COL874: Advanced Compiler Techniques - week 4

Presenter: Fahad Nayyar

[136] Group reuse

- Self reuse: When the same access was accessed again (or reused).
  - Temporal
  - Spatial.
- Group reuse: When reuse is across **different accesses.**
  - Temporal
    - Same memory location is being accessed by dynamic instances of different static accesses.
- Example 1:
  - ***for (i=0; i<n; i++)***
  - ***A[i] = A[i+1] + A[2*i+1];***
    - Anything being access by **kth** iteration by A[i+1] is accessed in **k+1th** iteration by A[i].
    - 3 groups:
      - **A[i] v/s A[i+1]: n-1** or **O(n)** dynamic instances of reuse
      - **A[i] v/s A[2*i+1]: O(n/2)** dynamic instances of reuse acros. Only odd elements of A are reused.
      - **A[i+1] v/s A[2*i+1]:**

[136] Group reuse (Cont.)

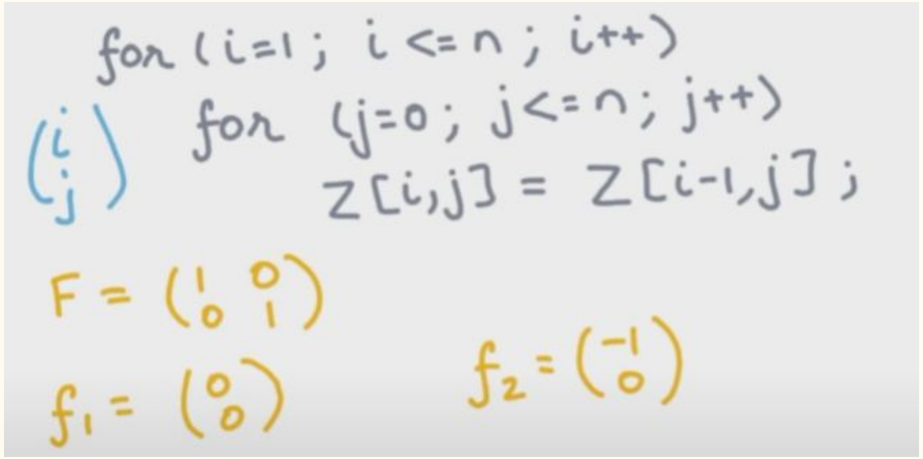
- **Example 2:**
  - ***for (i=0; i<n; i++)***
  - ***for(j=0; j<m' j++)***
  - ***A[i,j] = A[i+1, j+2] + A[2*i, 2*j]***
- **Groups:**
  - **A[i,j] v/s A[i+1, j+2]: O(m*n)** reuse (only boundary conditions are not reused)
  - **A[i,j] v/s A[2*i,2*j]: O(m*n/4)** reuse
    - The higher such multiple in accesses (2 here), the lower becomes the reuse.
  - **A[i+1,j+2] v/s A[2*i,2*j]:** ?

[136] Group reuse (Cont.)

- Reuse are higher if the static accesses have the **same coefficient matrix**.
    - Because if same coefficient matrix, then they will have reuse that is proportional to the number of accesses made by each of those static accesses.
- We want to look at group reuse for only those accesses which have the same coefficient matrix: so that there will be lot of reuse except for the boundary cases (constant offset)
    - **Complexity at compile time** v/s the **payoff**
    - Payoff are higher when the coefficient matrices are equal

[136] Group reuse (Cont.)

- **Example 2:**
    - *for (i=0; i<n; i++)*
    - *for(j=0; j<n' j++)*
    - *Z[i,j] = Z[i-1, j];*
- Iteration vector and coefficient matrices for both accesses:



```
for (i=1; i<=n; i++)
    for (j=0; j<=n; j++)
        Z[i,j] = Z[i-1,j];
```

$$\begin{pmatrix} i \\ j \end{pmatrix}$$

$$F = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

$$f_1 = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \qquad f_2 = \begin{pmatrix} -1 \\ 0 \end{pmatrix}$$

[136] Group reuse (Cont.)

