Parallelizing Compiler Transformations

Goal: Map a program written in a sequential language with embedded control flow into a parallel program with units of computation of useful size and not too much communication.

Approach: Source to Source Transformations

a) Recognize data dependence relations in code.

b) Determine desired structure for transformed program.

c) Map original program to equivalent program.

Technical Problems:

a) Determining desirable structures

b) Determining legal transformations

Parallelizing Compiler Transformations

Definition: Equivalent Programs - A transformation is legal if the original and the transformed programs produce exactly the same output for all identical executions.

Two executions of a program are identical executions if they are supplied with the same input data and if every corresponding pair of nondeterministic operations in the two executions produces the same result.

Nondeterminism can be introduced by language constructs (such as Ada's select statement), or by calls to system or library routines that return information about the state external to the program (such as Unix time( ) or read()).

Parallelizing Compiler Transformations

```
subroutine tricky (a, b,n, m,k)
integer n, m, k
real a[m], b[m]
doi=l, n
a[i] = b[k] + a[i] + 100000.0
end do
return
```

(a) original program

```
subroutine tricky(a,b,n,m,k)
integer n, m, k
real a[m], b[m], C
C = b[k] + 100000.0
doi=n, i,-1
a[i] = a[i] + C
end do
return
```

(b) transformed program

```
k = m+l
n=O
call tricky(a,b,n,m,k)
```

(c) possible call to tricky

```
equivalence (a[l], b[n])
k=n
call tricky(a,b,n,m,k)
```

(d) possible call to tricky

Parallelizing Compiler Transformations

Overflow.

If b[k] is a very large number,and a[l] is negative, then changing the order of the additions so that C = b [k] + 100000.0 is executed before the loop body could cause an overflow
to occur in the transformed program that did not occur in the original. Even if the original program would have overflowed, the transformation causes the exception to happen at a different point. This situation complicates debugging, since the transformation is not visible to the programmer.

Finally, if there had been a print statement between the assignment and use of C, the transformation would actually change the output of the program.

Different results

Even if no overflow occurs, the values that are left in the array a may be slightly different because the order of the additions has been changed. The reason is that floating-point numbers are approximations of real numbers ,and the order in which the approximations are applied(rounding) can affect the result. However, for a sequence of commutative and associative integer operations, if no order of evaluation can cause an exception, then all evaluation orders are equivalent.
We call operations that are algebraically but not computationally commutative (or associative) semi-commutative(or semi-associative) operations.
These issues do not arise with Boolean operations, since they do not cause exceptions or compute approximate values.

Memory fault

If k > m but n < 1, the reference to b[k] is illegal. The reference would not be evaluated in the original program because the loop body is never executed, but it would occur in the call shown in Figure l(c) to the transformed code in Figure l(b).e

Different results

a and b may be completelyor partially aliased to one another,changing the values assigned to a in the transformed program. Figurel(d) shows how this might occur: If the call is to the original subroutine, b[k]is changed when i = 1, since k = n and b [n] is aliased to a[l]. In the transformed version, the old value of b [k] is used for all i, since it is read before the loop. Even if the reference to b[k] were moved back inside the loop, the transformed version would still give different results because the loop traversal order has been reversed.

Parallelizing Compiler Transformations

Definition: Legal Transformation- A transformation is legal if, for all semantically correct program executions, the original and the transformed programs produce exactly the same output for identical executions.

Definition: A transformation is legal if, for all semantically correct program executions, the original and the transformed programs produce exactly the same output for identical executions.

Definition 2.1.3 A transformation is legal if, for all semantically correct executions of the original program, the original and the transformed programs perform equivalent operations for identical executions. All permutations of semicommutative operations are considered equivalent.

Parallelizing Compiler Transformations

Approaches

1. Derive Control Flow Graph and Map to Data Flow Graph

2. Map to Static Single Assignment Form

3. Restructure Loops

NOTE: Compilers use a number representations upon which to apply transformations. Control flow graphs, Program dependence graphs, and static-single assignment form are commonly used.

