RUTGERS

Principles of Programming Languages
CS 314

Recitation 12

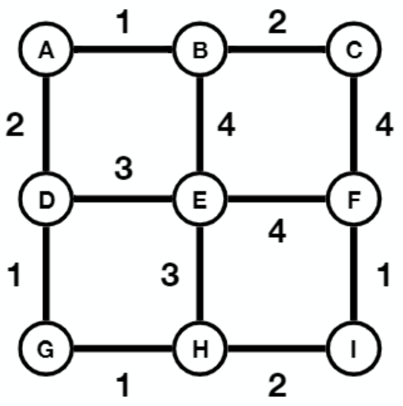
Topics Today

- Project 3
  - Data Structure
  - One-way handshaking
  - N-way handshaking
  - Filter matched nodes
- Loop Parallelization

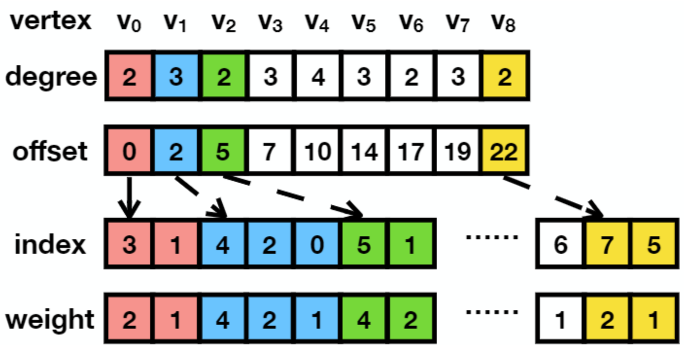
One-Way handshaking algorithm

- Find a matching in the graph

- Data Structure – adjacency array
  - degree: vertex degree
  - index: neighbor list
  - weight: edge weight
  - offset: location in the neighbor list

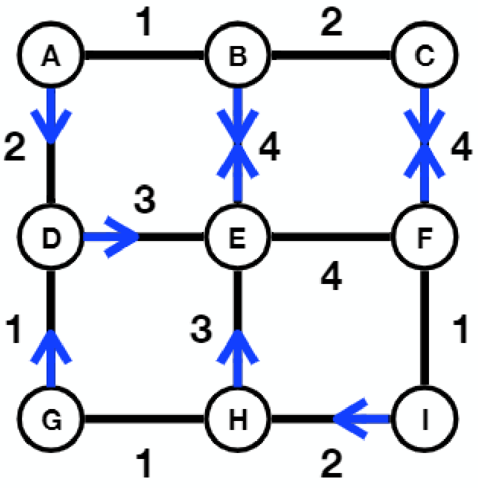


(a) Original Graph



One-Way handshaking algorithm

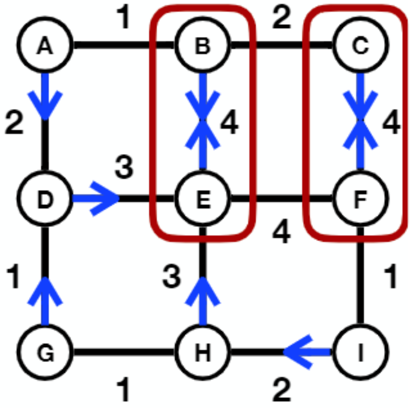
- Step 1: Each vertex extends a hand to its strongest neighbor
  - Strongest neighbor
    - Vertex on the maximum-weight edge
    - Vertex has smaller index
    - A greedy algorithm that aims to maximize the total weight in the matching
  - E.g. V_A> V_D, V_E -> V_B



