1. Provide one example of a program and a data-flow analysis formulation where a region-based analysis would yield better solutions than data-flow analysis.
   a. Specify the data-flow analysis through the set of values, the partial-order operator, and the transfer functions
   b. Specify the composition, meet, and closure operators for the transfer functions for the region-based analysis
   c. Show an example, where the above-formulated region-based analysis would provide a more precise solution than the above-formulated data-flow analysis.

[10]

**Constant-propagation**

**Data flow analysis**

Domain : map (C) from program variable to a constant value
Bottom : NAC (not a constant)
Top : UI (uninitialized or not known)
Direction : Forward

Transfer function : For a statement (S), variable (x) and input map (in),
\[ \text{TF}(S, \text{in})(x) = \begin{cases} \text{identity, if } S \text{ is not an assignment to } x \\ \text{if any operand value is NAC } \Rightarrow \text{TF}(S, \text{in})(x) = \text{NAC} \\ \text{If any operand is UI } \Rightarrow \text{TF}(S, \text{in})(x) = \text{UI} \\ \text{else } \text{TF}(S, \text{in})(x) = \text{value after substitution} \end{cases} \]

Meet : If all predecessor send the variable to same constant C, then C else NAC

**Region based analysis**

Transfer function at exit of subregion S of a region R, \( f_{R,\text{OUT}[s]} = \wedge(\text{Compose the transfer function for predecessor basic blocks } B_i \text{ and } S \text{ along all possible paths from entry of } R \text{ to } S) \)

Meet : \( (f_1 \land f_2)(v) = f_1(v) \land f_2(v) \)

Composition: \( (f_1 \circ f_2)(v) = f_1(f_2(v)) \)
BB1->BB2
BB2->BB3
BB1->BB3

BB1:
x = 2;
y = 3;

BB2:
x = 3;
y = 2;

BB3:
z = x+y

- Using dataflow analysis at the OUT of BB3 will give \( z = \text{NAC} \)
- Using above described region based analysis at the OUT of BB3 will give \( z = 5 \)
2. Consider the following loop nest:

1) for (ii = 0; ii < n; ii = ii+B)
2)   for (jj = 0; jj < n; jj = jj+B)
3)     for (kk = 0; kk < n; kk = kk+B)
4)       for (i = ii; i < ii+B; i++)
5)         for (j = jj; j < jj+B; j++)
6)           for (k = kk; k < kk+B; k++)
7)               Z[i,j] = Z[i,j] + X[i,k]*Y[k,j];

Express the iteration space and the data space of the loop-nest above through the four-tuples \(<F,f,B,b>\) to represent the data-spaces and the iteration space. How many matrices do you need to specify? What does each of them represent. What are the dimensions and the values of these matrices? [5]

**Iteration space:**
B,b for each loop nest specifying the lower and upper bound for iteration variable

\[
B = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}
\]
\[
b = \begin{bmatrix}
0
\end{bmatrix}
\]
\[
i = \begin{bmatrix}
i`
n`
nk`
i`
\end{bmatrix}
\]
\[
j = \begin{bmatrix}
-1
0
-1
0
B-1
0
B-1
0
\end{bmatrix}
\]
\[
k = \begin{bmatrix}
0
B-1
0
0
\end{bmatrix}
\]
\[
n/B = \begin{bmatrix}
0
B-1
0
0
\end{bmatrix}
\]

**Data space:**
F,f for each static access specifying the affine function of loop index variables that produce the array index for all dimensions
$Z[i, j]$

\[ F = \begin{bmatrix}
0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0
\end{bmatrix} \]
\[ f = \begin{bmatrix}
0 \\
0
\end{bmatrix} \]

$X[i, k]$

\[ F = \begin{bmatrix}
0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0
\end{bmatrix} \]
\[ f = \begin{bmatrix}
0 \\
0
\end{bmatrix} \]

$Y[k, j]$

\[ F = \begin{bmatrix}
0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 0
\end{bmatrix} \]
\[ f = \begin{bmatrix}
0 \\
0
\end{bmatrix} \]
3. Consider the following loop-nest

for (i = 0; i <= 1000; i++)
for (j = 0; j <= min(750,i); j++)
X[j+1] = (1 / 3) * (X[j] + X[j+1] + X[j+2]);