Control Flow Graph

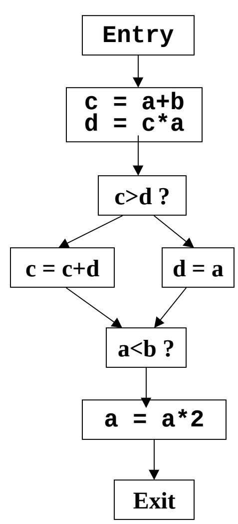
The behavior of a sequential program is determined by specification of flow of control among basic blocks. Control flow is captured in a control flow graph.

The behavior of a parallel program is determined by the dependence of one unit of computation upon data which is assigned values by some other unit of computation. These dependences are captured in a data flow graph.

To convert a sequential program to a parallel program one must map from a control flow graph to a data flow graph.

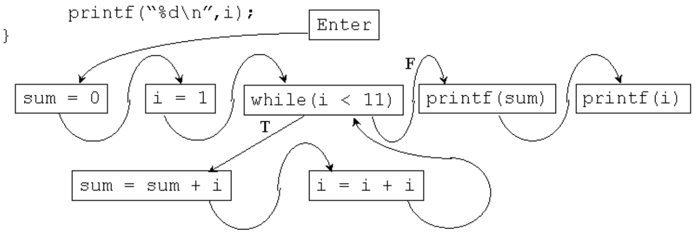
Definition: A control flow graph contains one node for each basic block in a program plus entry and exit nodes. Each node has an directed edge to every node to which it can transfer control. Entry nodes have a directed edge to each node which is an entry point to the program and an exit node has an incoming edge from every node from which the program can exit.


```
procedure test(a,b)
integer c,d
c = a+b
d = c*a
if (c>d) then
c = c+d
else
d = a
while (a<b)
a = a*2
return
```



Control Flow Graph

```
int main() {
      int sum = 0;
      int i = 1;
      while (i < 11) {
            sum = sum + i;
            i = i + 1;
      }
      printf("%d\n",sum);
      printf("%d\n",i);
}
```



Conversion of Control Flow Graphs to Data Flow Graphs

1. There are algorithms for use by compilers.

The primary tasks are:

To determine the units of computation for the parallel program

To determine the variables which are the inputs to the units of computation

To derive firing rules on the data values which are the inputs to the units of computation.

2. We will illustrated the concepts by some examples.

3. The examples are at a statement level but the same concepts apply for units of computation with larger granularity.

Parallelizing Compiler Transformations

```
DO 100 I = 1, 100
        S1
        IF (X(I)) GOTO 200
        S2
100 CONTINUE
        S3
200     S4
```

```
EX1 = .TRUE.
DO 100 I = 1, 100
        IF (EX1) S1
        IF (EX1) EX1 = .NOT.EX1
        IF (EX1) S1
100 CONTINUE
IF (.NOT.EX1) GOTO 200
        S3
200 S4
```


Parallelizing Compiler Transformations

```
DO 200 I = 1, 100
        50 S1
DO 100 J = 1, 100
        S2
        IF (X(I,J)) GOTO 300
        S3
        IF (Y(I,J)) GOTO 50
        S4
100 CONTINUE
        S5
200     CONTINUE
300     S6
```

```
EX1 = .TRUE.
DO 200 I = 1, 100
        50 IF (EX1) S1
IF (EX1) EX2 = .TRUE.
DO 100 J = 1, 100
IF (EX1.AND.EX2) S2
IF (EX1.AND.EX2) EX1 = .NOT.X(I,J)
IF (EX1.AND.EX2) S3
IF (EX1.AND.EX2) EX2 = .NOT.Y(I.J)
IF (EX1.AND.EX2) S4
100 CONTINUE
IF (EX1.AND.NOT.EX2) GOTO 50
IF (EX1) S5
200 CONTINUE
IF (.NOT.EX1) GOTO 300
300 S6
```

Parallelizing Compiler Transformations

```
DO 100 I = 1, 100
IF (A(I).GT.10) GOTO 60
(S1)        A(I) = A(I) + 10
IF (B(I).GT.10) GOTO 80
(S2)        B(I) = B(I) + 10
(S3)        60 A(I) = B(I) + A(I)
(S4)        80 B(I) = A(I) - 5
 100        CONTINUE
```

```
DO  100 I = 1, 100
BR1 = A(I).GT.10
IF (.NOT.BR1) A(I) = A(I) + 10
IF (.NOT.BR1) BR2 = B(I).GT.10
IF (.NOT.BR1.AND.NOT.BR2)  B(I)=B(I)+10
IF (.NOT.BR1.AND.NOT.BR2)  A(I)=B(I)+A(I)
B(I) = A(I) + 5
100  CONTINUE
```