(b) Extend a Hand to the Strongest Neighbor

One-Way handshaking algorithm

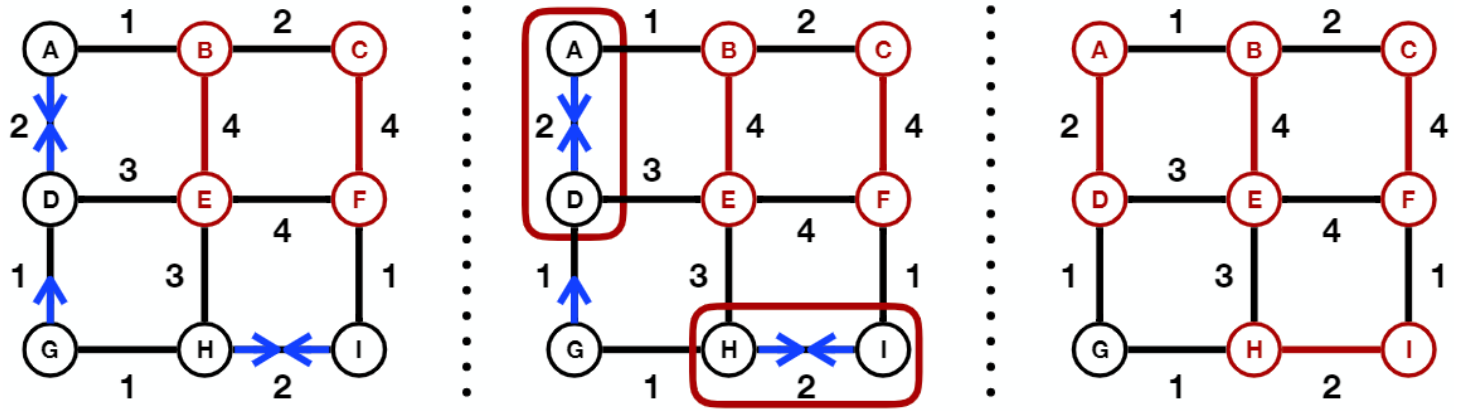
- Step 2: Each vertex checks its strongest neighbor
  - If strongest neighbor extends back a hand
    - Two vertices are matched (no data race)
  - If strongest neighbor extends a hand to other vertex
    - The vertex is not matched in this round
  - If there is no strongest neighbor (no unmatched neighbor vertex)
    - Set res[] to **NO_MATCHED_NODES**(= -2)



(c) Check Handshaking and Matching Vertices

One-Way handshaking algorithm

- Repeat Step 1 & 2



(d) Repeat (b), (c) with Unmatched Vertices

(e) Repeat (b), (c) with Unmatched Vertices

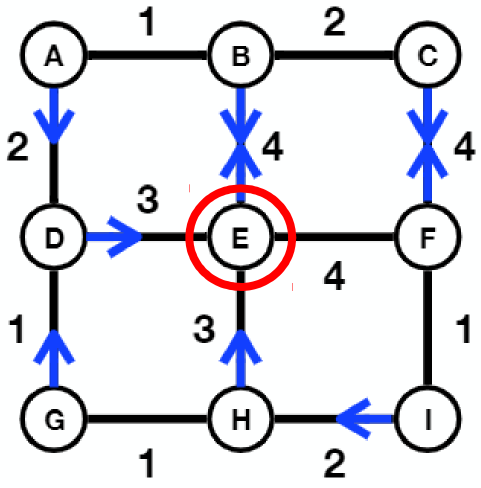
(f) One-Way Handshaking Matching Result

N-Way handshaking algorithm

- Some vertices in the graph may have many neighbors which extend a hand to.
- E.g. V_B, V_D, V_H -> V_E