Use Fourier-Motzkin elimination to transform this loop with outer-axis k=i+j. Show the working of the algorithm succinctly, to show how you obtain the result. [8]

**Iteration space:**
i ≥ 0, i ≤ 1000, j ≥ 0, j ≤ 750, j ≤ i

Introduce k = i+j

Let's eliminate variable j by using j = k-i

**New iteration space constraints:**
i ≥ 0, i ≤ 1000, k-i ≥ 0, k-i ≤ 750, k-i ≤ i
or
i ≥ 0, i ≤ 1000, k ≥ i, k ≤ i+750, k ≤ 2*i
or
i ≥ 0, i ≤ 1000, i ≤ k, i ≥ k-750, 2*i ≥ k
(1) (2) (3) (4) (5)

k should be outer-axis, so project away i from constraints:

Using 1 and 2, 0 ≤ 1000
Using 1 and 3, 0 ≤ k
Using 4 and 2, k-750 ≤ 1000 => k ≤ 1750
Using 4 and 3, k-750 ≤ k => -750 ≤ 0
Using 5 and 2, k ≤ 2*1000 => k ≤ 2000
Using 5 and 3, k ≤ 2*k

**Constraints for k:**
k ≥ 0, k ≤ 1750

**Constraints for i:**
i ≥ 0, i ≤ 1000, i ≤ k, i ≥ k-750, i ≥ ⌈k/2⌉
i ≥ max( ⌈k/2⌉, k-750), i ≥ 0 is redundant as ⌈k/2⌉ ≥ 0 for k ≥ 0
i ≤ min(k,1000)
for (k = 0; k <= 1750; k++)
    for (i = max(⌈k/2⌉, k-750); i <= min(k,1000); i++)
        X[k-i+1] = (1 / 3) * (X[k-i] + X[k-i+1] + X[k-i+2]);

Similarly can be done for eliminating i instead of j
Suppose there are two array accesses


In a 3-deep loop nest, with indices i, j, and k from the outer to the inner loop. What are all the types of temporal reuse (self-temporal and group-temporal) in this loop nest? Ignore spatial reuse.

Self-temporal reuse

For A[2*i, j, i+j],

\[
F = \begin{bmatrix}
2 & 0 & 0 \\
0 & 1 & 0 \\
1 & 1 & 0 \\
\end{bmatrix}
\quad f = \begin{bmatrix}
0 \\
0 \\
0 \\
\end{bmatrix}
\]

For A[2*i + 4, j-2, i+j],

\[
F = \begin{bmatrix}
2 & 0 & 0 \\
0 & 1 & 0 \\
1 & 1 & 0 \\
\end{bmatrix}
\quad f = \begin{bmatrix}
4 \\
-2 \\
0 \\
\end{bmatrix}
\]

In both the above cases,
Rank of F, r= 2 and loop nest depth d = 3
The difference between iteration variables (i,j,k), (i`,j`,k`) which access the same array index is given by nullspace of F :
2(i-i`) = 0, (j-j`) = 0; i.e. i = i`, j = j`
O(n) self temporal reuse, i.e. an element is accessed O(n) times where n is the num of iterations in loop with iteration variable k
Group-temporal reuse

The access have group temporal reuse, if there exists \((i,j,k)\) and \((i',j',k')\), such that

\[
F \cdot ([i,j,k]^T - [i',j',k']^T) = f_2 - f_1
\]

or

\[
\begin{bmatrix}
2 & 0 & 0 \\
0 & 1 & 0 \\
1 & 1 & 0
\end{bmatrix}
\begin{bmatrix}
i - i' \\
j - j' \\
k - k'
\end{bmatrix}
= \begin{bmatrix}
4 \\
-2 \\
0
\end{bmatrix}
\]

or

\[
i - i' = 2 \\
j - j' = -2
\]

\(k - k'\) can be arbitrary from 0 to \(n\), where \(n\) is the number of iterations in loop with iteration variable \(k\).
5. Consider the following program:

```c
for (i = 0; i < n; i++) {
    A[i + 1] = A[i + 1] * B[i + 1]; //S1
    for (j = i; j < n; j++) {
        C[i,j] = C[i,j] + D[0,i+1,2*j]; //S2
    }
}
```

Apply the algorithm to find parallelism with a constant number of synchronizations to this loop nest.

