automatic parallelisation
- Speculative parallelisation
- Then adaptive compilation

Lecture Overview

- Parallelism
- Types of dependence flow, anti and output
- Distance and direction vectors
- Classification of loop based data dependences
- Dependence tests: gcd, banerjee and Omega

References

- R. Allen and K Kennedy Optimizing compilers for modern architectures: A dependence based approach Morgan Kaufmann 2001. Main reference for this section of lecture notes
- Michael Wolfe High Performance Compilers for Parallel Computing Addison-Wesley 1996.
- H. Zima and B. Chapman. Supercompilers for Parallel and Vector Computers. ACM Press Frontier Series 1990
- Today : The Omega Test: a fast and practical integer programming algorithm for dependence analysis Supercomputing 1992

Programming Parallel Computers

- Two extremes: User specifies parallelism and mapping
- Compiler parallelises and maps "dusty deck" sequential codes
  - Debatable how far this can go
- A popular approach is to break the transformation process into stages
- Transform to maximise parallelism i.e minimise critical path of program execution graph
- Map parallelism so as to minimise "significant" machine costs i.e. communication/ non-local access etc.

Different forms of parallelism

Statement Parallelism

```
cobegin
  a := b + c
  d := e +f
coend
```

Function Parallelism

```
f (a) = if (a <= 1) then return 0
        else return f(a-1) + f(a-2)
        endif
```

Operation Parallelism

```
a = (b+c) * (d+e)
```

Loop Parallelism

```
Do i = 1 , n
  a(i) = b(i)
EndDo
```

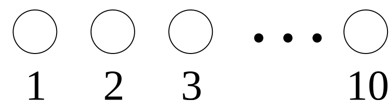
Loop Parallelism / Array Parallelism

Original loop	Parallel loop
`Do i = 1 , n`	`Doall i = 1 , n`
`  a(i) = b(i)`	`  a(i) = b(i)`
`EndDo`	`EndDo`

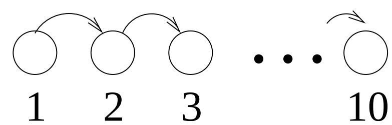
- All iterations of the iterator i can be performed independently
- Independence implies Parallelism
- We will concentrate on loop parallelism O(n) potential parallelism. Statement and Operation - O(1).
- Recursive parallelism is rich but very dynamic. Exploited in functional computational models

Parallelism and Data Dependence

```
Do i = 1,10
   a(i) = b(i) + c(i)
EndDo
```




```
Do i = 1,10
   a(i+1) = a(i)
      + function_call(i)
EndDo
```



Completely parallel each iteration is totally independent

Completely serial each iteration depends on the previous iteration

Note: iterations NOT array elements

Data Dependence

- The relationship between reads and writes to memory has critical impact on parallelism. 3 types of data dependence

```
Flow(true)          Anti       Output
 a =                  =a        a=
   =a               a=          a=
```

- Only data flow dependences are true dependences. Anti and output can be removed by remaining

- Dataflow analysis can be used to defines data dependences on a per block level for scalars but fails in presence of arrays. Need finer grained analysis

Data Dependence

- In general we need to know if two usages of an array access the same memory location and what type of dependence

- Helpful as this can be done relatively cheaply for simple programs

- General dependence is intractable - equivalent to Hilbert's tenth problem

  $a(f(i)) = a(g(i))$ for arbitrary f, g

- Decidable (NP) if f, g linear

Dependence in loops

```
Do i =1,N
 a(f(i)) =
         = a(g(i))
EndDo
```

- Conditions for flow dependence from iteration I_w to I_r
  $1 \le I_w \le I_r \le N \wedge f(I_w) = g(I_r)$

- Conditions for anti dependence from iteration I_r to I_w
  $1 \le I_r < I_w \le N \wedge g(I_r) = f(I_w)$

- Conditions for output dependence from iteration I_{w1} to I_{w2}
  $1 \le I_{w1} < I_{w2} \le N \wedge f(I_{w1}) = f(I_{w2})$

Dependence in loops lexicographical ordering

```
Do i =1,N
  Do j = 1,M
    a(f(i,j),g(i,j)) =
         = a(h(i,j),k(i,j))
  EndDo
EndDo
```