- Neither V_D and V_H can be matched at this round

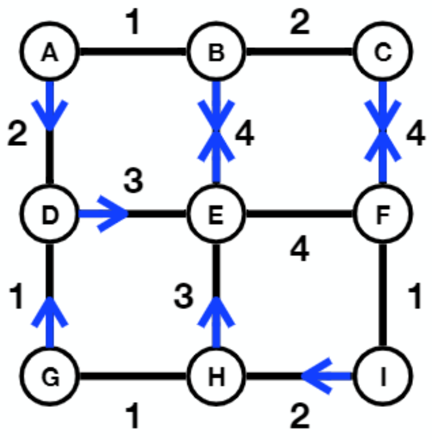
- Extends N hands instead of one



(b) Extend a Hand to the Strongest Neighbor

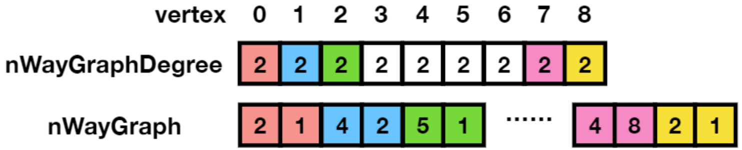
N-Way handshaking algorithm

- Step 1: Extends at most N hands at once (N = 2 in the example)
  - Extends hands to N strongest neighbors
  - May be less than N unmatched neighbors



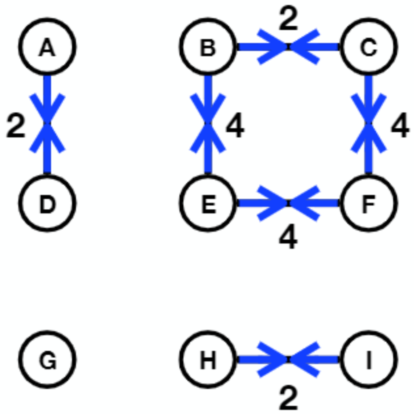
(b) Extend a Hand to the Strongest Neighbor

- N-way Graph Data Structure
  - nWayGraphDegree: vertex degree in n-way graph
  - nWayGraph: adjacency list of each vertex



N-Way handshaking algorithm

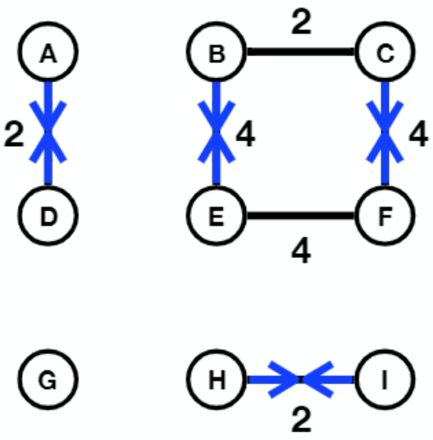
- Step 2: prune edges that two end vertices have no handshaking
  - So that vertices like V_D and V_H can match to other vertices
  - The pruned N-way graph has maximum degree of N (N=2 in the example)



(c) Discard Edges without a Handshaking

N-Way handshaking algorithm

- Step 3: Do one-way handshaking on the new graph
  - More vertices can be matched!
- Be careful that:
  - Vertices have no strongest neighbors in the pruned N-way graph **CANNOT** be set to **NO_MATCHED_NODES**
  - Only do one pass of one-way handshaking



(d) Do One-Way Hand-shaking on the new graph

N-Way handshaking algorithm

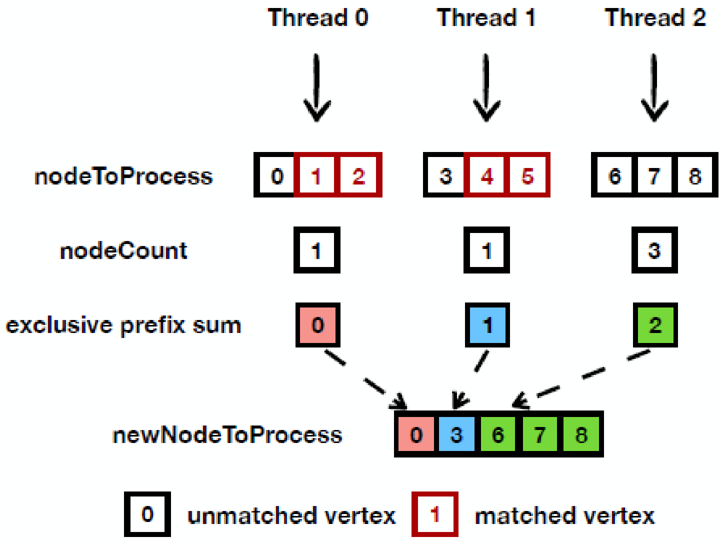
- Repeat Step 1, 2, 3
- May be different matching results, compared to one-way handshaking match

Filter matched results

- After each pass of handshaking matching, some vertices are matched.
- We want to filter out those matched vertices so that each process/thread can handle same amount of work.

