

tUPL Parallel Programming Paradigm

Data Flow Computing

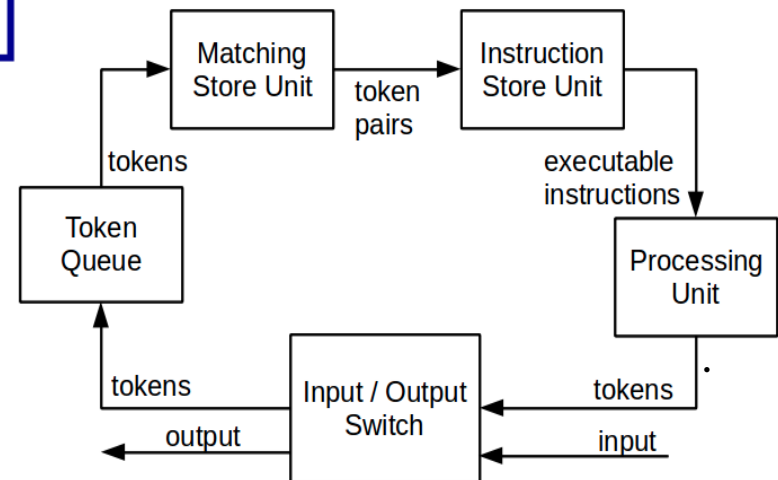
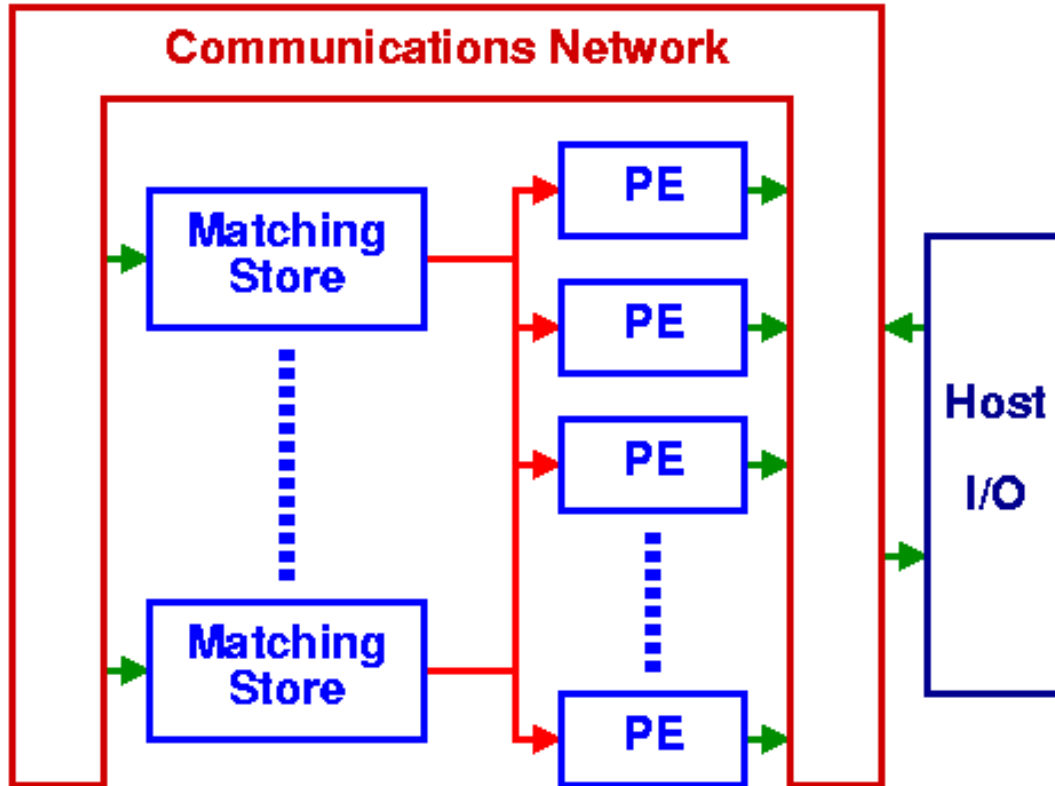
Traditionally, compilers analyze program source code for data dependencies between instructions in order to better organize the instruction sequences in the binary output files.

