tUPL Parallel Programming Paradigm

Data Flow Computing

Traditionally, compilers analyze program source code for data dependencies between instructions in order to better organize the instruction sequences in the binary output files.

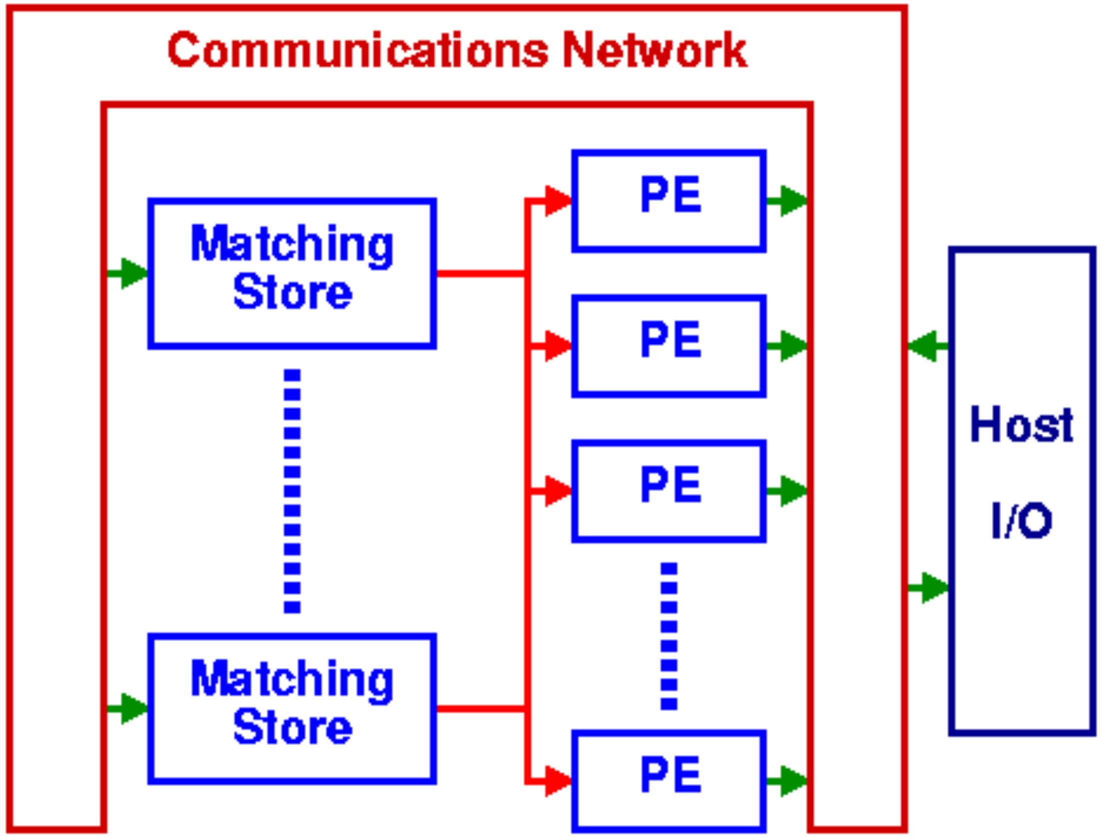
A dataflow compiler records these dependencies by creating unique tags for each dependency instead of using variable names. By giving each dependency a unique tag, it allows the non-dependent code segments in the binary to be executed *out of order* and in parallel.

Dataflow Execution

- Programs are loaded into the Content Addressable Memory (CAM) of a dynamic dataflow computer.
- When all of the tagged operands of an instruction become available (that is, output from previous instructions and/or user input), the instruction is marked as ready for execution by an execution unit. This is known as *activating* or *firing* the instruction.
- Once an instruction is completed by an execution unit, its output data is stored (with its tag) in the CAM. Any instructions that are dependent upon this particular datum (identified by its tag value) are then marked as ready for execution.

In a Picture

Manchester Data Flow Machine



Dataflow in Practice

However, in practice the following problems occurred:

– Efficiently broadcasting data tokens in a massively parallel system.
– Efficiently dispatching instruction tokens in a massively parallel system.
– Building Content Addressable Memory (Tag Memory) large enough to hold all of the dependencies of a real program.