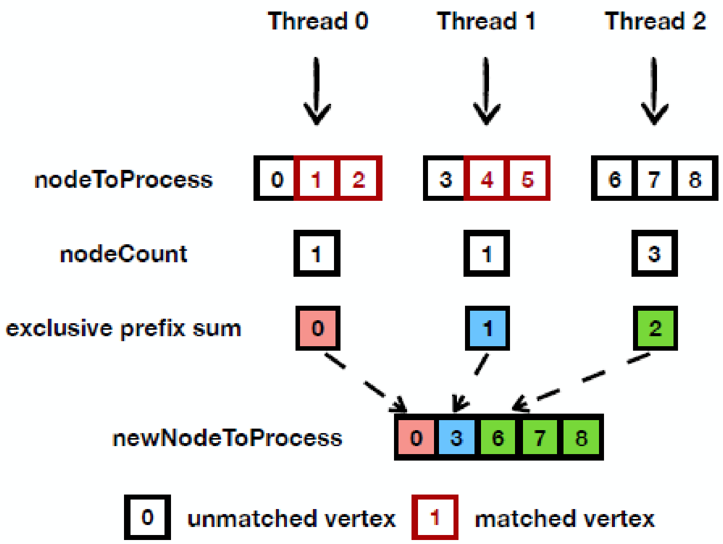
Filter matched results

- After each pass of handshaking matching, some vertices are matched.
- We want to filter out those matched vertices so that each process/thread can handle same amount of work.



Filter matched results

- Step 1: Each thread counts the number of unmatched vertices in its range
- Step 2: Perform an exclusive prefix sum
- Step: Use the result of prefix sum to put vertices into new location



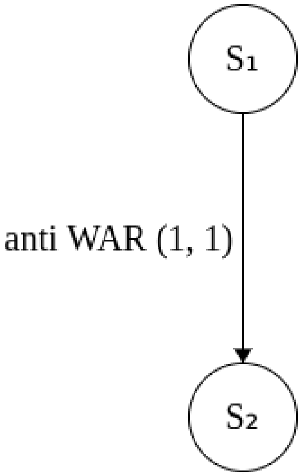
Loop Parallelization

- Give the following nested loop:

```
do i = 2, 10
        do j = 2, 10
S1:          a(i, j) = b(i + 1, j + 1) + 2
S2:          b(i, j) = i + j – 1
        enddo
enddo
```

Show the statement-level dependence graph with distance vectors, along with the dependence graph for statement instances