Parallelizing Compiler Transformations

```
IF (x) GOTO 200
100         S1
            GOTO 300
200         S2
IF (y) GOTO 100
300          S3
```

```
                    BR1 = x
100                 IF ((.NOT.BR1).OR. (BB1.AND.BR1)) S1
200                 IF (.NOT.BB1.AND.BR1) S2
                    IF (.NOT.BB1.AND.BR1).AND.y)) THEN  BB1 = .TRUE. GOTO 100
                    ENDIF
300                 S3
```

Parallelizing Compiler Transformations

Parallelization by Conversion to Single Assignment Form

Three conditions : **basic block**

: **Multiple definitions reach a single use point**

: **Loops**

Control of storage requirements

10/18/2001 Parallel Programming - Fall 2001

Parallelizing Compiler Transformations

Identification and Mapping of Basic Blocks

Original	Renamed
X =	X1 =
.	.
= X	= X1
X =	X2 =
.	.
= X	= X2
X =	X =

Renaming within a Basic Block

10/18/2001 Parallel Programming - Fall 2001

Parallelizing Compiler Transformations

Multiple definitions reaching a single use point

Original	Renamed
X = C+D	X2 = C+D
if (p)	if (p)
then X = A+B	then X1 = A+B
	X = X1
	else
	X = X2
	Birthpoint X
Q = X	Q = X

Renaming without loops

Parallelizing Compiler Transformations

Conversion of Loops

Original	Renamed
DO while p ;	DO while p ;
X = X + A(i)	NEW X = X + A(i)
ENDDO	ENDDO

Renaming with loops

Parallelizing Compiler Transformations

Example of Conversion

Original Program	Renamed Program
if (c)	if (c)
then X = ...	then X1 = ...
= X	= X1
DO while p	DO while p
X = ...	NEW X2 = ...
ENDDO	ENDDO
else DO while q	else DO while q
X = ...	NEW X2 = ...
ENDDO	ENDDO

Birthpoint X2

Nested Application of Transformations

Parallelizing Compiler Transformations

```
DO I = 1 to 100                 DO I = 1 to 100
        X = 0               S1          X0(I) = 0
        if (p) then X = 1   S2          if (p) then X1(I) = 1
                            S3             else X1(I) = X0(I)
        Z(I) = X            S4          Z(I) = X1(I)

        C(I) = A(I+1)       S5          C(I) = A1(I+1)
        A(I) = B(I)         S6          T(I) = B(I)
ENDDO                           ENDDO
                            S7          A(*) = T(*)
           (a)                             (b)
```

Example

Parallelizing Compiler Transformations

Dependence Analysis - Notation

A sequential program contains two statements, S1 and S2. S1 occurs before S2. A data dependence exists between the two statements if they read or write a common memory location in such a way that there order of execution must be preserved.

1. Flow dependence - S2 reads a location S1 writes. (true dependence)

2. Anti-dependence - S2 writes a location that S1 reads.

3. Output dependence - S2 writes a location which S1 also writes.

S1: a = b+d;

S2: c = a*3;

S3: a = b + c;

S4: e = a/2;

Parallelizing Compiler Transformations

Forward Substitution of LU Decomposition

```
For [i = 1 to n]
        S1: sum = 0.0;
        for [j = 1 to i -1]
                { S2: sum = sum +LU[ps(i),j] *x[j];
                  S3: x[i] = b[ps(i)] - sum;
                  // ps(i) contains an index value.
                }
```

S2 depends on S1 9
S3 depends on both S1 and S2
S2 depends on S2 (loop carries dependence)
s2 depends on S3. (reads and writes of x[i])

Parallelizing Compiler Transformations

General Problem Statement

```
for [11 = li to ui]
        for [I2 = l2 to u2]
        { S1: a[f1, …fn] = …..;
          S2: ……. = a[g1,….gn];
          S3: ………;
        }
```

f and g are index expressions
Determine dependences among Sn

Parallelizing Compiler Transformations

Loop Transformations

Loop Interchange - Interchange inner and outer loops

Privationization - Give each process a copy of a variable.