- **Example 2:**
  - ***for (i=0; i<n; i++)***
  - ***for(j=0; j<n’ j++)***
  - ***Z[i,j] = Z[i-1, j];***
- What is the **amount of reuse** in this loop nest?
  - Is there **self temporal reuse**?
    - Look at coefficient matrix F and calculate the rank of the matrix: **rank= 2** in this case. Since it is a **full rank matrix** there is **no self temporal reuse** for both the accesses **Z[i,j]** and **Z[i-1, j]**.
  - Is there **self spatial reuse**?
    - Get rid of the last row of the coefficient matrix F (assuming that the last row of the matrix would completely fit into the cache line), which leaves a **1-rank matrix** thus **there is a self spatial reuse.**

[136] Group reuse (Cont.)

- **Example 2:**
  - ***for (i=0; i<n; i++)***
  - ***for(j=0; j<n' j++)***
  - ***Z[i,j] = Z[i-1, j];***
- What is the amount of reuse in this loop nest?
  - Is there self **group (temporal) reuse**?
    - **Coefficient matrices are equal**: so yes, there is group reuse. **Assumption here:** boundary cases or offsets are small so that reuse is significant.
    - When is **F <u>i1</u> + f1 = F <u>i2</u> + f2**
      - <u>i1</u> - <u>i2</u> = **(-1, 0)**
      - Whenever **i1 = i2+1** and **j1 = j2** there will be a reuse.
  - Is there locality?
    - **Reuse distance?**
      - Subtract the **iteration distance** between accesses of reuse.
      - **O(n)**: Because we will have to go through **n iterations** of the inner loop to have a reuse.
        - (i=1, j=o): Z[i,j] = **Z[1,0]**
        - (i=2, j=0): Z[i-1,j] = **Z[1,0]**
      - Since the **reuse distance is very large O(n)**, there is **no locality in this example.**

[136] Group reuse (Cont.)

- How to calculate the reuse distance generally?
    - In the **distance vector <u>i1</u> - <u>i2</u>**: scan bottom up in this vector and find the **last non-zero dimension**. Is that dimension is **k**, then the reuse distance becomes **O(n^k).**

[137] Significance of reuse?

- **More reuse => more potential for caching**
- **Locality v/s reuse:**
  - **Locality** requires: **reuse + small reuse distance**
  - **Locality** is useful for caches which obey **temporal and spatial locality principle.**
  - **Reuse** can be useful for **different types of caches.**
- Compiler can potentially influence **cache replacement algos (LRU, LFU, FIFO)** or **caching policy** using the **analysis of reuse of access and reuse distances.**
- Some cache replacement algo like **LRU are more sensitive to locality in the workload** whole other like **LFU just care about frequency**.

[137] Significance of reuse analysis: caching

- **Caching** happens at multiple levels in the stack of computer system.
    - Compiler decides what to keep in **memory** and what to keep in **registers**.
    - **Scratch pad memories:** systems where cacheois are manages by the **software/compiler**. Compiler decides what lives in scratchpad/cache and what goes to main memory
        - **Analysis of reuse and reuse distance can influence the policy used by software/compiler to decide what to keep in scratchpad.**
    - **Hardware caches:** widely used. They **prefer recency over frequency** because **locality/recency gives best payoffs across large variety of applications.**

[137] Significance of reuse analysis?

- **More reuse => more potential for caching**
- Which is most important from compiler's developer's point of view?
    - **The one which has more payoff and less complexity?**
- We have talked about 3 kinds of reuse:
    - **Self temporal reuse**
        - **Most consequential** as it gives the most amount of reuse.
        - **K dim** null space of coefficient matrix => **O(n^k)** reuse
            - If there is so much reuse, one can potentially write compiler transformations to reduce the reuse distance and thus influencing the caching policy becomes more feasible for both scratch-pad based memory and hardware caches.
    - **Self spatial reuse**
        - **Limited by the cache line size:** because we make the b **assumption** that the **entire last dimension is available in the cache.**
    - **Group (temporal) reuse**
        - Limited by the number of references or **groups** with the **same coefficient matrix**.

[138] Array data dependencies

- **RAW** (true dependency)
- **WAR** (anti dependency)
- **WAW** (output dependency)
- If there are such data dependencies between accesses, then compiler cannot reorder these statements.
- **Example 1:**
  - ***for (i=0; i<n; i++) {***
  - ***A[2*i+i] = 0;***
  - ***A[2*i] = 1;***
  - ***}***
  - Only WAW dependency possible in these 2 stratic accesses. Is it there?
    - No because one write to only odd indices and other write to only even indices.
  - Thus transformations like **loop fission** is possible
    - ***for (i=0; i<n; i++)***
    - ***A[2*i+i] = 0;***
    - ***for (i=0; i<n; i++)***
    - ***A[2*i+i] = 0;***
  - Or this cabe be transformed into previous (**loop fusion**).