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



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
- 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

< token >_A

with A an address function, s.t. $data$ is stored at: $@A$ [token]

and the value which is stored at $@A$ [token] is: A [token]

So $data == A$ [token]

Address function $A(\text{addr})$

$A(\text{addr})$ can be **any invertible function**, i.e. affine mapping, but mostly it is an affine function:

$$\mathbb{Z}^n \rightarrow \mathbb{Z}^k$$

So Addr can be represented as

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

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 at location @A [I, J], or that the **data** value of $\langle I, J \rangle_A$ becomes 5.0, or that $\langle I, J, \text{data} \rangle = \langle I, J, 5.0 \rangle^*$

- Note that tokens can be more dimensional: **token tuples t**
- Notation: **t.i**, represents the i^{th} field of t

SO, **data structures** as we know them do not exist 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** $G(V,E)$:

$$T = \{ \langle u, v \rangle \mid u, v \in V \text{ and } (u,v) \in E \}$$

with **Weight** $[u, v]$ the weight of edge (u,v)

A **Sparse Matrix** A :

$$T = \{ \langle i, j \rangle \mid \text{at row } i \text{ and column } j \\ \text{there is a nnz element} \}$$

with **Value** $[i, j]$ the value of matrix $A [i, j]$

Examples of Tuple Reservoirs (II)

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

$$T = \{ \langle i_k, j_k \rangle \mid \begin{array}{l} 1 \leq k \leq n, \\ \text{for every } j_k, 1 \leq k < n, \\ \text{there exists exactly one } i_m, \\ \text{such that } j_k = i_m, \text{ and} \\ \text{for all } j_k, 1 \leq k \leq n, \\ \text{the values are different} \end{array} \}$$

Together with an address function **Value [i_k, j_k]** representing the value at the k^{th} position in the list.

OR address function **Value [i_k] !** (tUPL allows both)

Examples of Tuple Reservoirs (III)

Relational Database Tables

$T = \{ \langle i \rangle \mid 1 \leq i \leq n, \text{ with } i \text{ representing the } i^{\text{th}} \text{ record in the database table} \}$

and associated address functions:

$\text{field}_1 [i], \text{field}_2 [i], \dots, \text{field}_t [i]$

tUPL Loop Structures

Two **BASIC** Loop Structures:

forelem ($t; t \in T$)

whilelem ($t; t \in T$)

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  $\in$  T )  
{  
    Value_C[t.i] += Value_A[t.i, t.j]  
                  * Value_B[t.j]  
}
```


Example II (LU factorization)

```
for (k; k ∈ N)
{
  pivot = IDX_A<i,j>[(k,k)] ();
  forelem (t; t ∈ 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 ∈ A.<i,j>[<t.j,(t.j,∞)>])
    {
      cand = NULL
      forelem (q; q ∈ 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  $\in$  T )  
{  
    if ( X[t.i] > X[t.j] )  
        swap ( X[t.i], X[t.j] )  
}
```

Example IV: MaxFlow

$T = \{ \langle u,v,w \rangle \mid (u,v) \text{ and } (v,w) \text{ edges of } G \text{ and } w \neq u \text{ and (including back edges)} \}^*$

```
whilelem ( t; t ∈ 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|V|
= aver_out^2*|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

tUPL Loop Bodies: Two Cases

- I For all i, j, k , and m there is no flow (anti) dependence between

`Serial_Code_i` ($\langle t_k \rangle$) and
`Serial_Code_j` ($\langle t_m \rangle$)

- II There exists i, j, k , and m for which there is flow (anti) dependence between

`Serial_Code_i` ($\langle t_k \rangle$) and
`Serial_Code_j` ($\langle t_m \rangle$)

Case I

→ There are no Read on Write Dependencies, and/or Write on Read Dependencies

→ For all i and k : repeated execution of just

`Serial_Code_i (< tk >)`

does not have any effects, even when these executions are interleaved with the execution of other

`Serial_Code_j (< tm >)`

→ Use of

forelem (t ; $t \in T$)

→ As a result each `Serial_Code_i (< tk >)` is executed **exactly once**, albeit in arbitrary order

