tUPL Parallel Programming Paradigm

tUPL

- Free Computer Programming from common artifacts like data structures, data dependencies, explicit parallelism constructs
- Harness a compilation framework such that
  - Data structures are generated automatically
  - Data dependencies are turned into opportunities to optimize performance
  - Parallel execution is guaranteed

Basic **tUPL** Data Type

< token, data >

Formally, this basic data type is even further stripped down to

$$< \text{token} >_{(A,\ F_A)}$$

With A the "shared" space in which data is stored, and with F_A an address function on A, s.t. data is represented as:

$$A\ [F_A(<\text{token}>)]$$

So data == $A\ [F_A(<\text{token}>)]$

Address function F_A

F_A can be any function, but mostly it is an affine mapping/projection:

$$Z^n \rightarrow Z^k$$

With n being the number of fields in token and k the dimensionality of A. So F_A can be represented as

$$Addr(t) = \vec{m} + Mt^T = \begin{pmatrix} m_{10} \\ \dots \\ m_{k0} \end{pmatrix} + \begin{pmatrix} m_{11} & m_{12} & \dots & m_{1n} \\ \dots & \dots & \dots & \dots \\ m_{k1} & m_{k2} & \dots & m_{kn} \end{pmatrix} t^T$$

NOTE!!!!

A [I, J] = 5.0

does **NOT** mean that element [I, J] of

Matrix A, or of a

2-Dimensional Array A

is assigned the value 5.0.

BUT:

5.0 is stored in A at [F_A (I, J)], with F_A = Id, or that

the data value of < I, J >$_{(A, F_A)}$ becomes 5.0, or that

< I, J, data > = < I, J, 5.0 >*

*Note that tokens can be more dimensional: token tuples t

In case tuples have more than one field, then t.i represents the i^{th} field of t

Alternative (Short Hand) Notation

Note that from in general from the context it is clear with which address function a shared space (A) is being addressed. So, in these cases an alternative notation is used:

A [token] for A [F_A(<token>)] and
@A [token] for F_A(<token>)

In this case A [@A [token]] = A [F_A(<token>)] = A [token]

Multiple Shared Spaces and Associated Address Function per Shared Space

Consider the following **tUPL** code fragment:

```
A[I,J] = A[I-1,2*J] + B[J]
```

Then in this code fragment we have 2 shared spaces:

A and B

and 3 address functions:

$$F_A^1 = Id = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} < I, J >$$

$$F_A^2 = \begin{pmatrix} -1 \\ 0 \end{pmatrix} + \begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix} < I, J >$$

$$F_B = \begin{pmatrix} 0 & 1 \end{pmatrix} < I, J >$$

So, for token t = <I,J> perform:

$A[F_A^1(t)] <- A[F_A^2(t)] + B[F_B(t)]$

SO, data structures as we know them do not exists in **tUPL**, only

single storage locations for each data item, represented by token tuples

We need a means to express a collection or set of these single storage locations

➔ **(Token) Tuple Reservoirs**

Examples of Tuple Reservoirs (I)

A **Digraph G(V,E)**:

T = { **<u , v>** | **u**, **v** ε V and (**u, v**) ε E }

with address function Weigth [**u**, **v**] representing the address at which the weight of edge (**u**,**v**) is stored

A **Sparse Matrix A**:

T = { **<i , j>** | at row i and column j

there is a nnz element}

with address function Value [**i , j**] representing the address at which the value of matrix A [**i**, **j**] is stored

Examples of Tuple Reservoirs (II)

A **Linked List** (of single storage locations):

$\mathbf{T} = \{ \langle \mathbf{i}_k, \mathbf{j}_k \rangle \mid 1 <= \mathbf{k} <= n,$

for every $\mathbf{j}_k$, $1 <= \mathbf{k} < n$,

there exists exactly one $\mathbf{i}_m$, such that $\mathbf{j}_k = \mathbf{i}_m$, and

for all $\mathbf{j}_k$, $1 <= \mathbf{k} <= n$,

the values are different$\}$

Together with an address function Value [$\mathbf{i}_k$, $\mathbf{j}_k$] representing the value at the $\mathbf{k}^{th}$ position in the list.

OR address function Value [$\mathbf{i}_k$] ! (**tUPL** allows both)

Examples of Tuple Reservoirs (III)

Relational Database Tables

T = { **< i >** | 1 <= **i** <= n, with **i** representing the **i**th record in the database table}

and associated address functions:

$field_1$ [**i**], $field_2$ [**i**], …, $field_t$ [**i**]

tUPL Loop Structures

Two BASIC Loop Structures:

```
forelem ( t; t ε T )
whilelem ( t; t ε T )
```

Both structures are inherently

parallel and non-deterministic

This means that any tuple of T can be taken at any time and its execution is atomic.

In the **`forelem`** structure every tuple is taken exactly once, while in the **`whilelem`** every tuple can be taken an arbitrary number of times

tUPL Loop Body

```
if ( Cond_1 )
{
   Serial_Code_1 (< t >)
}
if ( Cond_2 )
{
   Serial_Code_2 (< t >)
}
…
if ( Cond_n )
{
   Serial_Code_n (< t >)
}
```

Scheduling **forelem** (t; t ε T)

- For each execution of a tuple exactly one of the tuples is chosen with a valid conditional serial code and which has not been executed so far.
- If there are no tuples left with a valid conditional serial code, then the **forelem** loop terminates.

Note that if the conditions are not carefully chosen it can happen that the **forelem** loop terminates before all tuples have been executed.

Scheduling **`whilelem`** (t; t ε T)

- For each execution of a tuple exactly one of the tuples with a valid conditional serial code is chosen.
- If there are no tuples left with a valid conditional serial code, then the **`whilelem`** loop terminates.
- Any loop scheduling for a **`whilelem`** loop must guarantee that every tuple with a valid conditional serial code that is continuously enabled beyond a certain point is taken infinitely many times (cf. just computation).

Example I

Sparse Matrix-Vector Multiplication

```
forelem ( t; t ε T )
{
  Value_C[t.i]+= Value_A[t.i,t.j]
                 * Value_B[t.j]
}
```

Example II

SORTING

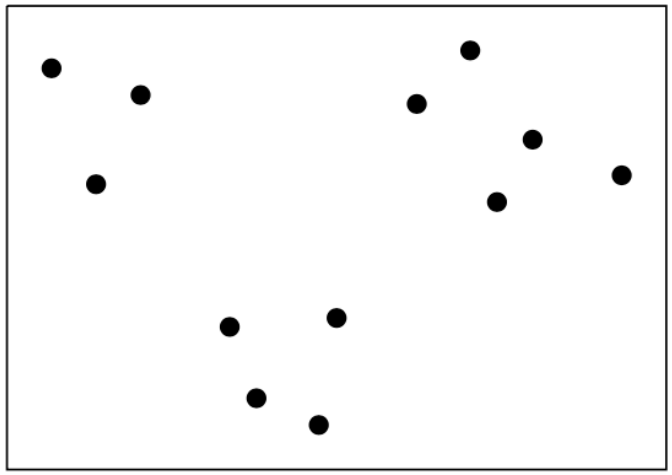
```
whilelem ( t; t ε T )
{
  if ( X[t.i] > X[t.j] )
    swap ( X[t.i], X[t.j] )
}
```

T being a Linked List Reservoir

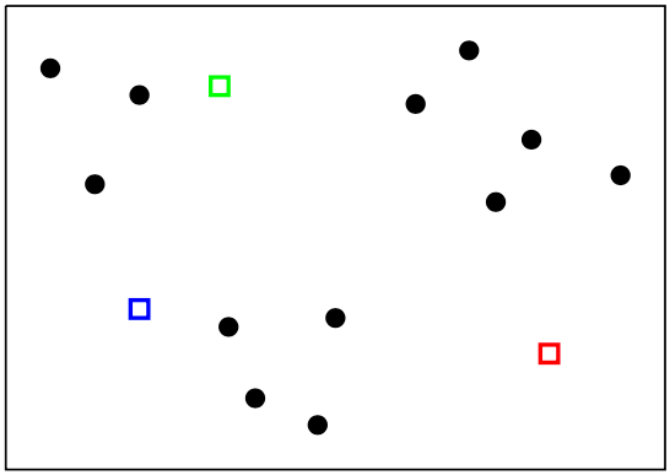
Arriving at a tUPL specification through Deconstruction