1. Show the queries you make to construct the program dependence graph. How many ILP queries did you have to make? [4]
2. After inserting a constant number of synchronizations, parallelize each separate loop nest (with no synchronization) if possible.
   a. What are the space-partition constraints? [3]
   b. Show the steps involved in solving the space-partition constraints. [3]
   c. What is the solution to your space-partition constraints? [2]
   d. What is the generated code before eliminating empty iterations and tests from the inner loop (i.e., before applying Fourier Motzkin and before doing case-analysis)? [1]
   e. What is the generated code after applying Fourier Motzkin at each level of the iteration. Show the case analysis and the final generated code at each step. [4]

**1. ILP Queries (Total 9 queries, one for each pair of statements)**

- **Edge between S1,S1:** For both array A
  \[\exists \ i, i', \text{ s.t. } 0 \leq i < n, 0 \leq i' < n, i' > i, i' + 1 = i + 1\]

- **Edge between S2,S2:** For both array C
  \[\exists \ i, i', \ j, j' \text{ s.t. } 0 \leq i < n, 0 \leq i' < n, i \leq j < n, i' \leq j' < n, (i', j') > (i, j), \ i' = i, j' = j\]

- **Edge between S3,S3:** For both array A
  \[\exists \ i, i', \ j, j' \text{ s.t. } 0 \leq i < n, 0 \leq i' < n, i \leq j < n, i' \leq j' < n, (i', j') > (i, j), j' = j\]

- **Edge between S1,S2 and S2,S1:** No dependency as different arrays are accessed
Edge between $S_1, S_3$:
$\exists i, i', j, s.t. \ 0 \leq i < n, 0 \leq i' < n, i' \leq j < n, i < i', i + 1 = j$

Edge between $S_3, S_1$:
$\exists i, i', j, s.t. \ 0 \leq i < n, 0 \leq i' < n, i' \leq j < n, i' < i, i + 1 = j$

Edge between $S_2, S_3$:
$\exists i, i', j, j', s.t. \ 0 \leq i < n, 0 \leq i' < n, i \leq j < n, i' \leq j < n, ((i < i') \text{ or } (i=i' \text{ and } j \leq j')) \ i = i' \text{ and } j = j'$

Edge between $S_3, S_2$:
$\exists i, i', j, j', s.t. \ 0 \leq i < n, 0 \leq i' < n, i \leq j < n, i' \leq j < n, ((i' < i) \text{ or } (i=i' \text{ and } j' < j)) \ i = i' \text{ and } j = j'$

PDG:
$S_1 \rightarrow S_3$
$S_3 \rightarrow S_1$
$S_3 \rightarrow S_3$
$S_2 \rightarrow S_3$
$S_2 \rightarrow S_3$

2. Constant number of synchronizations:
We will have a separate loop nest for each SCC in the PDG.
In this case, $S_1, S_3$ form 1 SCC and $S_2$ is another SCC. Further, $S_3$ depends on $S_2$, so $S_2$ loop will be executed first and $S_3$ will be executed later after synchronization

Loop1:
for (i = 0; i < n; i++) {
   for (j = i; j < n; j++) {
      $C[i,j] = C[i,j] + D[0,i+1,2*j]; //S2$
   }
}

barrier();

Loop2:
for (i = 0; i < n; i++) {
   $A[i + 1] = A[i + 1] * B[i + 1]; //S1$
   for (j = i; j < n; j++) {
   }
}
Space partition constraints