Linda Coordination Language

- Main usage: in combination with other existing languages, e.g. C/Fortran, provide a mean to link less expensive desktop computers together and combine their power so they can jointly tackle problems.
- A logically global associative memory, called a tuplespace, in which processes store and retrieve tuples.
- This model is implemented as a "coordination language" in which several primitives operating on ordered sequence of typed data objects, "tuples”
  - **in** atomically reads and removes—consumes—a tuple from tuplespace
  - **rd** non-destructively reads a tuplespace
  - **out** produces a tuple, writing it into tuplespace
  - **eval** creates new processes to evaluate tuples, writing the result into tuplespace

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

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

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

ith field of t

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:

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

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

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

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

$A[F_A^1(t)] \leftarrow A[F_A^2(t)] + B[F_B(t)]$

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 mean 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!!

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

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 (LU factorization)

```
for (k; kεN)
{
    pivot = IDX_A$_{<i,j>[(k,k)]}$();
    forelem (t; t ε A.<i,j>[<(k,∞),k>])
    {
        mult = Value[t.i,t.j]/Value[t.pivot,t.pivot];
        Value[t.i,t.j] = mult;
        forelem (r; r ε A.<i,j>[<t.j,(t.j,∞)>])
        {
            cand = NULL
            forelem (q; q ε A.<i,j>[<t.i,t.j>])
                cand = q;
            if (cand == NULL)
            {
                cand = <t,i,t.j>
                A = A U cand;
                Value[cand.i,cand.j] = 0
            }
            Value[cand.i,cand.j] -= mult*Value[r.i,r.j]
        }
    }
}
```

Example III

SORTING

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

Example IV: MaxFlow

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|

tUPL Loop Body

One or more conditionally executed serial codes operating on data items which are defined by the tokens from the Tuple Reservoir and their associated address functions*, i.e.

tUPL Loop Body:

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

- All `Cond_i`'s are **exclusive for forelem**. For **whilelem** multiple conditions can be **`true`** at the same time for a tuple.
- `n` can be `1` and `Cond_1` can be **`true`**.

*Except for local/temporary variables with respect to the Loop Body

Scheduling **`whilelem`** `(t; t ε T)`

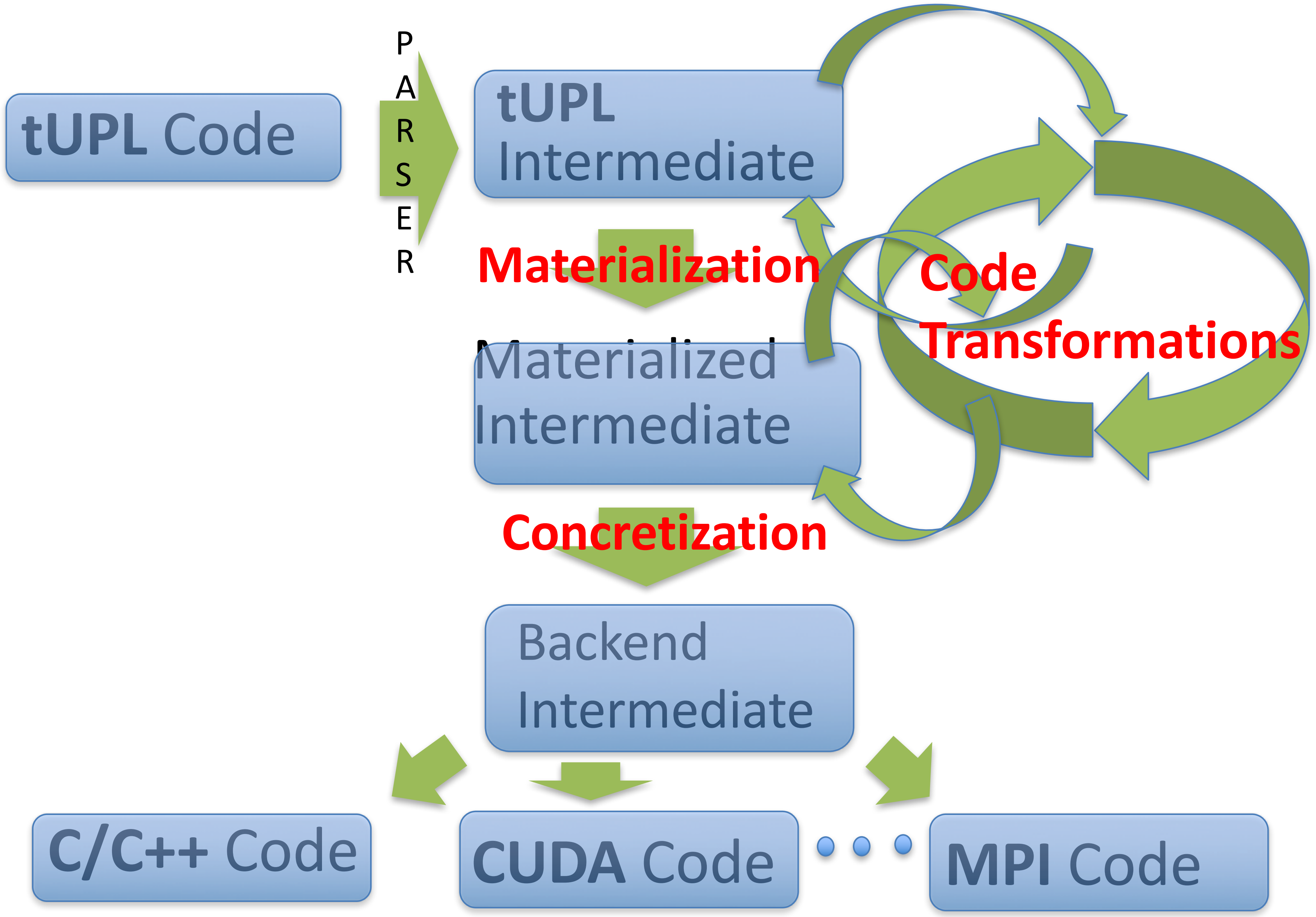
- For each execution of a tuple exactly one of the tuples with a valid conditional serial code is chosen.
- If there are no tuples left with a valid conditional serial code, then the **`whilelem`** loop terminates.
- Any loop scheduling for a **`whilelem`** loop must guarantee that every tuple with a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