Scalar Expansion - Replace a scalar by an array.

Loop Distribution - Split one loop into two loops.

Loop Fusion - Combine two loops into one loop.

Unroll and Jam - Combine unrolling, strip-mining and interchange.

Loop Unrolling - Replicate loop body and do fewer iterations.

Strip Mining - Divide iterations of one loop into two nested loops.

Tiling - Divide loop iterations into rectangular blocks.

Loop Skewing - Alter loop bounds to expose wavefront parallelism.

Parallelizing Compiler Transformations

Loop Interchange

```
for [i = 1 to n]
    for [j = 1 to n]
        a[i,j] = a[i,j] + f(i,j);
    f(i,j) has no side-effects.
cobegin [i = 1 to n]
    for [j = 1 to n]
        a[i,j] = a[i,j] + f(i,j);
Cobegin [j = 1 to n]
    for [i = 1 to n]
        a[i,j] = a[i,j] + f(i,j);
```

Column major versus row major?

Parallelize inner loop?

Interchange loops over i and j?

Each parallelization creates n processes each of which is the ith instance of the inner loop.

When is each preferable?

Parallelizing Compiler Transformations

Privatization and Scalar Expansion

```
for [ i = 1 to n]
    for [j = 1 to n]
    {sum = 0.0;
        for k = 1 to n]
        sum = sum + a[k,j]*b[k,j];
        c[i,j] = sum;
    }
```

Matrix Multiplication

What are the dependences?

```
for [ i = 1 to n]
    for [j = 1 to n]
    {c[i,j] = 0.0;
        for k = 1 to n]
         c[i,j] = c[i,j] + a[k,j]*b[k,j];
    {
```

Privatization

The dependencies among the loops are gone. So we can parallelize at what ever level of granularity and structure is desirable.

Parallelizing Compiler Transformations

Privatization and Scalar Expansion continued

```
Cobegin for [ i = 1 to n]
        for [j = 1 to n]
        c[i,j] = 0.0;
Cobegin [ i = 1 to n]
        for [j = 1 to n]
        {       for k = 1 to n]
                 c[i,j] = c[i,j] + a[i,k]*b[k,j];
        }


Cobegin [ i = 1 to n]
        for [j = 1 to n]
        c[i,j] = 0.0;
        Barrier;
        for [j = 1 to n]
        {       for k = 1 to n]
                 c[i,j] = c[i,j] + a[i,k]*b[k,j];
        }
```

What do these codes do and why would one be preferable over another?

Parallelizing Compiler Transformations

Loop Distribution and Loop Fusion

```
for [i = 1 to n]
        { a[i] = ……;
        ……… = + a[i-1];
        }
for [i = 1 to n]
        { a[i] = ……;
        }
for [i = 1 to n[
        {……… = + a[i-1];
        }
```

What is the dependence in this code?

Parallelizing Compiler Transformations

Unroll and Jam

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

for [ i = 1 to n by 2]
        for [j = 1 to n]
        {c[i,j] = 0.0;
                for k = 1 to n]
                { c[i,j] = c[i,j] + a[k,j]*b[k,j];
                  c[i+1,j] = c[i+1,j] + a[k,j]*b[k,j];
                }
```

Create fewer processes

Parallelizing Compiler Transformations

Strip Mining

```
for [ i = 1 to n by 2]
        for [i =  i to i +1]
        for [j = 1 to n]
        {for k = 1 to n
                for I = i to  i+1
                { c[I,j] = c[I,j] + a[k,j]*b[k,j];
        }
```

Tiling

```
for [i= 1 to n]
        for [j = 1, n]
                b[i,j] = a[j,i];

for [i= 1 to n by W]
        for [j = 1, n by W]
                for [I = I to min(I+W-1,n)]
                        for  [J = j to min(j+W-1,n)]
                        b[I,J] = a[J,I];
```

<u>Matrix Transpose</u>
For every Wth row and column use a block of width W.

How would you parallelize?