[138] Array data dependencies (cont.)

- **Example 2:** Access at loop nest depth=2
- ***for (i=0; i<n; i++)***
-     ***for(j=0; j<n’ j++) {***
-         ***A[i,2*i] = 0;***
-         ***A[i+j, 2*(i+j)+1] =1;***
- ***}***
    - No WAW: as static accesses are disjoint (first always access even and second always access odd memory in the second dimension).

[138] Array data dependencies (cont.)

- **Example 3:** Access at loop nest depth=2
- ***for (i=0; i<n; i++)***
- ***for(j=0; j<n' j++) {***
- ***A[i,2*i] = 0;***
- ***A[i+j, i+3*j+1] =1;***
- ***}***
  - WAW data dependence?
    - **Examples:** (i,1) =
      - (1,0) vs (1,0)
      - (3,*) vs (2,1)
      - (5,*) vs (3,2)

[138] Array data dependencies (cont.)

- **Example 4:** Reuse of accesses at different depth loop nests.
- ***for (i=0; i<n; i++)***
- ***A[i,2*i] = 0;***
- ***for(j=0; j<n' j++) {***
- ***A[i+j, i+4*j+1] =1;***
- ***}***
    - WAW data dependence?
        - Examples (i,1) =
            - (1,0) vs (1,0)
            - (4,*) vs (3,1)

[138] Array data dependencies (cont.)

A **static access** depends on another as long as **there exists a dynamic instance of the first access that depends on some dynamic instance of the second access.**

- When a dynamic instance depends on another dynamic instance?
  - When they **access the same memory location** and at least one of the accesses is a **write.**
- Formal:
  - Given 2 static access:
    - **<F, f, B, b>** in a d-depth loop nest
    - **<F', f', B', b'>** in a d'-depth loop nest
  - These accesses are data dependent iff
    - 1.) At least one of them is a write.
    - 2.) $\exists\ \underline{i} \in Z^d$ and $\exists\ \underline{j} \in Z^d$ such that
      - $\mathbf{B\,\underline{i} + b >= 0\ ,\ B'\,\underline{i'} + b' >= 0}$
      - $\mathbf{F\,\underline{i} + f\ = F'\,\underline{i'} + f'}$

[138] Array data dependencies (cont.)

- **Example 5:** Data dependence across dynamic instance of the same access
- ***for (i=0; i<n; i++)***
- ***A[i,2*i] = 0;***
- ***for(j=0; j<n' j++) {***
- ***A[i+j, i+2*j+1] =1;***
- ***}***
- We are interested in only those solns where <u>i</u> != <u>i'</u> oraccesses in different iterations

[138] Array data dependencies (cont.)

- **Example 6:** Data dependence across dynamic instance of the same access
- ***for (i=0; i<10; i++)***
-     ***Z[i] = Z[i-1]; // these are 2 accesses : t = Z[i-1] ; Z[i] = t***
- ***}***
- **Data dependence b/w Z[i] and Z[i-1]?**
  - **Iteration space boundaries: 1<=i<=10, 1<=i<=10**
  - **i = i' -1**
- **Data dependence b/w Z[i] and itself?**
  - **i = i' ? :** no soln as in case of same memory access, we do not care about same iteration

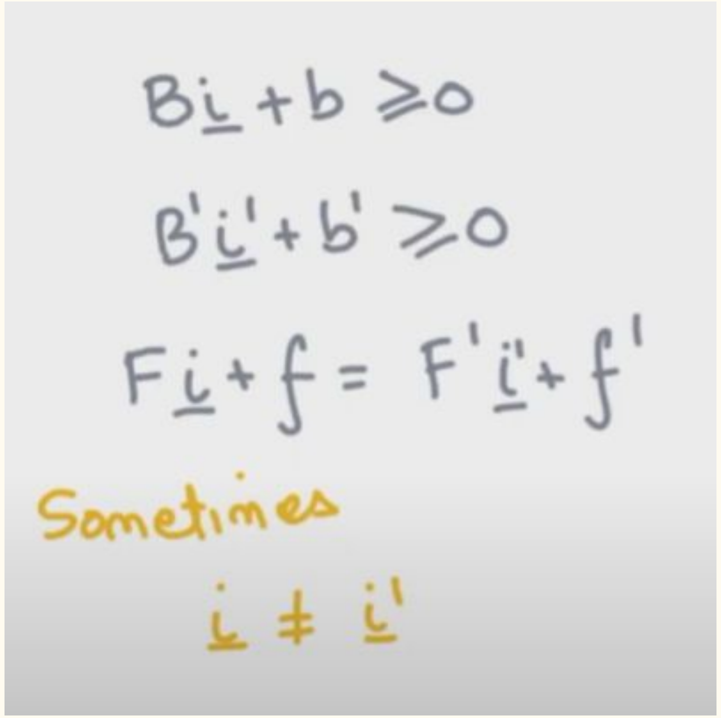
[138] Array data dependencies (cont.)