k-Means Clustering

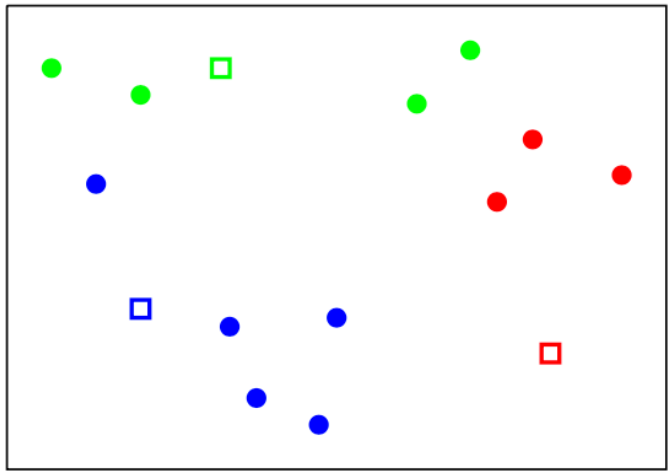
Input data



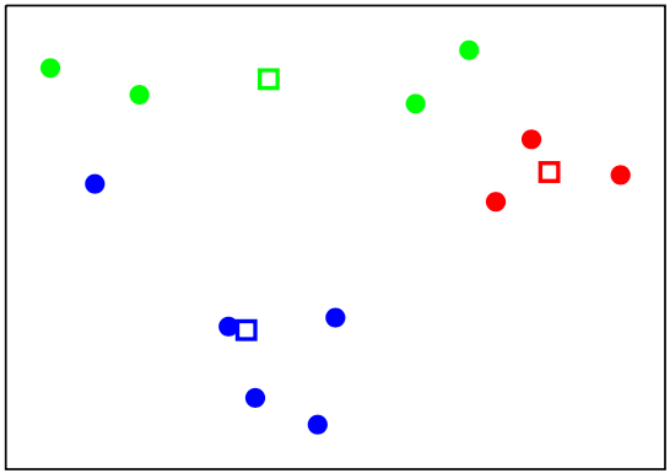
Initialization



It. 1 - Assign the data points

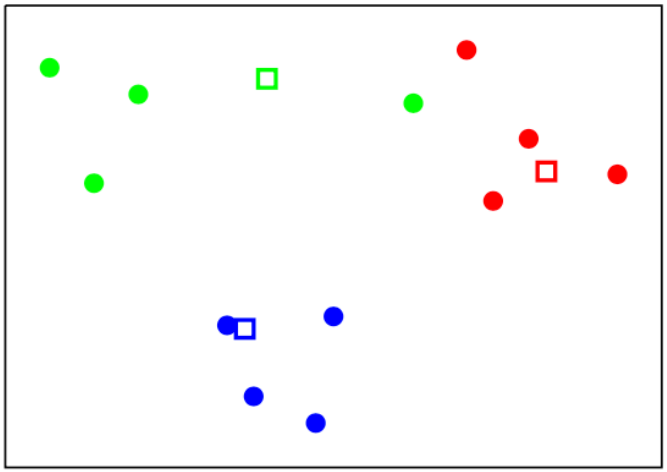


It. 1 - Calculate the means

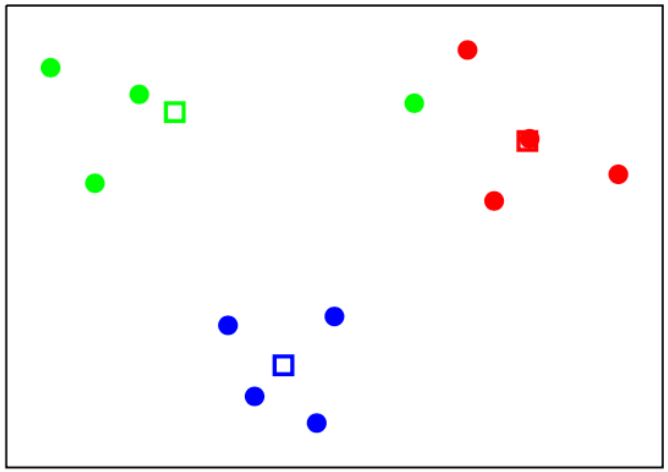


k-Means Clustering

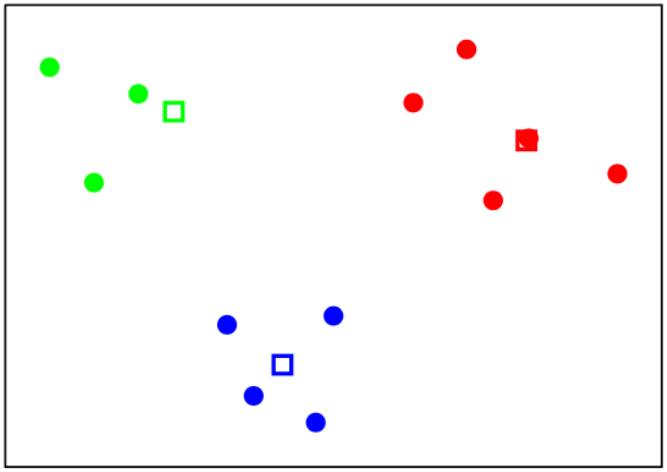
It. 2 - Assign the data points



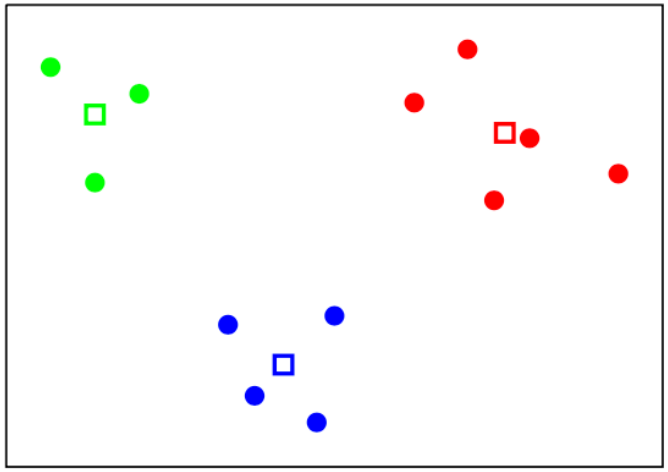
It. 2 - Calculate the means



It. 3 - Assign the data points



It. 3 - Calculate the means



Classic k-Means Algorithm

```
change = true
while (change) {
    change = false

    for ( x = 1 to N ) {
        mindist = LARGE
        for ( m = 1 to k )
            if (dist(DATA[x],VAL[m])
                <= mindist) {
                a = m
                mindist = dist(DATA[x],VAL[m])
            }
        if (a != M[x]) {
            M[x] = a
            change = true
        }
    }
    if (change) {
        for ( m = 1 to k ) {
            mean,count = 0
            for ( x = 1 to N )
                if (M[x] == m) {
                    mean = mean + DATA[x]
                    count = count + 1
                }
            VAL[m] = mean / count
        }
    }
}
```

Deconstructing k-Means

```
change = true
while (change) {
    change = false

    for ( x = 1 to N ) {
        mindist = LARGE
        for ( m = 1 to k )
            if (dist(DATA[x],VAL[m])
                <= mindist) {
                a = m
                mindist = dist(DATA[x],VAL[m])
            }
        if (a != M[x]) {
            M[x] = a
            change = true
        }
    }
    if (change) {
        for ( m = 1 to k ) {
            mean,count = 0
            for ( x = 1 to N )
                if (M[x] == m) {
                    mean = mean + DATA[x]
                    count = count + 1
                }
            VAL[m] = mean / count
        }
    }
}
```

Deconstructing k-Means

```
change = true
while (change) {
    change = false

    for ( x = 1 to N ) {
        mindist = LARGE
        for ( m = 1 to k )
            if (dist(DATA[x],VAL[m])
                <= mindist) {
                a = m
                mindist = dist(DATA[x],VAL[m])
            }
        if (a != M[x]) {
            M[x] = a
            change = true
        }
    }
    if (change) {
        for ( m = 1 to k ) {
            mean,count = 0
            for ( x = 1 to N )
                if (M[x] == m) {
                    mean = mean + DATA[x]
                    count = count + 1
                }
            VAL[m] = mean / count
        }
    }
}
```

```
whilelem (                ) {
                                    }
```

Deconstructing k-Means

```
change = true
while (change) {
    change = false

    for ( x = 1 to N ) {
        mindist = LARGE
        for ( m = 1 to k )
            if (dist(DATA[x],VAL[m])
                <= mindist) {
                a = m
                mindist = dist(DATA[x],VAL[m])
            }
        if (a != M[x]) {
            M[x] = a
            change = true
        }
    }
    if (change) {
        for ( m = 1 to k ) {
            mean,count = 0
            for ( x = 1 to N )
                if (M[x] == m) {
                    mean = mean + DATA[x]
                    count = count + 1
                }
            VAL[m] = mean / count
        }
    }
}
```

```
whilelem (                ) {
                                }
```

Deconstructing k-Means

```
change = true
while (change) {
    change = false

    for ( x = 1 to N ) {
        mindist = LARGE
        for ( m = 1 to k )
            if (dist(DATA[x],VAL[m])
                  <= mindist) {
                a = m
                mindist = dist(DATA[x],VAL[m])
            }
        if (a != M[x]) {
            M[x] = a
            change = true
        }
    }
    if (change) {
        for ( m = 1 to k ) {
            mean,count = 0
            for ( x = 1 to N )
                if (M[x] == m) {
                    mean = mean + DATA[x]
                    count = count + 1
                }
            VAL[m] = mean / count
        }
    }
}
```

```
T = {<m,x> | 1 <= m < k, 1 <= x <= N}

whilelem ( <m,x> ε T ) {

}
```