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.

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] …
```

$\Rightarrow$

```
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 …
```

$\Rightarrow$

```
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 $N_{10}$)
    forelem (k; k ε N*)
        … A[i][k] …
```

```
forelem (i; i ε $N_{10}$)
    forelem (k; k ε PA_len[i])
        … A[i][k] …
```

Some more code transformations (3)

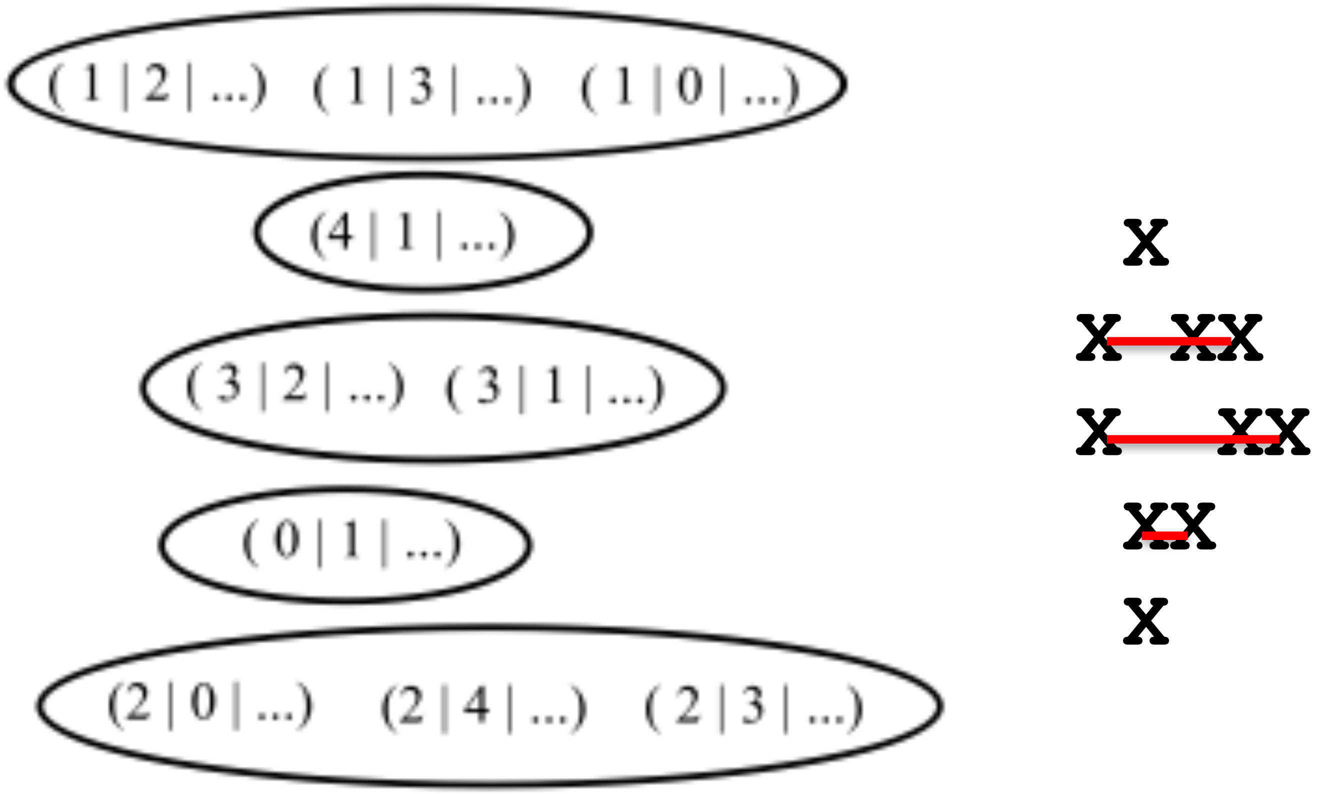
Data Localization

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

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

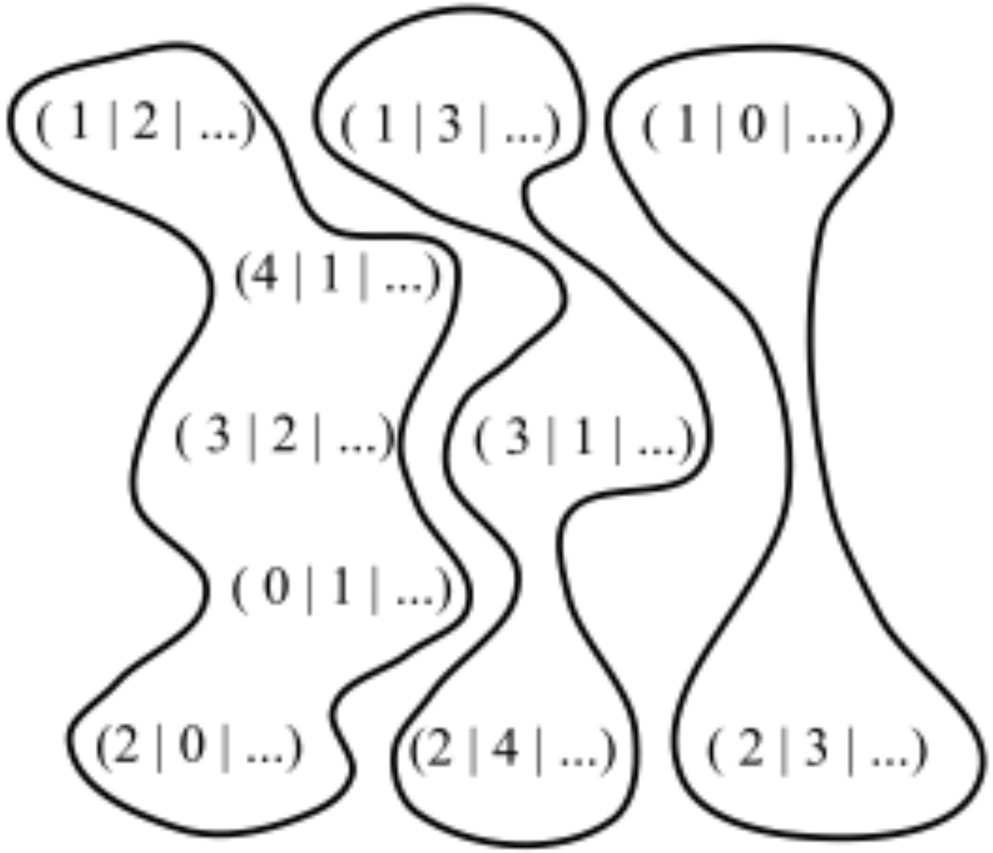
Here the tuples in reservoir A are being extended to include the data at address `@B[A[i].field_k}`. So A’ = { < t, B[t] > | t ε A }. By default, this transformation is only allowed for read only data at B.