For loop 1, S2 is not dependent on itself, so we can run all iterations in parallel.
Assuming a 2-D processor space,
The space partition constraints are:
For all \((i,j)\) and \((i’,j’)\) such that,
\[
i, i’ \geq 0, \quad j \geq i, \quad j’ \geq i’, \quad i, i’ < n, \quad j, j’ < n, \quad i = i’, \quad j = j’
\]
\[
\begin{bmatrix}
p1, p2 & [i + [p5 = [p1’, p2’ * [i’ + [p5’
\end{bmatrix}
\]
Substituting \(i = i’\) and \(j = j’\), we get,
\[
(p1-p1’) * i + (p2-p2’) * j + (p5-p5’) = 0
\]
\[
(p3-p3’) * i + (p4-p4’) * j + (p6-p6’) = 0
\]
The simplest solution to this is
\[
\begin{bmatrix}
1 & 0 & [p1 + [0 = [i
0 & 1 ] & p2 ] & 0 ] & j
\end{bmatrix}
\]
or
\[
p1 = i \text{ and } p2 = j
\]
The processor space index variables \(p_1 = i\) and \(p_2 = j\) will have range
\(0 \leq p_1 < n, \text{ and } i \leq p_2 < n\),
Applying Fourier-motzkin to eliminate \(i\) from constraints for \(p_2\), we get
\(0 \leq p_1 < n, \text{ and } p_1 \leq p_2 < n\),

For loop2, S3 is dependent on itself and S1 and S3 are also interdependent.

For S1 to S3 interdependence, the space-partition constraint imposed are:
For all \((i)\) and \((i’,j’)\) such that,
\[
i, i’ \geq 0, \quad j’ \geq i’, \quad i, i’ < n, \quad j’ < n, \quad i+1 = j’
\]
\[
\begin{bmatrix}
p1 & [i] & [p5 = [p1’, p2’ * [i’ + [p5’
\end{bmatrix}
\]
\[
p1 * i + p5 = p1’ * i’ + p2’ * j’ + p5’
\]
\[ p_3 \times i + p_6 = p_3' \times i' + p_4'' \times j' + p_6' \]

or

\[ p_1 \times i + p_5 = p_1' \times i' + p_2'' \times (i + 1) + p_5' \]
\[ p_3 \times i + p_6 = p_3' \times i' + p_4'' \times (i + 1) + p_6' \]

or

\[ i \times (p_1 - p_2') + p_5 - p_2' - p_5' = p_1' \times i' \]
\[ i \times (p_3 - p_4') + p_6 - p_6' - p_4' = p_3' \times i' \]

or

\[ p_1 = p_2', \ p_3 = p_4', \ p_5 = p_2' + p_5', \ p_6 = p_6' + p_4', \ p_1' = 0, \ p_3' = 0 \]

The simplest solution for the above constraints are:

\[ p_1 = p_2' = 1, \ p_3 = p_4' = 1, \ p_5 = 0, p_5' = -1, \ p_6 = 0, p_6' = -1, \ p_1' = 0, \ p_3' = 0 \]

For S2,
\[
[ p_1 \ = \ [ \ 1 \ \times \ [ i \ ] + [ 0 \ = \ [ i \\
p_2 \ ] \ ] \\
1 ] \ ]
\]

For S3,
\[
[ p_1 \ = \ [ \ 0, \ 1 \ \times \ [ i' \ + [-1 \ = \ [ j' -1 \\
p_2 \ ] \ ] \\
0, 1 ] \ ] \ ]
\]

The processor space index variables for S1, \( p_1 = p_2 = i \) will have range \( 0 \leq p_1, p_2 < n \)

The processor space index variables for S3, \( p_1 = p_2 = j-1 \) will have range \( i-1 \leq p_1, p_2 < n-1 \)

Applying Fourier-motzkin to eliminate \( i \) from constraints for \( p_1, p_2 \), we get \( -1 \leq p_1, p_2 < n-1 \)
The generated code is:

Loop1:
for (p1 = 0; p1 < n; p1++) {
  for (p2 = p1; p2 < n; p2++) {
    for (i = 0; i < n; i++) {
      for (j = i; j < n; j++) {
        if (p1 == i and p2 == j)
          C[i,j] = C[i,j] + D[0,i+1,2*j]; //S2
      }
    }
  }
}

barrier();

Loop2:
for (p1 = -1; p1 < n; p1++) {
  for (p2 = -1; p2 < n; p2++) {
    for (i = 0; i < n; i++) {
      if (p1 == i and p1 == p2)
        A[i + 1] = A[i + 1] * B[i + 1]; //S1
      for (j = i; j < n; j++) {
        if (p1 == j-1 and p1 == p2)
      }
    }
  }
}