- Lexicographic order on iteration space: $(1,1) < (1,2) \ldots < (1,N) < (2,1) \ldots < (N,M)$
- Conditions for flow dependence from iteration (I_1, J_1) to (I_2, J_2) $(1,1) < (I_1, J_1) < (I_2, J_2) < (N, M)$
- $\wedge f(I_1, J_1) = h(I_2, J_2) \wedge g(I_1, J_1) = k(I_2, J_2)$

Dependence distance and direction: (approx) summarising dependence

```
Do i =1,N
  Do j = 1,M
    a(i,j) = a(i-1,j+1) +1
  EndDo
EndDo
```

- Flow dependence $\{(1,2) \rightarrow (2,1), (1,3) \rightarrow (2,2), (2,2) \rightarrow (3,1)\}$
- Dependence $(I_w, J_w) \rightarrow (I_r, J_r) : I_r - I_w = 1, J_r - J_w = -1$
- Distance vector is [1,-1]. Direction vector [+,-] or [$<, >$] sign of direction. Any: [*], 0 [=],Positive [$<$], Negative [$>$]
- First non zero vector element cannot be negative - why?

Hierarchical Computing of Dependence Directions in loops.

```
Do i =1,N
 a(f(i)) =
         = a(g(i))
EndDo
```

- Test for any dependence from iteration I_w to I_r: $1 \leq I_w, I_r \leq N$ $\wedge f(I_w) = g(I_r)$
- Use this test to test any direction[*].
- If solutions add additional constraints:$[<]$ direction: add $I_w < I_r$, $[=]$ add $I_w = I_r$.
- Extend for multi loops, $[*, *]$ then $[<, *], [=, *]$ etc - hierarchical testing

Classification for simplification : Kennedy approach

```
Do i =1,N
 Do j= 1,N
  Do k = 1,N
    a(5,I+1,J) = a(N,I,K)+c
EndDo
```

- Test for each subscript in turn. If any subscript has no dependence - then no solution
- Subscript in 1st dim contains zero index variables (ZIV)
- Subscript in 2nd dim contains single (I) index variables (SIV)
- Subscript in 3rd dim contains multi (J,K) index variables (MIV)

Separable SIV test

```
Do i =1,N
 Do j= 1,N
  Do k = 1,N
    x(aI+b,..,..) = x(cI+d,..,..)
EndDo
```

- If equations for one iterator appear in only one subscript, we can separate it and solve independently.
- $a \times I_w + b = c \times I_r + d$ Strong SIV, $a = c$, so $I_r - I_w = (b - d)/a$
- If a divides b-d and result is in range of I, then we have the dependence distance. Weak SIV : a or c $=$0.

General SIV test or Greatest Common Divisor

```
Do i =1,N
 Do j= 1,N
  Do k = 1,N
    x(aI+b,..,..) = x(cI+d,..,..)
EndDo
```

- We have $a \times I_w + b = c \times I_r + d$
- If gcd(a,c) does not divide d-b no solution try a=c=2, d=1,b=0
- ELSE ... potentially many solutions.

Banerjee Test

- Basically test for a real solution to a Diophantine equation
- Inaccurate: A real solution does not imply an integer solution

```
Do i =1,N
    x(aI+b,..,..) = x(cI+d,..,..)
EndDo
```

- Flow constraint: $aI_w + b = cI_r + d$ or $h(I_w, I_r) = aI_w - cI_r + b - d = 0$
- Test if h ever becomes 0 in region implies equality
- Intermediate value theorem if max (h) ≥ 0 and min(h) $\leq$ 0 then this is true.

Example using flow constraint

```
Do i =1,100
   a(2*i+3) = a(i+7)
```

- We have $2I_w + 3 = I_r + 7, h = 2I_w - I_r - 4$ and $1 \leq I_w \leq I_r \leq 100$
- Min h = (2*1 -100 -4) = -102, Max h = (2*100-1-4)=195
  $195 > 0 > -102$ hence solution
- Simple example can be extended. Technical difficulties with complex iteration spaces
- Performed sub-script at a time, Used for MIV

Omega Test - Read the paper!

- Most compilers still use classification and special tests for dependence
- However Pugh's Omega Test can solve exactly using integer linear programming.
- Basically state constraints and put into a smart Fourier-Motzkin elimination based solver
- Shown that worst case double exponential cost on manipulating Presburgher formula is frequently low-end polynomial

Lecture Overview

- Parallelism
- Types of dependence flow, anti and output
- Distance and direction vectors
- Classification of loop based data dependences
- Dependence tests: gcd, banerjee and Omega
- Next lecture loop and data transformations