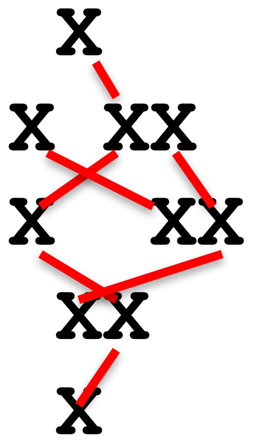
Regrouping of Single Storage Locations (Tuples)



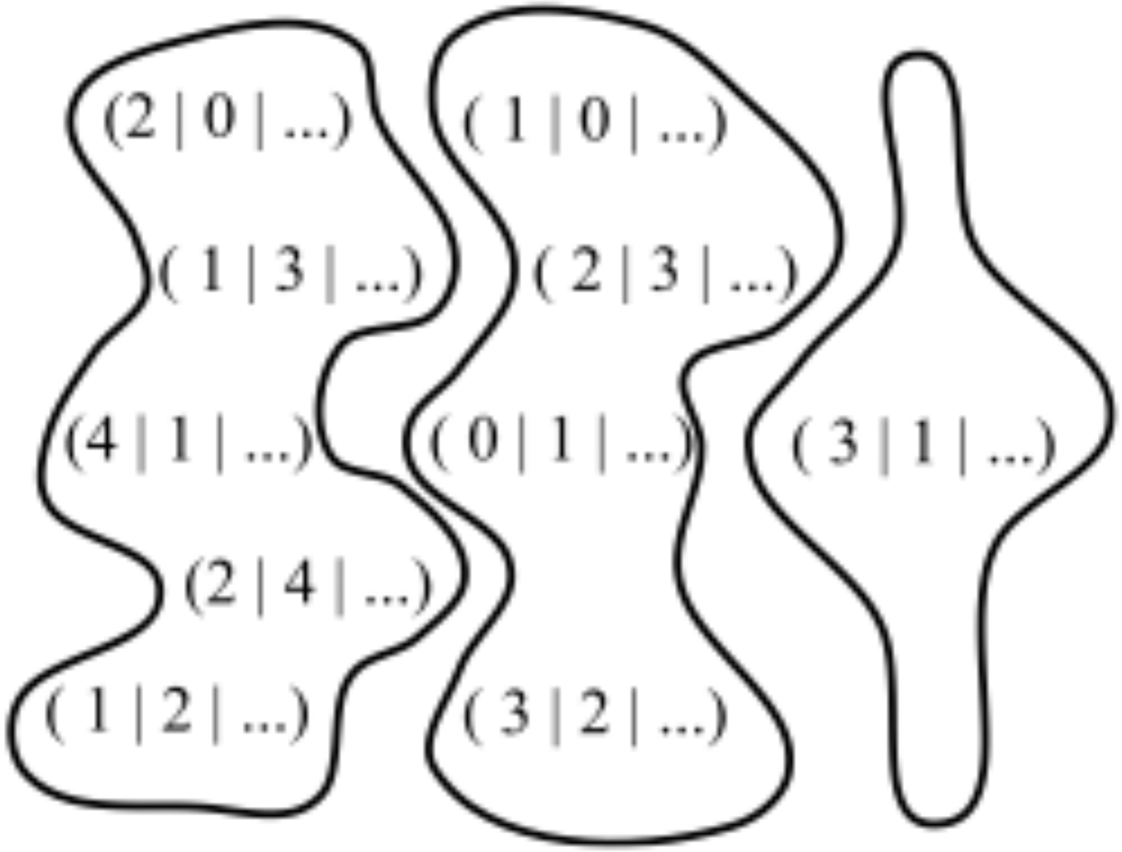
Regrouping as a result of **orthogonalization** on the first field

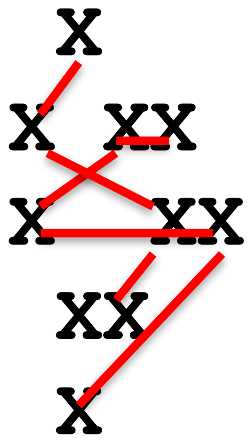
Regrouping after **Materialization** and **Loop Interchange**





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]…
```