Case II

- There are dependencies between the execution of one conditional serial code using one tuple with the execution of (possibly another) conditional code on another (the same) tuple
- **tUPL** relies on tuples to be taken at any time
- These dependencies have to be broken
- Leading to possible repetitive execution of the same tuples (see next slides)
- Use of

whilelem ($t; t \in T$)

Case III

In case, a prefixed order of tuples is required for the execution of the **tUPL** loop body, then **tUPL** foresees in the use of a **ready** clause in the condition clause of the **whilelem** construct.

```
whilelem ( t ; t  $\in$  T )  
  if ( ready (q) [q.i < t.j] ) )
```

The use of this clause will severely limit the optimization possibilities of the **tUPL** framework.

Example

```
T = {<1>, <2>, ..., <100>}; # <iter>
```

```
whilelem ( t; t ∈ T )  
{  
    if ( ready(q) [q.iter < t.iter] )  
    {  
        ...  
    }  
}
```

Note that:

- **tUPL** also allows the following notation (tuples t have only 1 field)

```
if ( ready(q) [q < t] )
```

- If there are no tuples found which fulfill the ready clause condition then ready evaluates to true
- The use of ready does not prevent that for each iteration the loop body is being executed multiple times. Use of **forelem** is recommended in combination with the **ready** clause if only one execution per iteration is meant.

Example (continued)

```
T = {<1>, <2>, ..., <100>}; # <iter>
```

```
whilelem ( t; t ∈ T )  
{  
    if ( ready(q) [q < t] )  
    {  
        ...  
    }  
}
```

In this case **tUPL** allows the following shorthand notation:

```
for (k; k ∈  $\mathbf{N}^+_{100}$ ) *  
{  
    ...  
}
```

* $k \in \mathbf{N}^+_{100}$ is used for $k \in T = \{<1>, <2>, \dots, <100>\}$,
 $k \in \mathbf{N}_{100}$ is used for $k \in T = \{<0>, <1>, \dots, <100>\}$

Case IV

As a last resort the programmer can use the condition clause in the **whilelem** loop body to explicitly control the order in which the tuples are visited.

```
whilelem ( q; q  $\in$  T )
{ if ( q.row > q.col &&
      Count[q.col] == 0 &&
      Visited[q] = False )
  {
    B[q.row] = B[q.row] - Value[q]*B[q.col];
    Count[q.row + 1]--;
    Visited[q] = True;
  }
}
```

Data Dependencies Denial (DDD) in tUPL

The tUPL compiler framework will (by default):

translate data dependent instructions into independent instructions,
by introducing **an extra address function (\$Old_....)** for each address function on which a data dependence occurs.

Suppose w.l.o.g. that data dependence occurs on address function X , so that

for tuple t_1 : $X[t_1] = \dots$

for tuple t_2 : $y = \dots X[t_2] \dots$

and that the storage locations are equal: $@X[t_1] == @X[t_2]$

tUPL automatically transforms this into:

Initialize new address function.

forall t : $\$Old_X[t] = NULL$

And the read instructions are transformed into:

while $(X[t] != \$Old_X[t])$ $y = \dots X[t] \dots$; $\$Old_X[t] = X[t]$

The “while” construct is merged into the **whilelem** construct.

forall t: \$Old_X[t] = NULL

X[t₁] = 5;

while (X[t₂] != \$Old_X[t₂]) y = X[t₂]; \$Old_X[t₂] = X[t₂]

→ y = 5



forall t: \$Old_X[t] = NULL

while (X[t₂] != \$Old_X[t₂]) y = X[t₂]; \$Old_X[t₂] = X[t₂]
(y = ...)

X[t₁] = 5;

while (X[t₂] != \$Old_X[t₂]) y = X[t₂]; \$Old_X[t₂] = X[t₂]

→ y = 5



Why not use the same construct for write instructions ?

$X[t_1] = 5$
 $X[t_2] = 10$

and $@X[t_1] == @X[t_1]$, then

```
while (X[t1] != $Old_X[t1]) X[t1] = 5; $Old_X[t1] = X[t1]  
while (X[t2] != $Old_X[t2]) X[t2] = 10; $Old_X[t2] = X[t2]  
while (X[t1] != $Old_X[t1]) X[t1] = 5; $Old_X[t1] = X[t1]  
while (X[t2] != $Old_X[t2]) X[t2] = 10; $Old_X[t2] = X[t2]
```

.....