- **Example 7:** Data dependence across dynamic instance of the same access
- ***for (i=0; i<10; i++)***
- ***Z[0] = Z[i-1]; // these are 2 accesses : t = Z[i-1] ; Z[i] = t***
- ***}***
- **Data dependence bw Z[0] and itself?**
  - **0 = 0 …** all pairs of (i, i') such that i != i' satisfies.

[138] Array data dependencies (cont.)

- **Example 7:** Data dependence across dynamic instance of the same access
- ***for (i=0; i<10; i++)***
- ***Z[i] = Z[0]; // these are 2 accesses : t = Z[i-1] ; Z[i] = t***
- ***}***
- **Data dependence bw Z[0] and itself?**
  - No, as RAR

[139] Solving Data Dependence Constraints



$$B\underline{i} + b \geq 0$$

$$B'\underline{i}' + b' \geq 0$$

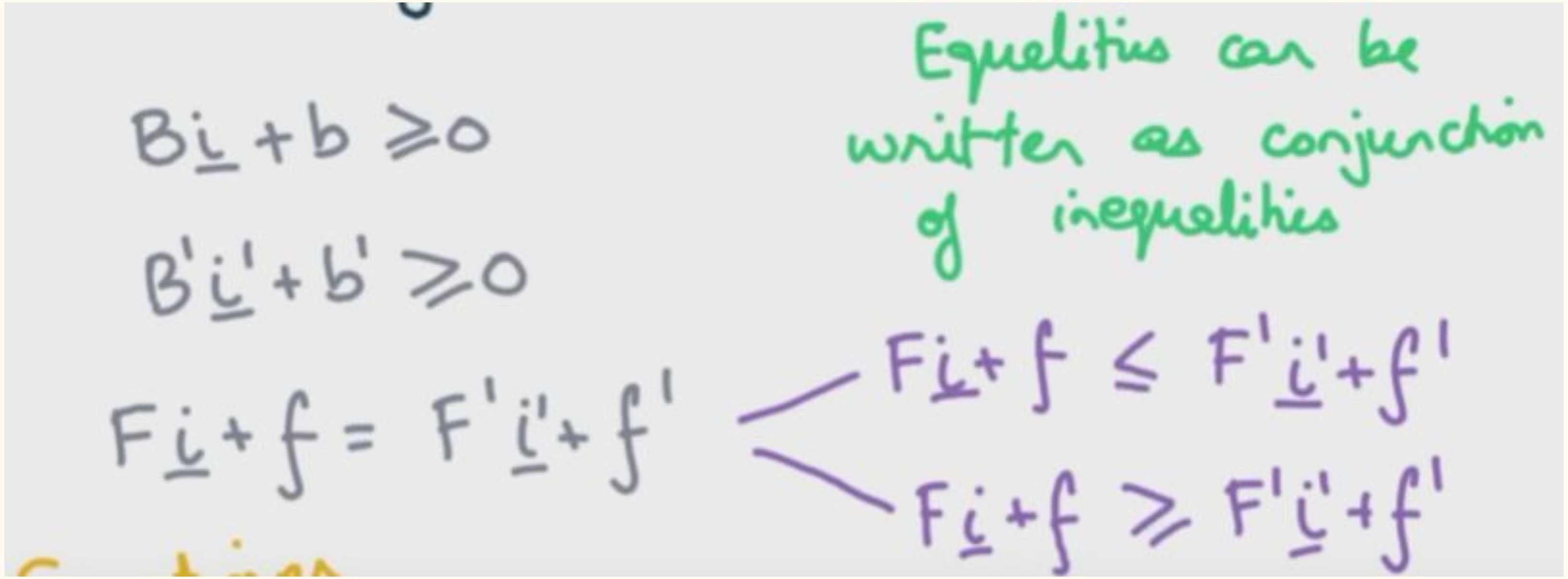
$$F\underline{i} + f = F'\underline{i}' + f'$$