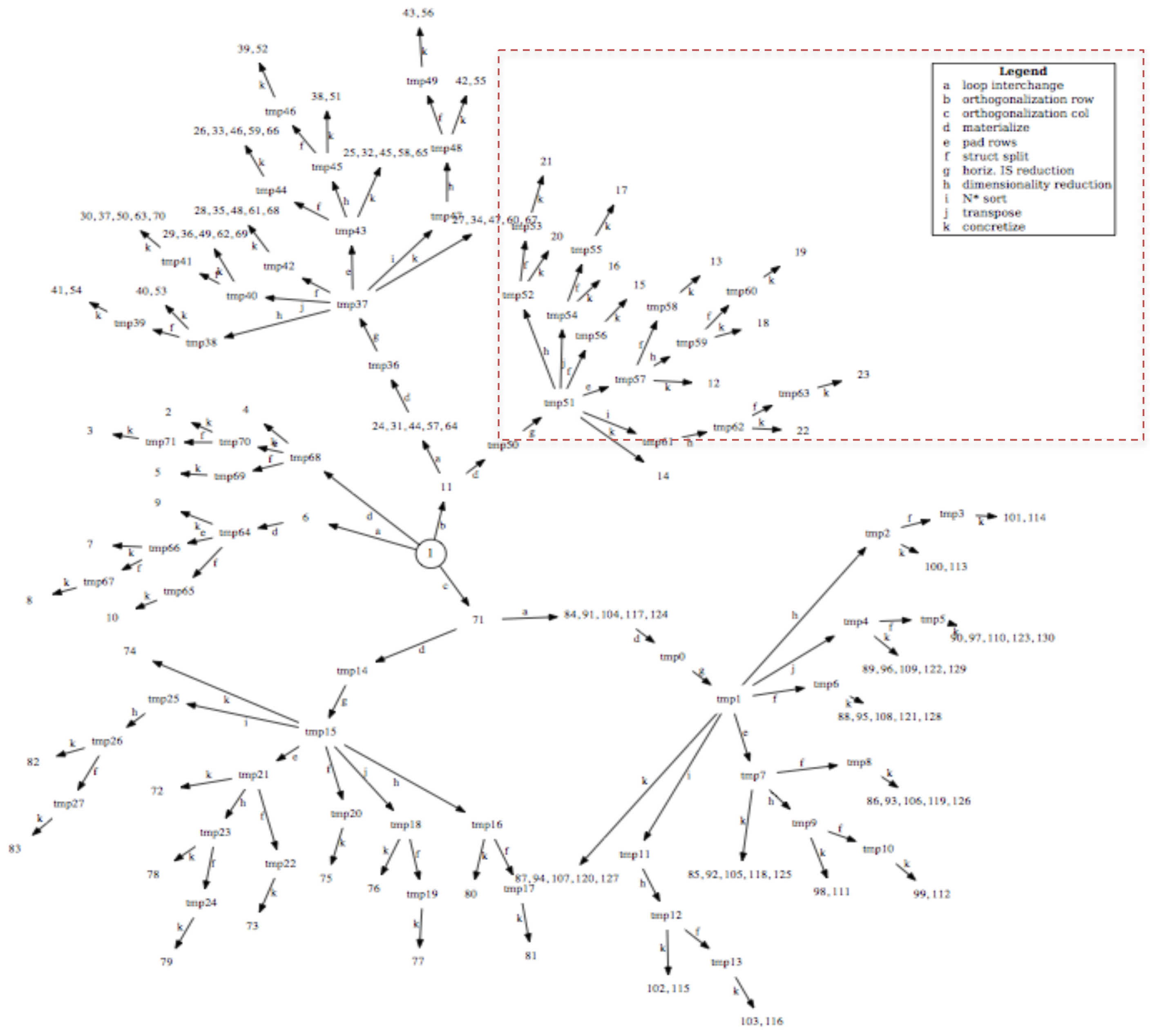
Several Arrays for each field of A'