Deconstructing k-Means

```
change = true
while (change) {
    change = false

    for ( x = 1 to N ) {
        mindist = LARGE
        for ( m = 1 to k )
            if (dist(DATA[x],VAL[m])
                  <= mindist) {
                a = m
                mindist = dist(DATA[x],VAL[m])
            }
        if (a != M[x]) {
            M[x] = a
            change = true
        }
    }
    if (change) {
        for ( m = 1 to k ) {
            mean,count = 0
            for ( x = 1 to N )
                if (M[x] == m) {
                    mean = mean + DATA[x]
                    count = count + 1
                }
            VAL[m] = mean / count
        }
    }
}
```

```
T = {<m,x> | 1 <= m < k, 1 <= x <= N}

whilelem ( <m,x> ε T ) {

                                    }
```

Deconstructing k-Means

```
change = true
while (change) {
    change = false

    for ( x = 1 to N ) {
        mindist = LARGE
        for ( m = 1 to k )
            if (dist(DATA[x],VAL[m])
                  <= mindist) {
                a = m
                mindist = dist(DATA[x],VAL[m])
            }
        if (a != M[x]) {
            M[x] = a
            change = true
        }
    }
    if (change) {
        for ( m = 1 to k ) {
            mean,count = 0
            for ( x = 1 to N )
                if (M[x] == m) {
                    mean = mean + DATA[x]
                    count = count + 1
                }
            VAL[m] = mean / count
        }
    }
}
```

```
T = {<m,x> | 1 <= m < k, 1 <= x <= N}

whilelem ( <m,x> ε T ) {
  if (M[x] != m &&
      dist(DATA[x],VAL[m])
      < dist(DATA[x],VAL[M[x]])) {

    M[x] = m
  }
}
```

Deconstructing k-Means

```
change = true
while (change) {
    change = false

    for ( x = 1 to N ) {
        mindist = LARGE
        for ( m = 1 to k )
            if (dist(DATA[x],VAL[m])
                <= mindist) {
                a = m
                mindist = dist(DATA[x],VAL[m])
            }
        if (a != M[x]) {
            M[x] = a
            change = true
        }
    }
    if (change) {
        for ( m = 1 to k ) {
            mean,count = 0
            for ( x = 1 to N )
                if (M[x] == m) {
                    mean = mean + DATA[x]
                    count = count + 1
                }
            VAL[m] = mean / count
        }
    }
}
```

```
T = {<m,x> | 1 <= m < k, 1 <= x <= N}

whilelem ( <m,x> ε T ) {
  if (M[x] != m &&
      dist(DATA[x],VAL[m])
      < dist(DATA[x],VAL[M[x]])) {

    M[x] = m
  }
}
```

Deconstructing k-Means

```
change = true
while (change) {
    change = false

    for ( x = 1 to N ) {
        mindist = LARGE
        for ( m = 1 to k )
            if (dist(DATA[x],VAL[m])
                  <= mindist) {
                a = m
                mindist = dist(DATA[x],VAL[m])
            }
        if (a != M[x]) {
            M[x] = a
            change = true
        }
    }
    if (change) {
        for ( m = 1 to k ) {
            mean,count = 0
            for ( x = 1 to N )
                if (M[x] == m) {
                    mean = mean + DATA[x]
                    count = count + 1
                }
            VAL[m] = mean / count
        }
    }
}
```

```
T = {<m,x> | 1 <= m < k, 1 <= x <= N}

whilelem ( <m,x> ε T ) {
  if (M[x] != m &&
      dist(DATA[x],VAL[m])
      < dist(DATA[x],VAL[M[x]])) {
    VAL[M[x]] = (VAL[M[x]]*SIZE[M[x]]
                 - DATA[x])
                 /(SIZE[M[x]] - 1)
    SIZE[M[x]] -= 1
    VAL[m] = (VAL[m]*SIZE[m]+DATA[x])
              /(SIZE[m] + 1)
    SIZE[m] += 1
    M[x] = m
  }
}
```

Final k-Means tUPL Specification

```
T = { <m,x> | 1 <= m < k, 1 <= x <= N}

whilelem ( <m,x> ε T ) {
  if (M[x] != m &&
      dist(DATA[x],VAL[m])
      < dist(DATA[x],VAL[M[x]])) {
    VAL[M[x]] = (VAL[M[x]]*SIZE[M[x]]
                 - DATA[x])/(SIZE[M[x]] - 1)
    SIZE[M[x]] -= 1
    VAL[m] = (VAL[m]*SIZE[m]+DATA[x])
             /(SIZE[m] + 1)
    SIZE[m] += 1
    M[x] = m
  }
}
```

PageRank tUPL Specification

```
E = { <u,v> | (u,v) ε G(E) }

whilelem ( <u,v> ε E ) {
    if (PR[u] != OLD[u,v]) {
        PR[v] += d *(PR[u]-OLD[u,v]) / Dout[u]
        OLD[u,v] = PR[u]
    }
}
```

$$PR(v) = \frac{1-d}{|V|} + d \sum_{u \in \text{nbh}^-(v)} \frac{PR(u)}{\deg^+(u)}$$

```
while (PR != PRold) {
  PRold = PR
  for (v = 1 to N) {
    PR[v] = (1-d)/N
    for (u = 1 to N)
      if (edge (u, v) exists)
        PR[v] += d∗PRold[u]/Dout[u]
  }
}
```

MaxFlow tUPL specification

T = { <**u**,**v**,**w**> | (**u**,**v**) and (**v**,**w**) (back)edges of G and **w!=u** }*

```
whilelem ( t; t ε T )
{    if (Delta[t.u,t.v] > 0 && Remainder[t.v,t.w] > 0)
     {
          delta_change = min(Remainder[t.v,t.w],Delta[t.u,t.v]);
          Delta[t.v,t.w]+= delta_change;
          Remainder[t.v,t.w] -= delta_change;
          Remainder[t.w,t.v] += delta_change;
          F[t.u,t.v] += delta_change;
          Delta[t.u,t.v] -= delta_change
     }
     if (Delta[t.u,t.v] > 0 && Remainder[t.v,t.w] == 0)
     {
          if (t.v == 's' || t.v == 't')
          {
               F[t.u,t.v] += Delta[t.u,t.v];
               Delta[t.u,t.v] = 0
          }
          else
          {    # Reverse Flow
               Delta[t.v,t.u] += Delta[t.u,t.v];
               Remainder[t.v,t.u]-= Delta[t.u,t.v];
               Delta[t.u,t.v] = 0
          }
     }
}
```

*|**T**| ≈ (aver_out+aver_in)*(aver_out+aver_in-1)*|V|
≈ aver_out^4*|V|

MaxFlow even more Deconstructed

```
T = { <u,v> | (u,v) ε E or (v,u) ε E }

whilelem ( <u,v> ε T) {
    if (Remainder[u,v] > 0 && Overflow[u] > 0) {
      flow = min(Remainder[v,u], Overflow[u])
      Remainder[u,v] -= flow
      Remainder[v,u] += flow
      Overflow[u] -= flow
      Overflow[v] += flow
    }
}
```