Sometimes

$$\underline{i} \neq \underline{i}'$$

- Find <u>i</u> and <u>i’</u>
  - <u>i</u> and <u>i’</u> are the iteration vectors of the loop nest of first and second static access respectively.
- Whether there exist dynamic instance of first access which conflicts with the dynamic instance of the second access.
- For same static access: extra constraints.

[139] Solving the system of equations : **linear** inequalities



$$B\underline{i} + b \geq 0$$

$$B'\underline{i}' + b' \geq 0$$

$$F\underline{i} + f = F'\underline{i}' + f'$$

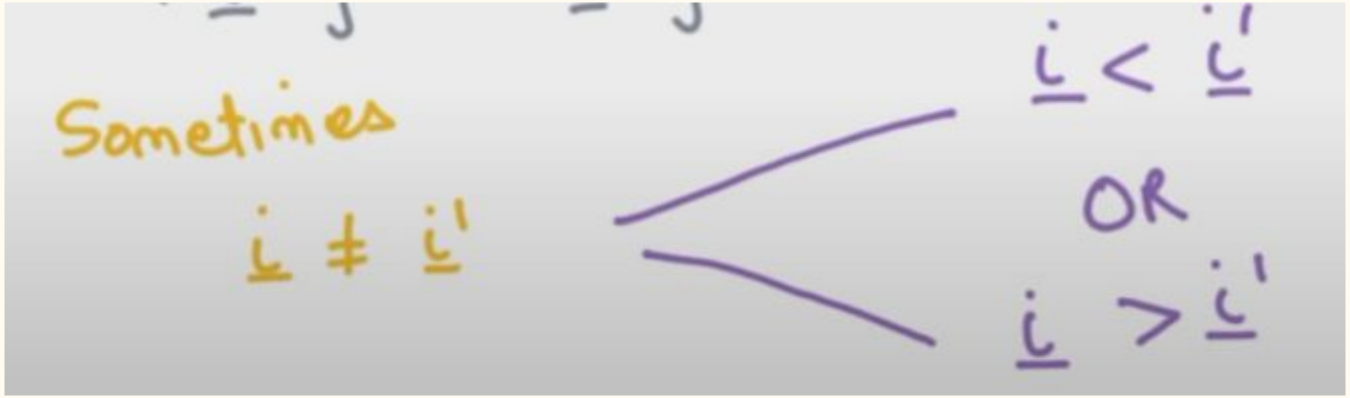
$$F\underline{i} + f \leq F'\underline{i}' + f'$$

$$F\underline{i} + f \geq F'\underline{i}' + f'$$

Equalities can be written as conjunction of inequalities

[139] Solving the system of equations: **disequality** (issues of **disjunction**)

- Disequalities can be written only as disjunctions.


Sometimes
$\underline{i} \neq \underline{i}'$
$\underline{i} < \underline{i}'$
OR
$\underline{i} > \underline{i}'$

[139] Solving the system of equations (cont.)