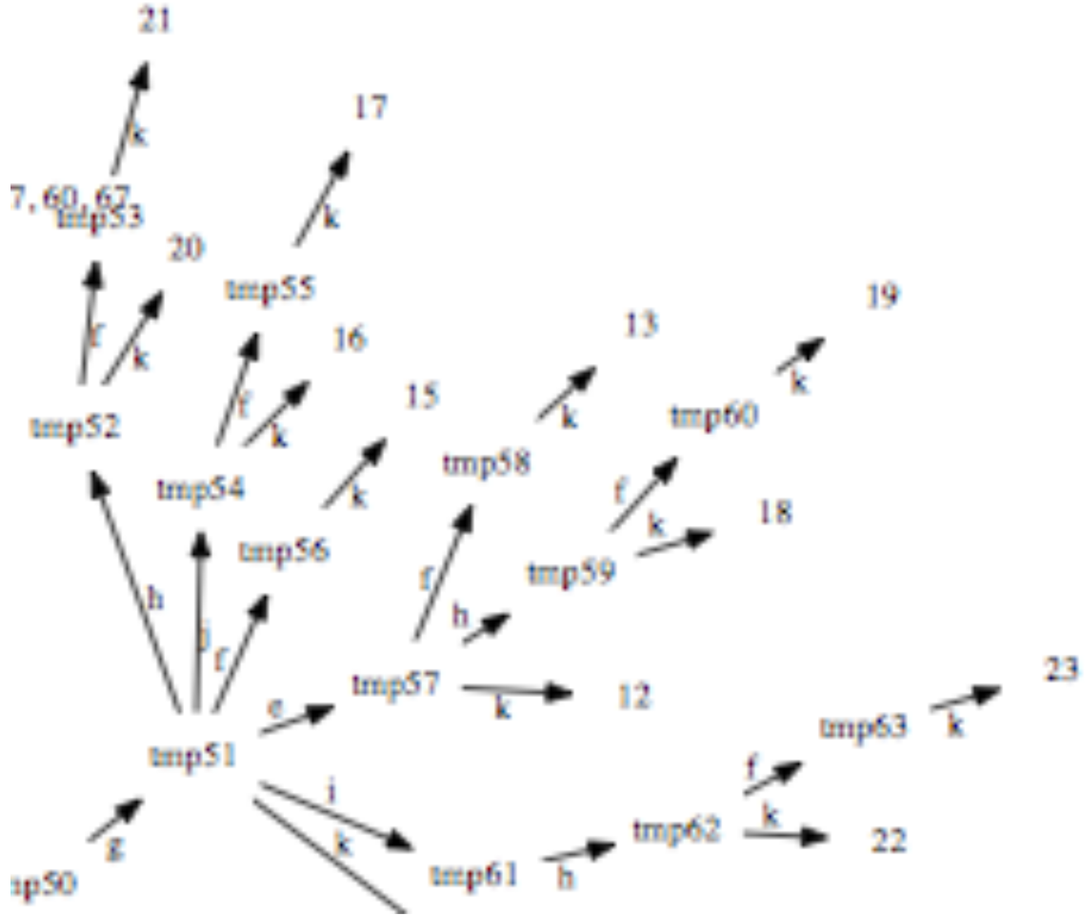
Horizontal Iteration Space Reduction

```
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

Algorithmic Optimization

- **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, `Serial_Code_i`)*

  representing

  `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, `Serial_Code_i`), (t_2, `Serial_Code_i`), …

- 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, `Serial_Code_i`), (t_2, `Serial_Code_i`), …,
  (t_k, `Serial_Code_j`), (t_{k+1}, `Serial_Code_j`), …

- 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

Main Theorem I

For every profit chain C:

all consecutive elements in C:

(t_1, `Serial_Code_i`), (t_2, `Serial_Code_j`)

have a data dependence on an address function

A used in both serial codes: `Serial_Code_i`, `Serial_Code_j`, i.e.

@A[t_1] == @A[t_2]

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.

In fact, the orthogonalization code optimization is a direct result of this chaining

Covering Chain Set

A covering chain set CCS is a set of Chains C_i such that for every tuple ($\texttt{t}_k$, `Serial_Code_i`) there is an i such that

($\texttt{t}_k$, `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 ($\texttt{t}_k$, `Serial_Code_i`) pairs which were not covered, to obtain a covering chain set.

Main Theorem II

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 <u>semantically equivalent</u>.

```
   **forelem** ( t; t ε T )
```

and

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

are semantically equivalent just based on the covering property of CCS.

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!!!

Examples 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!!!

Parallel Programming II (this spring)

- 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.
- In fact within tUPL **new algorithms can be automatically generated** which will not only execute in parallel but will also be adaptive to the underlying problem to be solved.

END OF COURSE