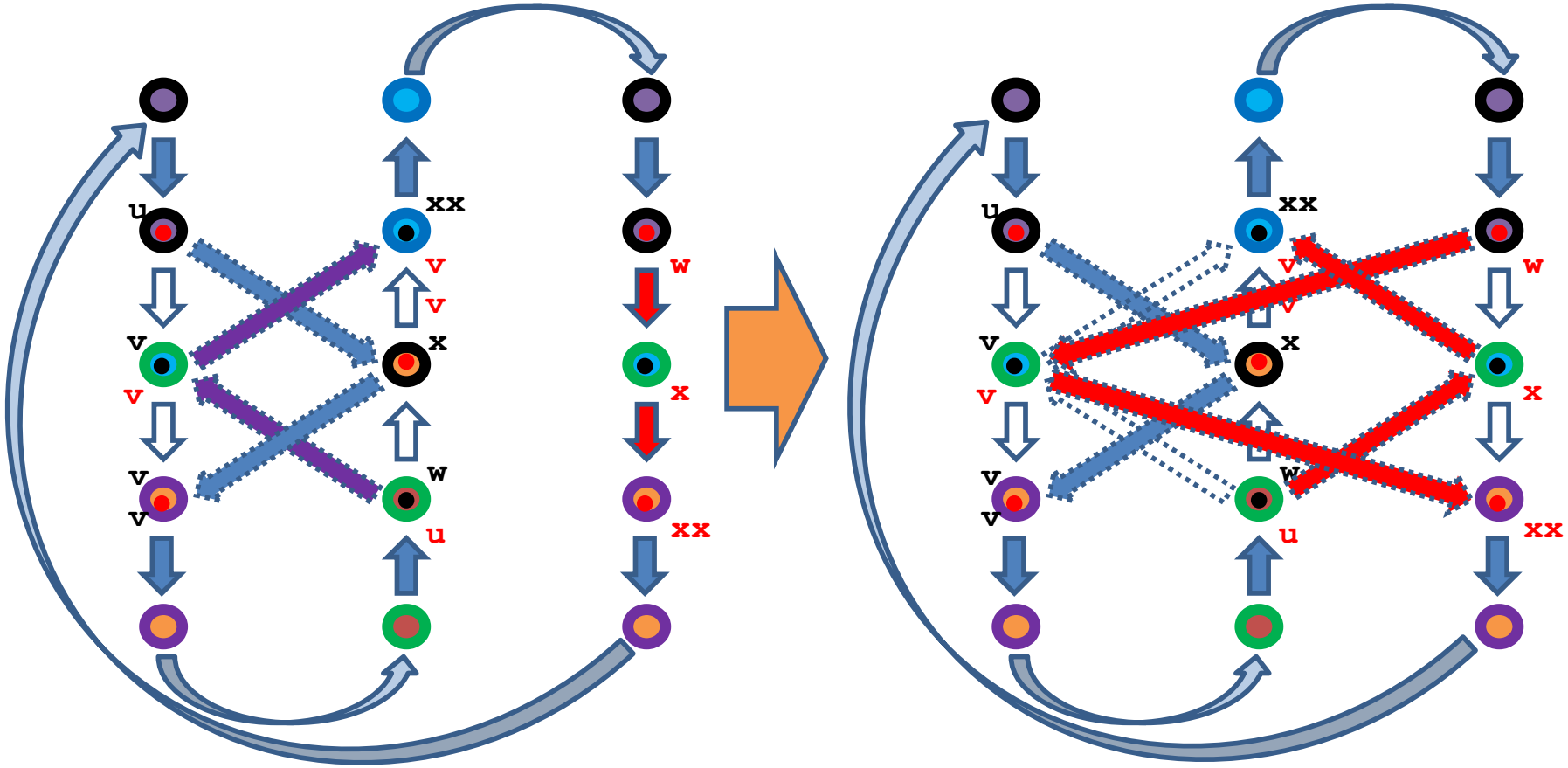
Strongly Connected Components
tUPL Specification

```
T = { <u,v> | (u,v) ε G(E) }

forelem ( <u,v> ε T ) {
   X[v] = {v}
}

whilelem ( <u,v> ε T ) {
   X[v] = X[v] U X[u]
}
```

(3-OPT) Traveling Salesman Problem tUPL Specification

```
R = {<u,v,vv,w,x,xx> |  (u,v),(x,xx),(v,vv),(w,x),
                        (u,x),(v,xx),(w,v),(x,vv)  ε G(E)}

whilelem {<u,v,vv,w,x,xx> ε R
 if ( In_TSP[u,v]==True && In_TSP[x,xx]==True &&
      In_TSP[v,vv]==True && In_TSP[w,x]==True )
    In_TSP[u,v]=False
    In_TSP[x,xx]=False
    In_TSP[v,vv]=False
    In_TSP[w,x]=False
    In_TSP[u,x]=True
    In_TSP[v,xx]=True
    In_TSP[w,v]=True
    In_TSP[x,vv]=True
    W = W - (W[u,v]+W[x,xx]+W[v,vv]+W[w,x] -
             (W[u,x]+W[v,xx]+W[w,v]+W[x,vv]))
```

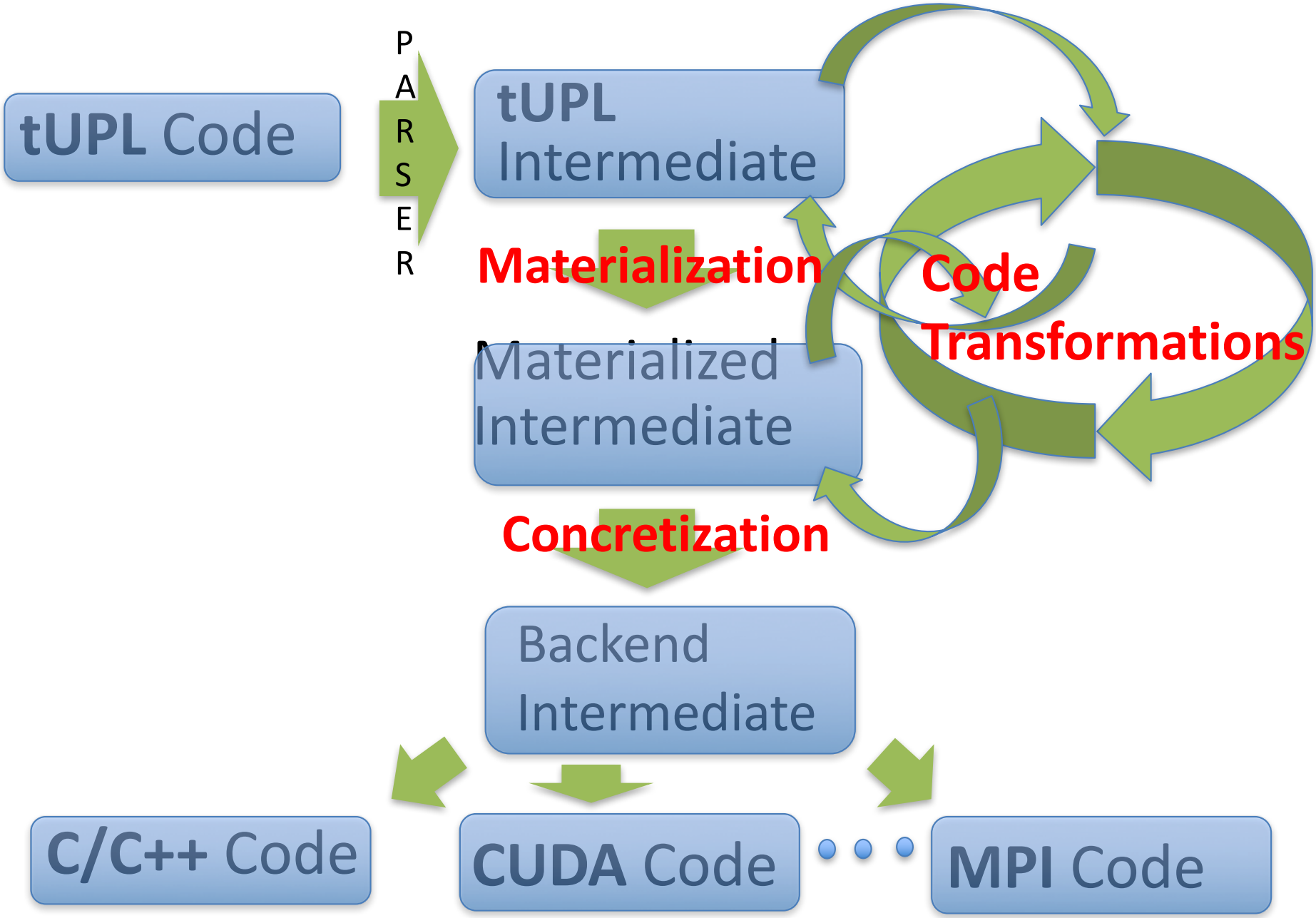
2 Concatenated, Concatenated 2-OPTs == Concatenated 3 OPT



Replace Edges (u,v),(x,xx) by (u,x),(v,xx)
Replace Edges (v,vv),(w,x) by (w,v),(x,vv)

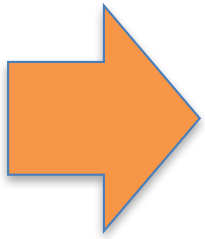
Data Structure Creation

Automatic Data Structure Generation in **tUPL**



tUPL Intermediate

```
forelem ( t; t ε T )
{
    … t …
}
whilelem ( t; t ε T )
{
    … t …
}
```



```
forelem ( i; i ε pT )
{
    … T[i] …
}
whilelem ( i; i ε pT )
{
    … T[i] …
}
```

- pT and T[i] notation allows for a more clear expression of the materialization and concretization phase
- tUPL allows mix use of **tUPL** notation and intermediate notation

Some Code Transformations*

Orthogonalization

```
forelem (i; i ε pA)
    … A[i]…
```

```
forelem (ii; ii ε A.field1)
    forelem (i; i ε pA.field1[ii])
        … A[i]…
```

`A.field1` is the set of all possible field1 values of tuples in A: { i.field1 | i ε A }

Encapsulation

```
forelem (i; i ε pA.field1)
    … …
```

```
forelem (i; i ε N10)
    … …
```

If `A.field1` would be { 0, 1, 3, 4, 7, 9, 10 }, for instance. This transformation only makes sense, if the execution of the inner loop for the other i-value's results into a NOP. i.e. C[i] = C[i] + B[i], and B[i] == 0 for 2, 5, 6 and 8.