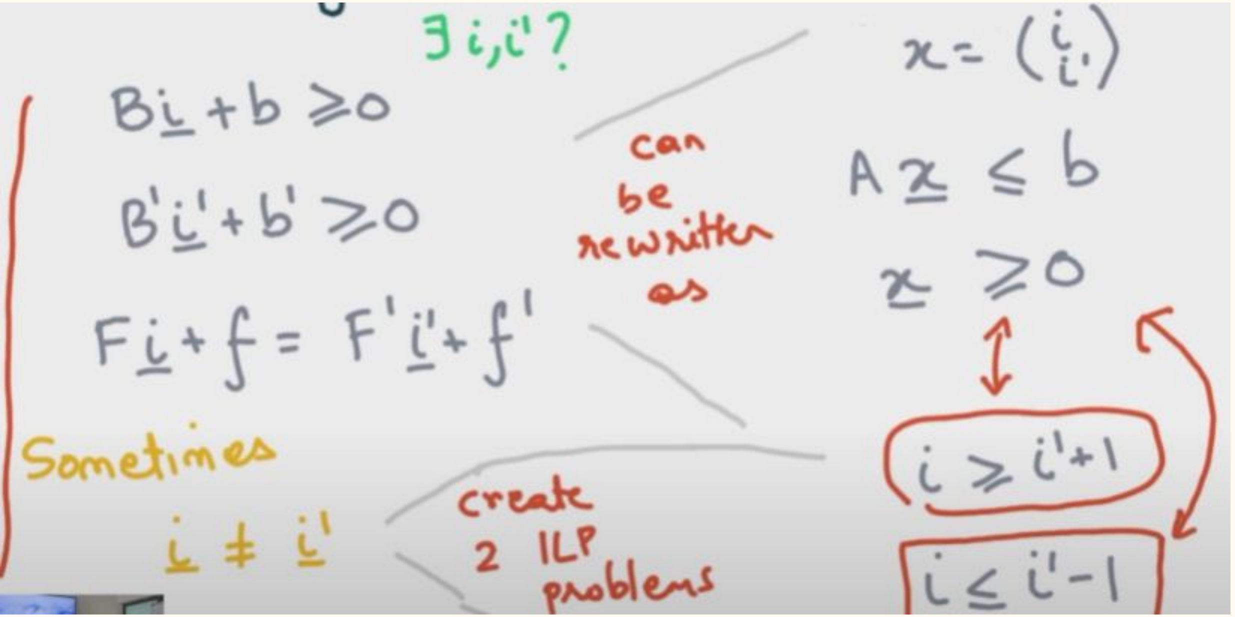
- The problem is **NP - complete** or intractable
  - No polynomial time algo is known.
- But there are many efficient heuristical algos which solve common/practical cases very fast.
- If loop nest is not very large (vectors i and i’ are small) then even exponential time algo works fine until we can do good optimizations!

[139] Solving the system of equations: ILP

- Also NP-complete
- Our system of equations can be **reduced** to ILP
- Many **solvers** exist for ILP as it is an important class of problem (many practical problems can be reduced to ILP).
  - Chapter 5 of this book: Decision Procedures -- An Algorithmic Point of View (decision-procedures.org)
  - **Note** that LP is solvable in poly-time and many of the ILP solvers run multiple instances of LP problem to solve ILP.

$$\begin{aligned} A\underline{x} &\leq \underline{b} \\ \underline{x} &\geqslant 0 \\ \underline{x} &\in \mathbb{Z}^n \end{aligned} \qquad \exists x?$$

[139] Reduction to **2** ILP problems



$\exists\, i, i'$?

$B\underline{i} + b \geq 0$

$B'\underline{i'} + b' \geq 0$

$F\underline{i} + f = F'\underline{i'} + f'$

can be rewritten as

$\underline{x} = \begin{pmatrix} i \\ i' \end{pmatrix}$

$A\underline{x} \leq b$

$\underline{x} \geq 0$

$\updownarrow$

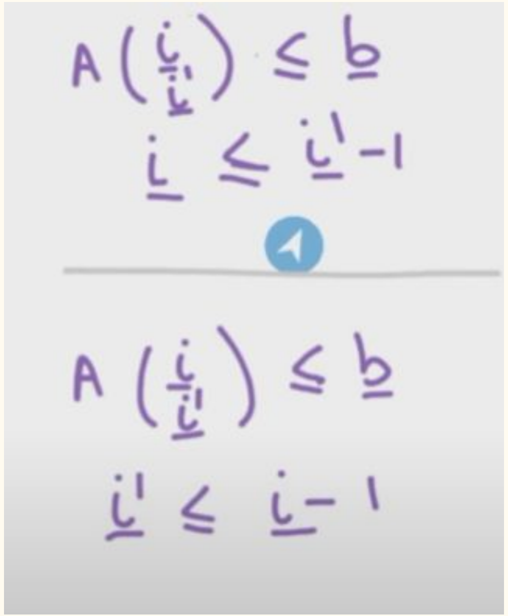
$\boxed{i \geq i' + 1}$

$\boxed{i \leq i' - 1}$

Sometimes

$\underline{i} \neq \underline{i'}$

create 2 ILP problems



$A \begin{pmatrix} \underline{i} \\ \underline{i'} \end{pmatrix} \leq \underline{b}$