Fourier-Motzkin for S1
p1 ≤ i ≤ p1, 0 ≤ i < n, 0 ≤ p1, -1 ≤ p1, 1 ≤ p2 < n
=> 0 ≤ p1 < n, p2 = p1

Fourier-Motzkin for S3
0 ≤ i < n, p1 + 1 ≤ j ≤ p1+1, i ≤ j < n, -1 ≤ p1, < n, p1 ≤ p2 ≤ p1, 1 ≤ p2 < n
=> -1 ≤ p1 < n-1, p2 = p1
Loop1:
for \( (p_1 = 0; p_1 < n; p_1++) \) {
    for \( (p_2 = p_1; p_2 < n; p_2++) \) {
        for \( (i = 0; i < n; i++) \) {
            for \( (j = i; j < n; j++) \) {
                if\( (p_1 == i \) \) and \( p_2 == j) \)
                    \( C[p_1, p_2] = C[p_1, p_2] + D[0, p_1+1, 2*p_2]; \quad \text{//S2} \)
            }
        }
    }
}

barrier();

Loop2: Case analysis
\( p_1 = -1; \)
for \( (i = 0; i < n; i++) \) {
    for \( (j = i; j < n; j++) \) {
        if\( (p_1 == j-1) \)
            \( A[j] = A[j] * C[i, j]; \quad \text{//S3} \)
    }
}

for \( (p_1 = 0; p_1 < n-1; p_1++) \) {
    for \( (i = 0; i < n; i++) \) {
        if\( (p_1 == i) \)
            \( A[i+1] = A[i+1] * B[i+1]; \quad \text{//S1} \)
        for \( (j = i; j < n; j++) \) {
            if\( (p_1 == j-1) \)
                \( A[j] = A[j] * C[i,j]; \quad \text{//S3} \)
        }
    }
}

\( p_1 = n-1 \)
for \( (i = 0; i < n; i++) \) {
    if\( (p_1 == i) \)
        \( A[i+1] = A[i+1] * B[i+1]; \quad \text{//S1} \)
}
Fourier Motzkin

For S3 in case 1:
\[ i \leq j \leq n-1, \quad p_i + 1 \leq j \leq p_i + 1 \Rightarrow \]
\[ i \leq p_i + 1, \quad p_i + 1 \leq n-1, \quad -1 \leq p_i \leq -1 \Rightarrow \]
\[ i \leq 0, \quad 1 \leq n, \quad 0 \leq i < n \Rightarrow i = 0, \quad n > 0 \]

// p_i = -1;
if(n > 0) \ A[0] = A[0] * C[0, 0 ]; //S3

For S1 in case 2:
\[ 0 \leq i \leq n-1, \quad p_i \leq i \leq p_i \Rightarrow \]
\[ 0 \leq p_i, \quad p_i \leq n-1, \quad 0 \leq p_i \leq n-2 \Rightarrow \]
\[ 0 \leq n-1, \quad 0 \leq n-2 \Rightarrow n > 1 \]

For S3 in case 2:
\[ i \leq j \leq n-1, \quad p_i + 1 \leq j \leq p_i + 1 \Rightarrow \]
\[ i \leq p_i + 1, \quad 0 \leq i \leq n-1 \Rightarrow \]
\[ 0 \leq i \leq \min(n-1, p_i + 1) \Rightarrow \]

For S1 in case 3:
\[ 0 \leq i \leq n-1, \quad p_i \leq i \leq p_i \Rightarrow \]
\[ 0 \leq p_i, \quad p_i \leq n-1, \quad n-1 \leq p_i \leq n-1 \Rightarrow \]
\[ 0 \leq n-1 \Rightarrow n > 1 \]

Loop2:
// p_i = -1;
if(n > 0) \ A[0] = A[0] * C[0, 0 ]; //S3

for (p_i = 0; p_i < n-1; p_i++) {
    \ A[p_i + 1] = A[p_i + 1] * B[p_i + 1]; //S1
    for (i = 0; i \leq \min(n-1, p_i + 1); i++) {
        \ A[p_i + 1] = A[p_i + 1] * C[i, p_i + 1]; //S3
    }
}

// p_i = n-1
\ A[n] = A[n] * B[n]; //S1