Etc. etc.

➔ RACE CONDITION

Example

$T = \{ \langle 1, 3 \rangle, \langle 2, 1 \rangle, \langle 3, 2 \rangle \}; \# \langle i, j \rangle$

$A[1, 1] = 6; A[1, 3] = 2; A[2, 1] = -1; A[2, 2] = 5; A[3, 2] = 2; A[3, 3] = 4$

$B[1] = 8; B[2] = 9; B[3] = 8;$

$X[1] = 0; X[2] = 0; X[3] = 0;$

```
whilelem ( t; t ∈ T )
```

```
{
```

```
     $X[t.i] = (B[t.i] - X[t.j] * A[t.i, t.j]) / A[t.i, t.i]$ 
```

```
}
```

(What is being computed???)

→ **THERE IS A FLOW DEPENDENCE ON X !!!!!**

For instance $\langle 1, 3 \rangle : X[1] = (B[1] - X[3] * A[1, 3]) / A[1, 1]$

$\langle 2, 1 \rangle : X[2] = (B[2] - X[1] * A[2, 1]) / A[2, 2]$

Loop is transformed

```
T = {<1, 3>, <2, 1>, <3, 2>}; #<i, j>
```

```
A[1, 1]=6; A[1, 3]=2; A[2, 1]=-1; A[2, 2]=5; A[3, 2]=2; A[3, 3]=4
```

```
B[1]=8; B[2]=9; B[3]=8;
```

```
X[1]=0; X[2]=0; X[3]=0;
```

```
$Old_X[<1, 3>]=NULL; $Old_X[<2, 1>]=NULL; $Old_X[<3, 2>]=NULL;
```

```
whilelem ( t; t ∈ T )
```

```
{
```

```
    if (X[t.j] != $Old_X[t])
```

```
    {
```

```
        X[t.i] = (B[t.i] - X[t.j] * A[t.i, t.j]) / A[t.i, t.i];
```

```
        $Old_X[t] = X[t.j]
```

```
    }
```

```
}
```