$\underline{i} \leq \underline{i'} - 1$

$A \begin{pmatrix} \underline{i} \\ \underline{i'} \end{pmatrix} \leq \underline{b}$

$\underline{i'} \leq \underline{i} - 1$

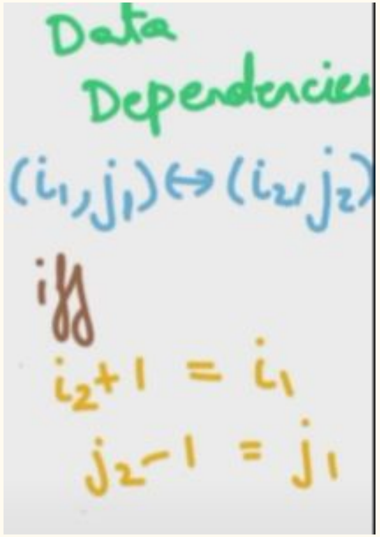
[140] Finding synchronization free parallelism

- Why we want to find **data dependency**?
    - To find parallelism in the program.
    - If there is no data dependency across different statements in loop body or across different iterations of the same statement then we can potentially parallelize that loop on the available number of processors.

[140] Possibility of parallelization even in presence of data dependency

- Example 1:
  - for (i=0; i<1000; i++)
  -     for(j=0; j<100 j++)
  -         A[i,j] = A[i+1, j-1];
- Data dependencies ? yes!
- Still we can parallelize this loop almost to the order of 1000
  - We do not want any synchronization in the parallelization