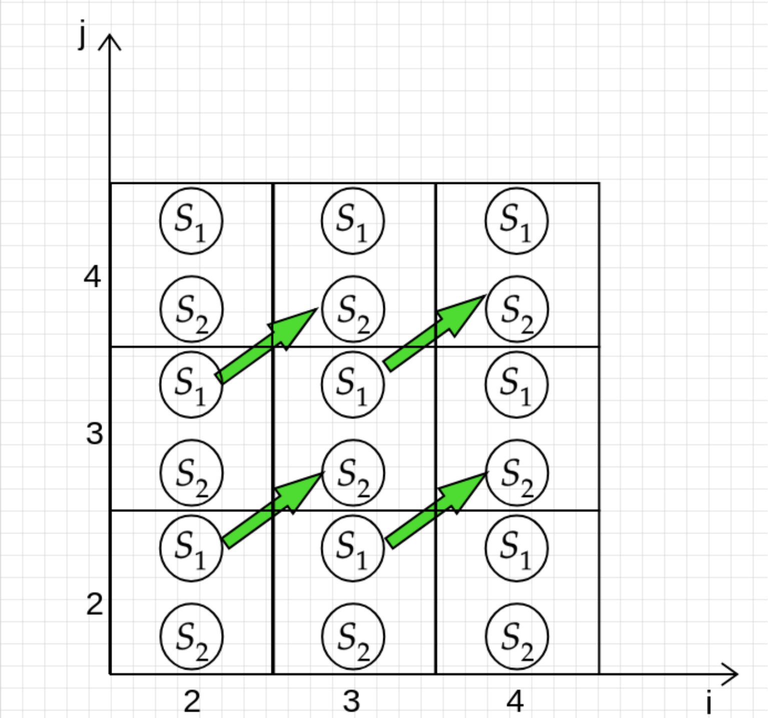
Loop Parallelization



Statement Level Dependence graph & distance vector

Loop Parallelization

Dependence graph for statement instances



Loop Parallelization

- Provide one valid affine schedule for statements S_1 and S_2 such that $p(S_1) = C_{11} * i + C_{12} * j + d_1$ and $p(S_2) = C_{21} * i + C_{22} * j + d_2$ in order to achieve synchronization-free parallelism. (Hint: $d_1 = d_2 = 0$)

```
do i = 2, 10
      do j = 2, 10
S1:       a(i, j) = b(i + 1, j + 1) + 2
S2:       b(i, j) = i + j – 1
      enddo
enddo
```

Loop Parallelization

- First, for statement S_1 and S_2, using the distance vector

$$i_1 + 1 = i_2,\ j_1 + 1 = j_2$$

- To find C_{11} , C_{12} , d_1 and $C_{21,}$ C_{22} , d_2 such that $p(S_1) = p(S_2)$

$$C_{11} * i_1 + C_{12} * j_1 + d_1 = C_{21} * i_2 + C_{22} * j_2 + d_2$$

- Simplify this results:

$$(C_{11} – C_{21}) * i_1 + (C_{12} – C_{22}) * j_1 + (d_1 – d_2 – C_{21} – C_{22}) = 0$$

- Solving this gives

$$C_{11} – C_{21} = 0,\ C_{12} – C_{22} = 0,\ d_1 – d_2 – C_{21} – C_{22} = 0$$

Loop Parallelization

$$C_{11} – C_{21} = 0, C_{12} – C_{22} = 0, d_1 – d_2 – C_{21} – C_{22} = 0$$

- Use the hint: $d_1 = d_2 = 0$, one possible solution

$$C_{11} = C_{21} = 1, C_{12} = C_{22} = -1$$

$$p(S_1) = i - j, p(S_2) = i - j$$

Loop Parallelization

- Generate two-level loop code for the affine schedule you provided. (outermost loop: p, innermost loop: i)

```
do i = 2, 10
      do j = 2, 10
S1:       a(i, j) = b(i + 1, j + 1) + 2
S2:       b(i, j) = i + j – 1
      enddo
enddo
```

$p(S_1) = i - j, \; p(S_2) = i - j$

Loop Parallelization

- First find the range for p:

```
do i = 2, 10
      do j = 2, 10
S1:       a(i, j) = b(i + 1, j + 1) + 2
S2:       b(i, j) = i + j – 1
      enddo
enddo
```

$p(S_1) = i - j,\ p(S_2) = i - j$

```
do p = -8, 8
   do i = 2, 10
      do j = 2, 10
         if (i - j == p)
            a(i, j) = b(i + 1, j + 1) + 2
            b(i, j) = i + j - 1
         endif
      enddo
   enddo
enddo
```

Loop Parallelization

- Eliminate j by doing Fourier-Motzkin elimination: (p = i – j)

$$2 <= i - p <= 10$$

- also since 2 <= i <= 10, so we have

```
do p = -8, 8
    do i = max(2, 2 + p), min(10, p + 10)
        a(i, i - p) = b(i + 1, i - p + 1) + 2
        b(i, i - p) = i + i - p - 1
    enddo
enddo
```