forelem** is used in the examples but the trafo's equally apply to **whilelem

Some Code Transformations (2)

Loop Collapse

```
forelem (i; i ε pA)
    forelem (j; j ε pB.field_b[A[i].field_a])
        … A[i].field_c … B[j].field_d …
```

```
forelem (i; i ε pAxB.field_b[field_a])
    … AxB[i].field_c … AxB[i].field_d …
```

AxB is the cross product of the two tuple sets A and B: { < i, j > | i ε A and j ε B }

Some Code Transformations (3)

Loop Interchange

```
forelem (i; i ε pA)
    forelem (j; j ε pB)
        … A[i] … B[j] …
```

```
forelem (j; j ε pB)
    forelem (i; i ε pA)
        … A[i] … B[j] …
```

Horizontal Iteration Space Reduction

```
forelem (i; i ε pA)
    … A[i].field2 … A[i].field3 …
```

```
forelem (i; i ε pA')
    … A'[i].field2 … A'[i].field3 …
```

With A' = { <field2,field3> | <field1,field2,field3> ε A }

Materialization

```
forelem (i; i ε pA.field[X])
    … A[i]…
```

```
forelem (i; i ε N*)
    … PA[i]…
```

N* represents the set { 1, 2, … , |PA| }, with PA an enumeration of the set:

{ i | i ε A and i.field == X }

DO NOT CONFUSE PA with a linear array data structure

Some more code transformations

Tuple Splitting

```
forelem (i; i ε A.field)
    forelem (k; k ε pB.field[i])
        … B[k].value …
```

⇨

```
forelem (i; i ε N10)
    forelem (k; k ε pB.field[i])
        … B[k].value …
```

⇨

```
forelem (i; i ε N10)
    forelem (k; k ε N*)
        … B[i][k].value …
```

⇨

```
forelem (i; i ε N10)
    forelem (k; k ε N*)
        … B[i].value[k] …
```

2 dimensional materialization into B[][] necessary because of outerloop dependence.

Some more code transformations (2)

N* Materialization

```
forelem (i; i ε N10)
    forelem (k; k ε N*)
        … A[i][k] …
```

```
forelem (i; i ε N10)
    forelem (k; k ε PA_len[i])
        … A[i][k] …
```

Some more code transformations (3)

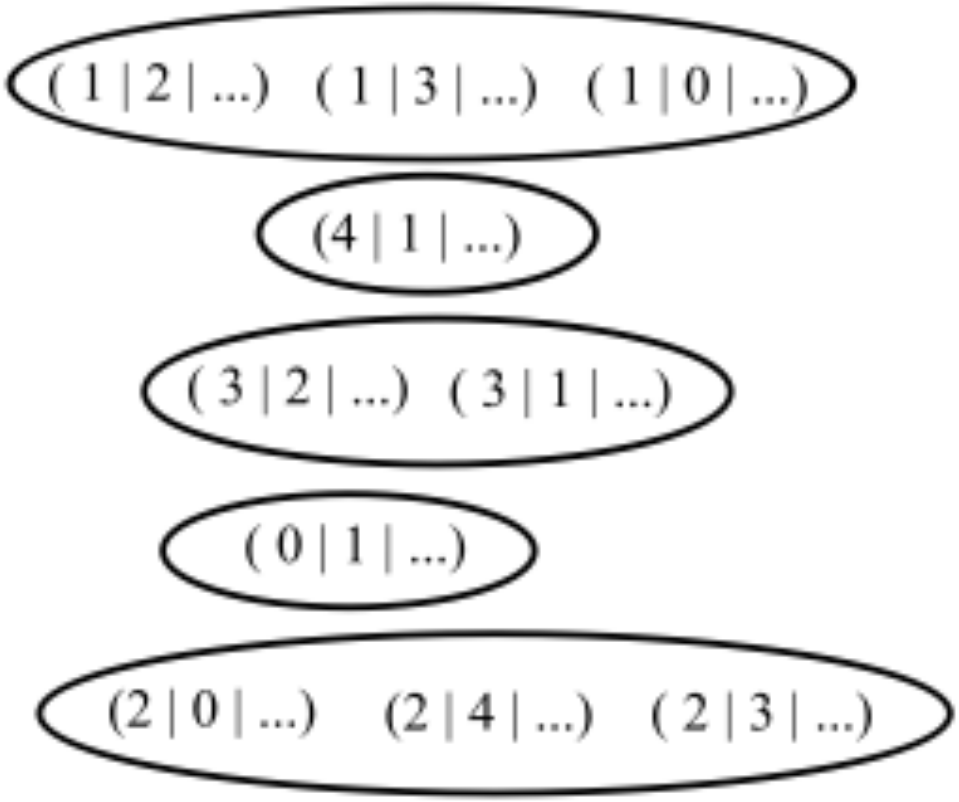
Data Localization

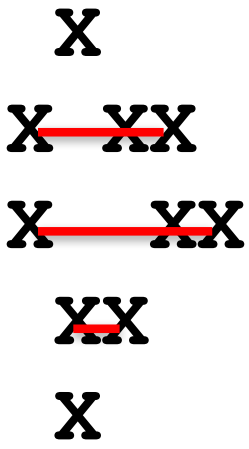
```
forelem (i; i ε pA)
   … B [ A[i] ] …
```

```
forelem (i; i ε pA')
   … A'[i].field_B …
```

Here the tuples in reservoir A are being extended to include the data at address @`B[A[i].field_k}`. So A' = { < t, B[t] > | t ε A }. By default, this transformation is only allowed for read only data at B.

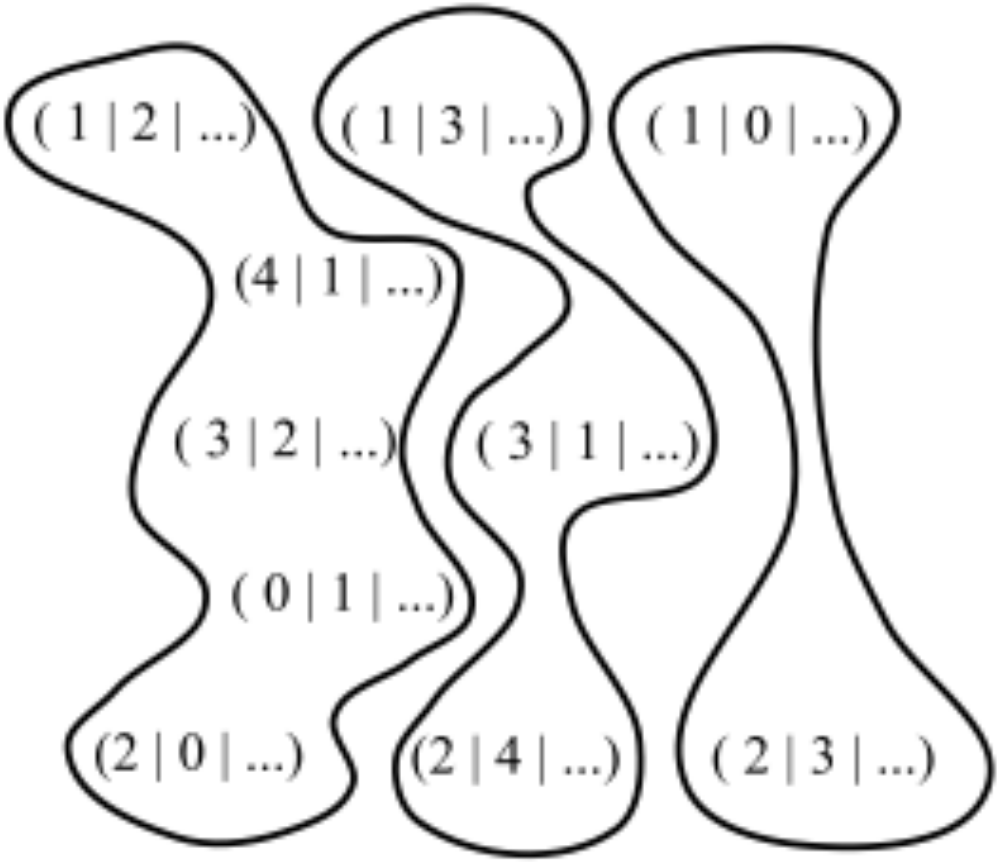
Regrouping of Single Storage Locations (Tuples)