[140] Possibility of parallelization even in presence of data dependency



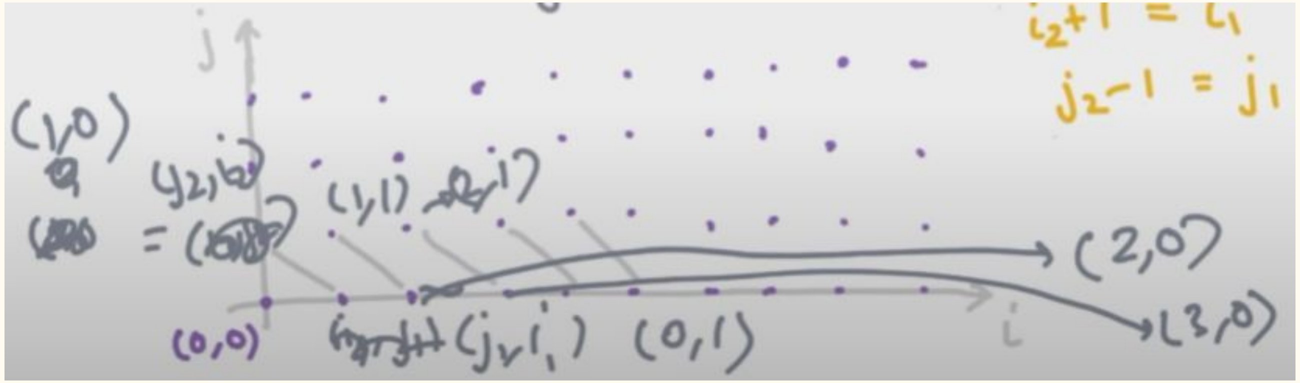
Data Dependencies

$(i_1, j_1) \leftrightarrow (i_2, j_2)$

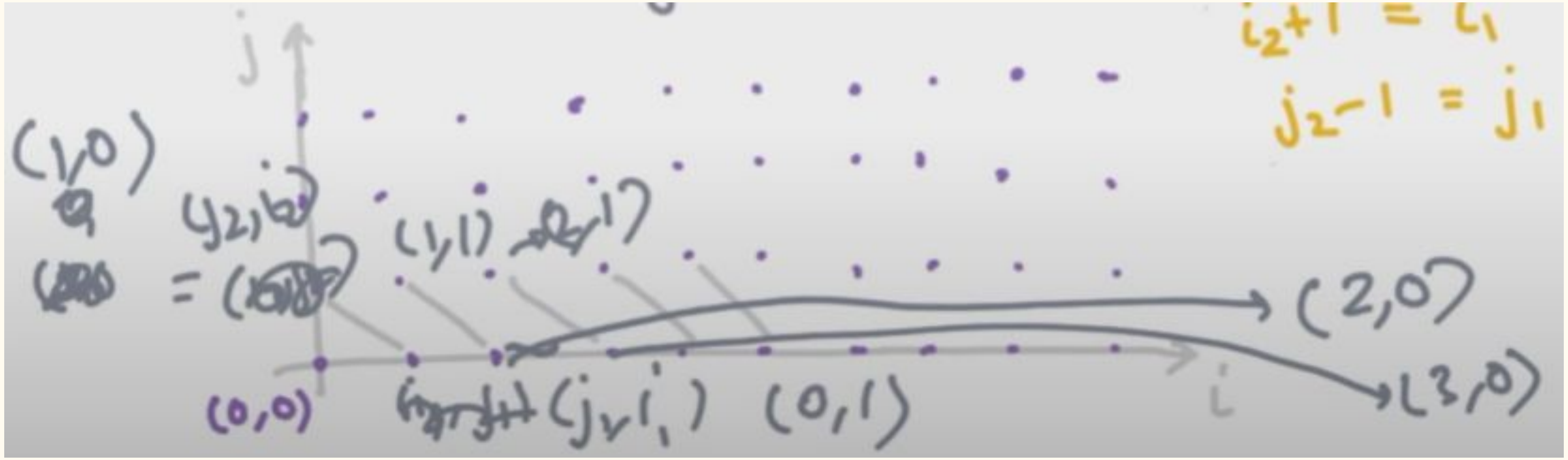
iff

$i_2 + 1 = i_1$

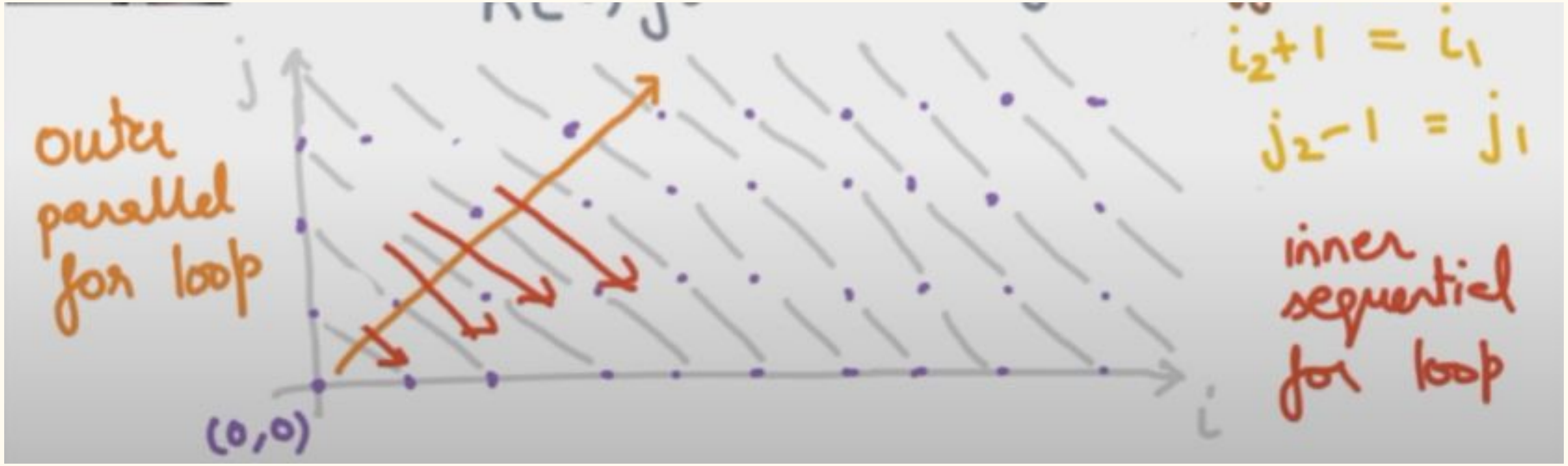
$j_2 - 1 = j_1$



[140] Possibility of parallelization even in presence of data dependency



[140] Possibility of parallelization even in presence of data dependency



[140] Transformed loop

```
par for (k=0; k<=1998; k++)
    for (l=0; l<=min(k,1999-k); l++)
        A[k-l, l] = A[k-l, l-1]
```

Redefined indices from (i,j) to (k,l)

- Even for uni-processor system: this transformation is useful as it has reduced the reuse distance bw data accesses. Better locality of reference!