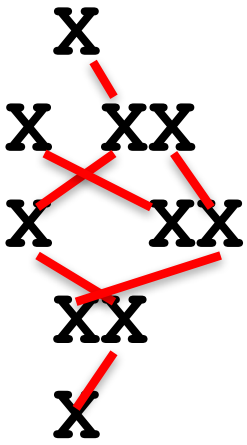




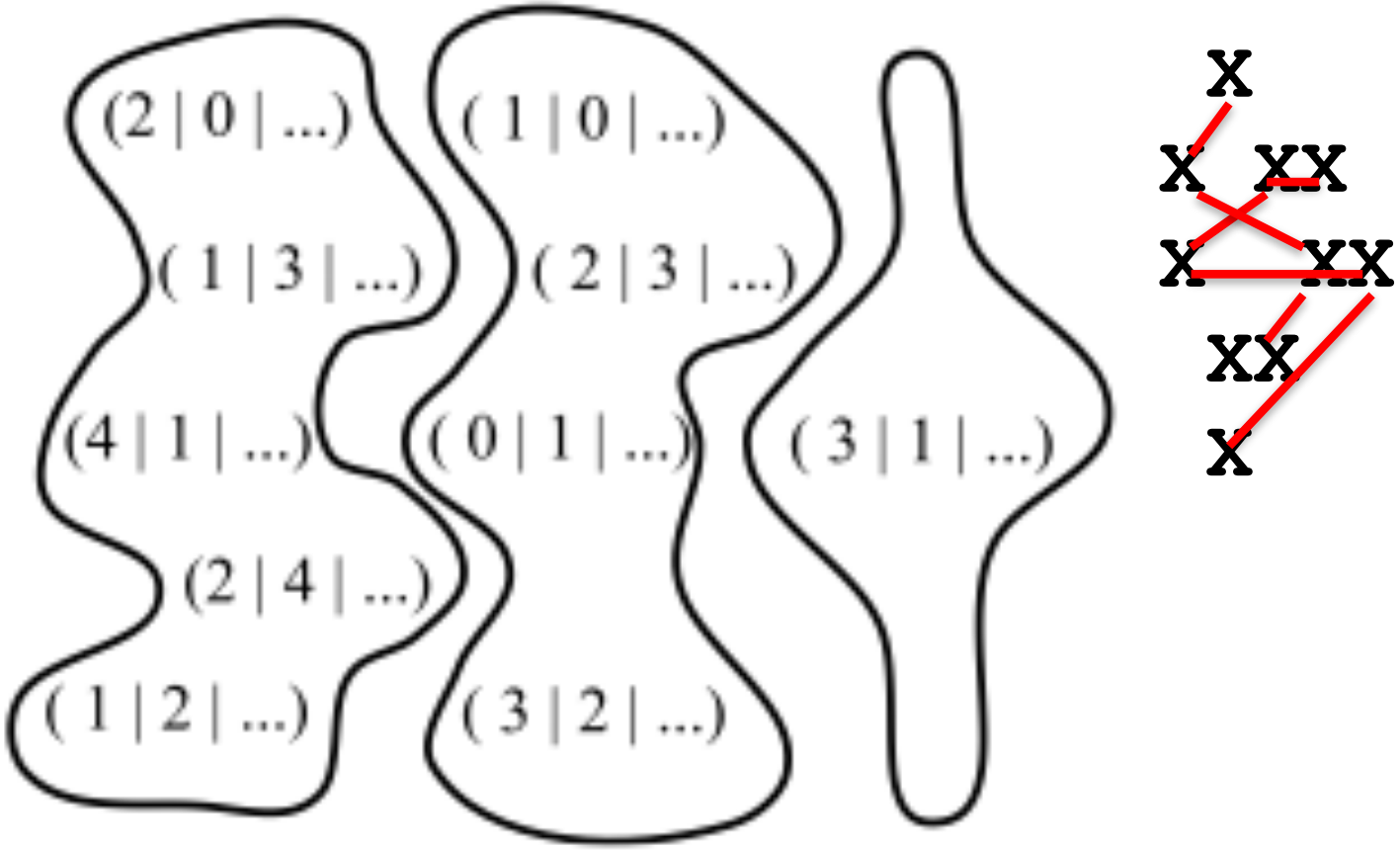
Regrouping as a result of **orthogonalization** on the first field

Regrouping after **Materialization** and **Loop Interchange**





Regrouping after **orthogonalization** on the second field followed by **materialization** and **loop interchange**



Concretization

```
forelem (i; i ε N*)
    … PA[i]…
```

```
forelem (i; i ε PA_len[i])
    … PA[i] …
```

```
for (i = 0; i < PA_len; i++)
    ... PA[i] ...
```

Some Concretization Steps

tUPLE loop construct	Concretization
forelem (i; i ε pA) … A[i]…	Linked list of struct's
forelem (i; i ε $\mathbf{N_{10}}$) … A[i]…	An array of struct's
forelem (i; i ε $\mathbf{N_{10}}$) **forelem** (k; k ε PA_len[i]) … A[i][k] …	An array of arrays of struct's
forelem (i; i ε $\mathbf{N_{10}}$) **forelem** (k; k ε PA_len[i]) … A[i][k].value …	An array of arrays of struct's
forelem (i; i ε $\mathbf{N_{10}}$) **forelem** (k; k ε PA_len[i]) … A[i].value[k] …	An array of arrays of values

Example

```
forelem (i;iε pA)
    … B[A[i]]…
```

A linked list of struct's: A +
A multidimensional array: B

Data Localization

```
forelem (i;iε pA')
    … A'[i].field_B …
```

An linked list of struct's: A

Materialization

```
forelem (i;iε PA'_len)
    … PA'[i].field_B …
```

An array of struct's A'

Tuple Splitting

```
forelem (i;iε pA'_len)
    … PA'.field_B[i]…
```

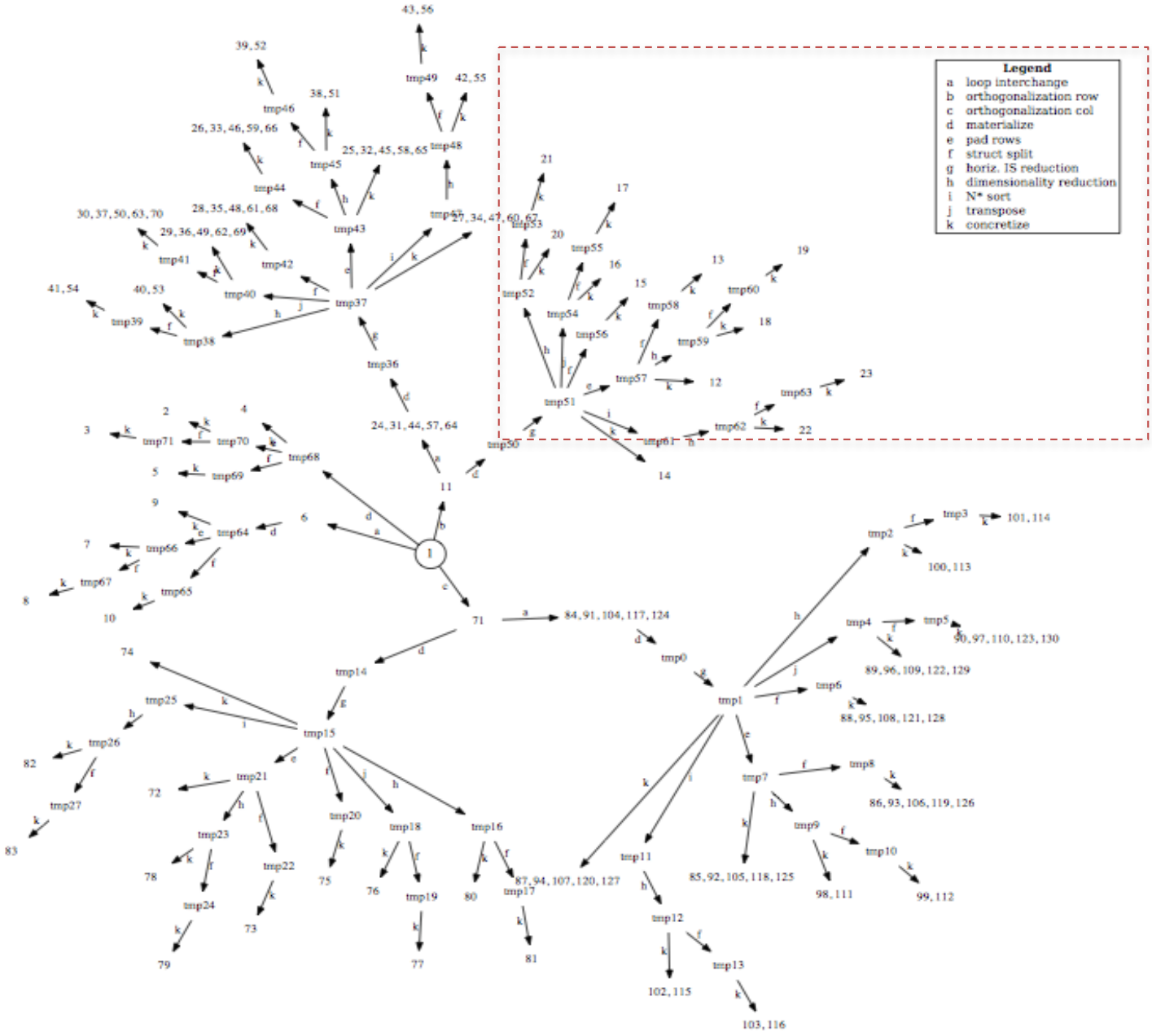
Several Arrays for each field of A'

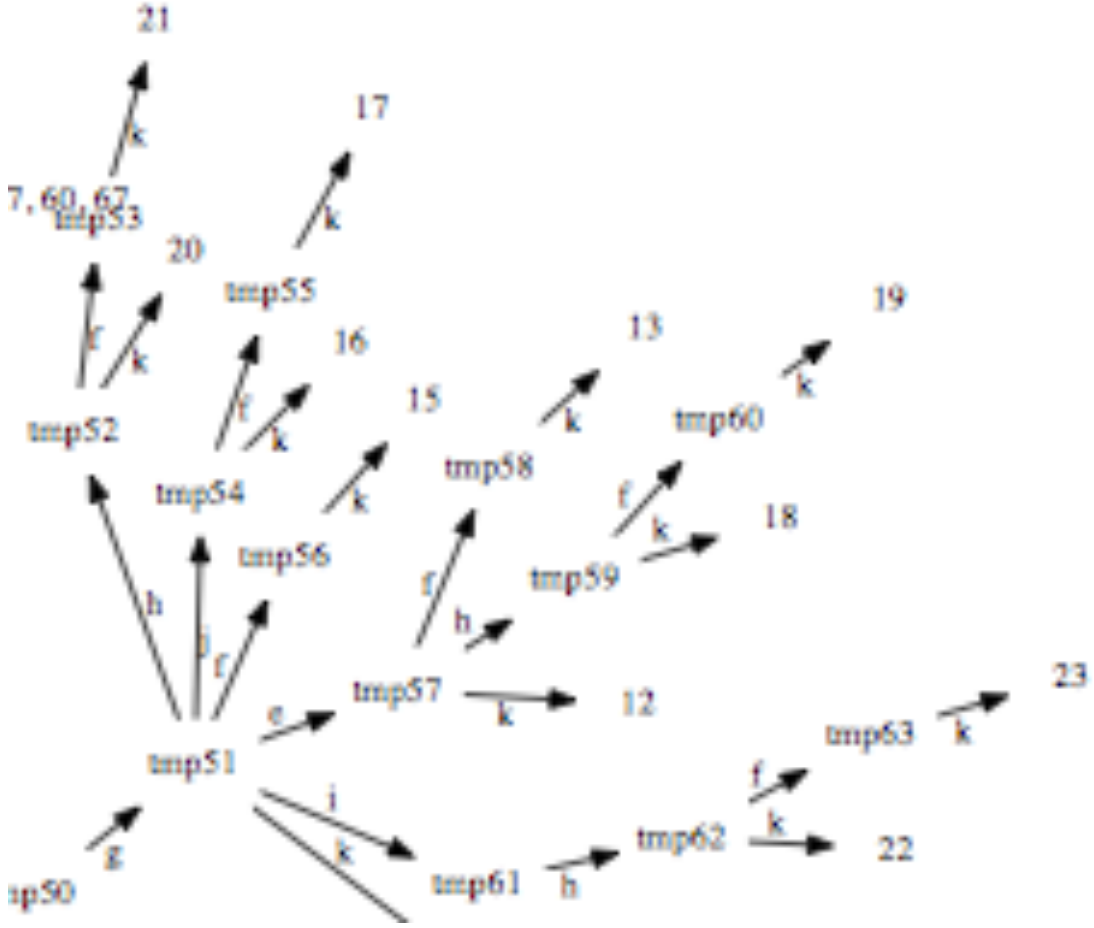
Horizontal Iteration Space Reduction

```
forelem (i;iε pA'_len)
    … PA'.field_B[i]…
```

Just one array of field_B values

The Transformation Search Space for SpMxM





Legend

a	loop interchange
b	orthogonalization row
c	orthogonalization col
d	materialize
e	pad rows
f	struct split
g	horiz. IS reduction
h	dimensionality reduction
i	N* sort
j	transpose
k	concretize

Algorithm Creation

Chaining

- **tUPL** will automatically choose sequences of valid serial codes to be executed one after the other, so that their execution is being optimized.
- So, next to the automatic generation of data structures **tUPL** will also automatically optimize and change the order in which operations are performed and by doing so will change the actual algorithm being used to compute the results.
- These sequences are being identified as chains of pairs of tuples and serial codes:

  $(t_k, \texttt{Serial_Code_i})^*$

  representing

  $\texttt{Serial_Code_i}\ (< t_k >)$

*Note that `Cond_i` has to evaluate to true for every t_k

Recap

tUPL Loop Body:

```
if ( Cond_1 )
{
   Serial_Code_1 (< t >)
}
if ( Cond_2 )
{
   Serial_Code_2 (< t >)
}
…
if ( Cond_n )
{
   Serial_Code_n (< t >)
}
```

Different kind of chains

- Mono Chains (MC), every element in the chain has the same serial code:

  $(t_1, \texttt{Serial_Code_i}), (t_2, \texttt{Serial_Code_i}), \ldots$

- Two Typed Chains:
  - Alternating Chains (AC), consecutive elements in the chain alternate between `Serial_Code_i` and `Serial_Code_j`
  - Cascading Chains (CC), first part of the chain uses `Serial_Code_i` the second part of the chain uses `Serial_Code_j`

    $(t_1, \texttt{Serial_Code_i}), (t_2, \texttt{Serial_Code_i}), \ldots,$
    $(t_k, \texttt{Serial_Code_j}), (t_{k+1}, \texttt{Serial_Code_j}), \ldots$

- Hybrid Chains (HC)

Profitable Chain

A chain C is profitable* iff

- The consecutive execution of the elements in C can be optimized such that the execution time of the whole chain is less than the sum of the execution times of the individual elements
- AND the chain is minimal in such a way that the chain C cannot be broken into smaller chains C_1 and C_2 such that $C = C_1 \,||\, C_2$ and

$$\text{Exec}(C) = \text{Exec}(C_1) + \text{Exec}(C_2)$$

* C is being referred to as a profit chain

Profit Chains in SpMxV

```
forelem ( t; t ε T )
      {
         Value_C[t.i]+= Value_A[t.i,t.j]
                        * Value_B[t.j]
      }
```

(<1,1>, Serial_Code_1), (<1,2>, Serial_Code_1), … can be optimized such that subsequent reads of Value_C[t.i] are eliminated. So these chains are identified as profit chains.

Covering Chain Set

A covering chain set CCS is a set of Chains C_i such that for every tuple (t_k, Serial_Code_i) there is an i such that

(t_k, Serial_Code_i) ε C_i

Note that if the possible set of profit chains is not covering then this set can be completed with single (non-profit) chains, consisting out of the (t_k, Serial_Code_i) pairs which were not covered, to obtain a covering chain set.

Transforming Execution

If

```
whilelem ( t; t ε T )
```

is just scheduled, then if

```
whilelem ( C; C ε CCS )
  forelem ( t; t ε C )
```

is also just scheduled, then both loop structures are semantically equivalent.

Examples of profit chains I

```
whilelem ( t; t ε T )
      {
         if ( X[t.i] > X[t.j] )
            swap ( X[t.i], X[t.j] )
      }
```

(<1,2>, Serial_Code_1),
(<2,3>, Serial_Code_1),
(<3,4>, Serial_Code_1),…, (<n-1,n>, Serial_Code_1)
with X[1]>X[2], X[2]>X[3], etc, results in a sequence of n swaps, whereas it can be optimized by executing just one insert!!!

Example of profit chains II

```
whilelem ( t; t ε T )
    {   if (Delta[t.u,t.v] > 0 && Remainder[t.v,t.w] > 0)
        {
            delta_change = min(Remainder[t.v,t.w],Delta[t.u,t.v]);
Serial_Code_1   Delta[t.v,t.w]+= delta_change;
            Remainder[t.v,t.w] -= delta_change;
            Remainder[t.w,t.v] += delta_change;
            F[t.u,t.v] += delta_change;
            Delta[t.u,t.v] -= delta_change
        }
        if (Delta[t.u,t.v] > 0 && Remainder[t.v,t.w] == 0)
        {
            …
            else
            {   # Reverse Flow
                Delta[t.v,t.u] += Delta[t.u,t.v];
Serial_Code_2   Remainder[t.v,t.u]-= Delta[t.u,t.v];
                Delta[t.u,t.v] = 0
            }
        }
    }
```

Then (<s,4,6>,Serial_Code_1), (<4,6,52>, Serial_Code_1),…,(<100,105,107>, Serial_Code_1), (<105, 107,111>, Serial_Code_2), (<111,107, 105>, Serial_Code_1), … (<6,4,s>, Serial_Code_1) with `Remainder[4,6]>0`, with `Remainder[6,52]>0`,… etc. , and `Remainder[107,111]==0` is a profit chain.

As well as

(<s,4,6>,Serial_Code_1), (<4,6,52>, Serial_Code_1),…,(<100,105,107>, Serial_Code_1), (<105, 107, t>, Serial_Code_1), with `Remainder[4,6]>0`, with `Remainder[6,52]>0`,… etc.

Note that the latter profit chain is in fact the augmented path as defined by Ford and Fulkerson!!!

More detailed: Maxflow Candidate ladder

```
if Delta[t.u, t.v] > 0 and Rem[t.v, t.w] > 0:
   if Rem[t.v, t.w] <= Delta[t.u, t.v]:
      delta_change = Rem[t.v, t.w]
   else:
      delta_change = Delta[t.u, t.v]
   Delta[t.v, t.w] = Delta[t.v, t.w] + delta_change
   Rem[t.v, t.w] = Rem[t.v, t.w] - delta_change
   Rem[t.w, t.v] = Rem[t.w, t.v] + delta_change
   if F[t.v, t.u] >= delta_change:
      F[t.v, t.u] = F[t.v, t.u] - delta_change
   else:
      F[t.u, t.v] = F[t.u, t.v] + delta_change
   Delta[t.u, t.v] = Delta[t.u, t.v] - delta_change
if Delta[t.v, t.w] > 0 and Rem[t.w, t.x] > 0:
   if Rem[t.w, t.x] <= Delta[t.v, t.w]:
      delta_change = Rem[t.w, t.x]
   else:
      delta_change = Delta[t.v, t.w]
   Delta[t.w, t.x] = Delta[t.w, t.x] + delta_change
   Rem[t.w, t.x] = Rem[t.w, t.x] - delta_change
   Rem[t.x, t.w] = Rem[t.x, t.w] + delta_change
   if F[t.w, t.v] >= delta_change:
      F[t.w, t.v] = F[t.w, t.v] - delta_change
   else:
      F[t.v, t.w] = F[t.v, t.w] + delta_chang]
   Delta[t.v, t.w] = Delta[t.v, t.w] - delta_change
if Delta[t.w, t.x] > 0 and Rem[t.x, t.y] > 0:
   if Rem[t.x, t.y] <= Delta[t.w, t.x]:
      delta_change = Rem[t.x, t.y]
   else:
      delta_change = Delta[t.w, t.x]
   Delta[t.x, t.y] = Delta[t.x, t.y] + delta_change
   Rem[t.x, t.y] = Rem[t.x, t.y] - delta_change
   Rem[t.y, t.x] = Rem[t.y, t.x] + delta_change
   if F[t.x, t.w] >= delta_change:
      F[t.x, t.w] = F[t.x, t.w] - delta_change
   else:
      F[t.w, t.x] = F[t.w, t.x] + delta_change
   Delta[t.w, t.x] = Delta[t.w, t.x] - delta_change
```

```
if Delta[t.w, t.x] > 0 and Rem[t.x, t.w] > 0:
   if Rem[t.x, t.w] <= Delta[t.w, t.x]:
      delta_change = Rem[t.x, t.w]
   else:
      delta_change = Delta[t.w, t.x]
   Delta[t.x, t.w] = Delta[t.x, t.w] + delta_change
   Rem[t.x, t.w] = Rem[t.x, t.w] - delta_change
   Rem[t.w, t.x] = Rem[t.w, t.x] + delta_change
   if F[t.x, t.w] >= delta_change:
      F[t.x, t.w] = F[t.x, t.w] - delta_change
   else:
      F[t.w, t.x] = F[t.w, t.x] + delta_change
   Delta[t.w, t.x] = Delta[t.w, t.x] - delta_change
```

```
if Delta[t.v, t.u] > 0 and Rem[t.u, t.e] > 0:
   if Rem[t.u, t.e] <= Delta[t.v, t.u]:
      delta_change = Rem[t.u, t.e]
   else:
      delta_change = Delta[t.v, t.u]
   Delta[t.u, t.e] = Delta[t.u, t.e] + delta_change
   Rem[t.u, t.e] = Rem[t.u, t.e] - delta_change
   Rem[t.e, t.u] = Rem[t.e, t.u] + delta_change
   if F[t.u, t.v] >= delta_change:
      F[t.u, t.v] = F[t.u, t.v] - delta_change
   else:
      F[t.v, t.u] = F[t.v, t.u] + delta_change
   Delta[t.v, t.u] = Delta[t.v, t.u] - delta_change
if Delta[t.w, t.v] > 0 and Rem[t.v, t.u] > 0:
   if Rem[t.v, t.u] <= Delta[t.w, t.v]:
      delta_change = Rem[t.v, t.u]
   else:
      delta_change = Delta[t.w, t.v]
   Delta[t.v, t.u] = Delta[t.v, t.u] + delta_change
   Rem[t.v, t.u] = Rem[t.v, t.u] - delta_change
   Rem[t.u, t.v] = Rem[t.u, t.v] + delta_change
   if F[t.v, t.w] >= delta_change:
      F[t.v, t.w] = F[t.v, t.w] - delta_change
   else:
      F[t.w, t.v] = F[t.w, t.v] + delta_change
   Delta[t.w, t.v] = Delta[t.w, t.v] - delta_change
if Delta[t.x, t.w] > 0 and Rem[t.w, t.v] > 0:
   if Rem[t.w, t.v] <= Delta[t.x, t.w]:
      delta_change = Rem[t.w, t.v]
   else:
      delta_change = Delta[t.x, t.w]
   Delta[t.w, t.v] = Delta[t.w, t.v] + delta_change
   Rem[t.w, t.v] = Rem[t.w, t.v] - delta_change
   Rem[t.v, t.w] = Rem[t.v, t.w] + delta_change
   if F[t.w, t.x] >= delta_change:
      F[t.w, t.x] = F[t.w, t.x] - delta_change
   else:
      F[t.x, t.w] = F[t.x, t.w] + delta_change
   Delta[t.x, t.w] = Delta[t.x, t.w] - delta_change
```

Example of Profit Chains III: Strongly Connected Components

```
T = { <u,v> | (u,v) ε G(E) }

forelem ( <u,v> ε T ) {
   X[v] = {v}
}

whilelem ( <u,v> ε T ) {
   X[v] = X[v] U X[u]
}

if X[u]==X[v] print "u,v"
```

An example **profit chain**:

<1,2>,<2,3>,<3,1>,<1,2>,<2,3>

```
X[2] = X[1] U X[2];
X[3] = X[2] U X[3];
X[1] = X[3] U X[1];
X[2] = X[1] U X[2];
X[3] = X[2] U X[3];
if (X[2]==X[1])
{   print tuple_2};
if (X[3]==X[1])
{   print tuple_3};
```

Forward Substitution

```
X[3] = X[1] U X[2] U X[3];
X[1] = X[3] U X[1];
X[2] = X[1] U X[2];
X[3] = X[1] U X[2] U X[3];
if (X[1] U X[2]==X[1])
{   print tuple_2 };
if (X[3]==X[1])
{   print tuple_3 };
```

Forward Substitution

```
X[1] = X[1] U X[2] U X[3]*;
X[2] = X[1] U X[2];
X[3] = X[1] U X[2] U X[3];
if (X[1] U X[2]==X[1])
{   print tuple_2 };
if (X[1] U X[2] U X[3]==X[1])
{   print tuple_3 };
```

Forward Substitution

*Further use of X[2] (Def-Use analysis) also results in forward substitution

```
X[2] = X[1] U X[2] U X[3];
X[3] = X[1] U X[2] U X[3];
if (X[1] U X[2] U X[3]==
    X[1] U X[2] U X[3])
{   print tuple_2 };
if (X[1] U X[2] U X[3]==
    X[1] U X[2] U X[3])
{   print tuple_3 };
```

Forward Substitution

```
X[3] = X[1] U X[2] U X[3];
if (X[1] U X[2] U X[3]==
    X[1] U X[2] U X[3])
{   print tuple_2 };
if (X[1] U X[2] U X[3]==
    X[1] U X[2] U X[3])
{   print tuple_3 };
```

Forward Substitution

```
if (X[1] U X[2] U X[3]==
    X[1] U X[2] U X[3])
{   print tuple_2 };
if (X[1] U X[2] U X[3]==
    X[1] U X[2] U X[3])
{   print tuple_3 };
```

If Elimination

```
print tuple_2;
print tuple_3;
```

Reduction: From 5 to 0 assignments and elimination of 2 conditional tests !!

The previous Profit Chain eventually leads to an implementation of Tarjan's Algorithm for strongly connected components.

Tarjan's Algorithm is inherently sequential, but using other profit chains or none leads to parallel implementations of Tarjan's Algorithm.

Summarizing

- **tUPL** will determine at “compile time” the possible sequences of tuples in the tuple reservoir (**Candidate Profit Chains**) of (sequential) codes to be executed one after the other, so that their execution is being optimized.
- At runtime the existence of these candidate profit chains are checked, starting with the candidate profit chain with the highest amount of potential profit.
- Whenever a profit chain is detected at runtime the (random) tuple selection in the **`whilelem/forelem`** loop structures is interrupted and replaced by the execution of this profit chain.
- By doing so the actual algorithm being used to compute the results will change **during the computation**.