Resulting Execution Orders

$\langle 1,3 \rangle$ # $X[t.j] = X[3] = 0$ and $\$Old_X[t] = NULL$
 $X[1] = (8 - X[3] * 2) / 6 = 8 / 6 = 1.333$
 $\$Old_X[\langle 1,3 \rangle] = 0.000$
 $\langle 2,1 \rangle$ # $X[t.j] = X[1] = 1.333$ and $\$Old_X[t] = NULL$
 $X[2] = (9 - X[1] * -1) / 5 = 10.333 / 5 = 2.067$
 $\$Old_X[\langle 2,1 \rangle] = 1.333$
 $\langle 3,2 \rangle$ # $X[t.j] = X[2] = 2.067$ and $\$Old_X[t] = NULL$
 $X[3] = (8 - X[2] * 2) / 4 = (8 - 4.134) / 4 = 1.466$
 $\$Old_X[\langle 3,2 \rangle] = 2.067$
 $\langle 1,3 \rangle$ # $X[t.j] = X[3] = 1.466$ and $\$Old_X[t] = 0.000$
 $X[1] = (8 - X[3] * 2) / 6 = (8 - 2.932) / 6 = 0.845$
 $\$Old_X[\langle 1,3 \rangle] = 1.466$
 $\langle 2,1 \rangle$ # $X[t.j] = X[1] = 0.845$ and $\$Old_X[t] = 1.333$
 $X[2] = (9 - X[1] * -1) / 5 = (9 + 0.845) / 5 = 1.969$
 $\$Old_X[\langle 2,1 \rangle] = 0.845$
 $\langle 3,2 \rangle$ # $X[t.j] = X[2] = 1.969$ and $\$Old_X[t] = 2.067$
 $X[3] = (8 - X[2] * 2) / 4 = (8 - 3.938) / 4 = 1.015$
 $\$Old_X[\langle 3,2 \rangle] = 1.969$
 $\langle 1,3 \rangle$ # $X[t.j] = X[3] = 1.015$ and $\$Old_X[t] = 1.466$
 $X[1] = (8 - X[3] * 2) / 6 = (8 - 2.030) / 6 = 0.995$
 $\$Old_X[\langle 1,3 \rangle] = 1.015$
 $\langle 2,1 \rangle$ # $X[t.j] = X[1] = 0.995$ and $\$Old_X[t] = 0.845$
 $X[2] = (9 - X[1] * -1) / 5 = (9 + 0.995) / 5 = 1.999$
 $\$Old_X[\langle 2,1 \rangle] = 0.995$
 $\langle 3,2 \rangle$ # $X[t.j] = X[2] = 1.999$ and $\$Old_X[t] = 1.969$
 $X[3] = (8 - X[2] * 2) / 4 = (8 - 3.998) / 4 = 1.000$
 $\$Old_X[\langle 3,2 \rangle] = 1.999$
 $\langle 1,3 \rangle$ # $X[t.j] = X[3] = 1.000$ and $\$Old_X[t] = 1.015$
 $X[1] = (8 - X[3] * 2) / 6 = (8 - 2.000) / 6 = 1.000$
 $\$Old_X[\langle 1,3 \rangle] = 1.000$
 $\langle 2,1 \rangle$ # $X[t.j] = X[1] = 1.000$ and $\$Old_X[t] = 0.995$
 $X[2] = (9 - X[1] * -1) / 5 = (9 + 1.000) / 5 = 2.000$
 $\$Old_X[\langle 2,1 \rangle] = 1.000$
 $\langle 3,2 \rangle$ # $X[t.j] = X[2] = 2.000$ and $\$Old_X[t] = 1.999$
 $X[3] = (8 - X[2] * 2) / 4 = (8 - 4.000) / 4 = 1.000$
 $\$Old_X[\langle 3,2 \rangle] = 2.000$

OR

$\langle 2,1 \rangle$ # $X[t.j] = X[1] = 0$ and $\$Old_X[t] = NULL$
 $X[2] = (9 - X[1] * -1) / 5 = 9 / 5 = 1.800$
 $\$Old_X[\langle 2,1 \rangle] = 0.000$
 $\langle 1,3 \rangle$ # $X[t.j] = X[3] = 0$ and $\$Old_X[t] = NULL$
 $X[1] = (8 - X[3] * 2) / 6 = 8 / 6 = 1.333$
 $\$Old_X[\langle 1,3 \rangle] = 0.000$
 $\langle 2,1 \rangle$ # $X[t.j] = X[1] = 1.333$ and $\$Old_X[t] = 0.000$
 $X[2] = (9 - X[1] * -1) / 5 = 10.333 / 5 = 2.067$
 $\$Old_X[\langle 2,1 \rangle] = 1.333$
 $\langle 1,3 \rangle$ # $X[t.j] = X[3] = 0$ and $\$Old_X[t] = 0.000$
NOP
 $\langle 2,1 \rangle$ # $X[t.j] = X[1] = 1.333$ and $\$Old_X[t] = 1.333$
NOP
 $\langle 3,2 \rangle$ # $X[t.j] = X[2] = 2.067$ and $\$Old_X[t] = NULL$
 $X[3] = (8 - X[2] * 2) / 4 = (8 - 4.134) / 4 = 1.466$
 $\$Old_X[\langle 3,2 \rangle] = 2.067$
 $\langle 1,3 \rangle$ # $X[t.j] = X[3] = 1.466$ and $\$Old_X[t] = 0.000$
 $X[1] = (8 - X[3] * 2) / 6 = (8 - 2.932) / 6 = 0.845$
 $\$Old_X[\langle 1,3 \rangle] = 1.466$
 $\langle 3,2 \rangle$ # $X[t.j] = X[2] = 2.067$ and $\$Old_X[t] = 2.067$
NOP
 $\langle 2,1 \rangle$ # $X[t.j] = X[1] = 0.845$ and $\$Old_X[t] = 1.333$
 $X[2] = (9 - X[1] * -1) / 5 = (9 + 0.845) / 5 = 1.969$
 $\$Old_X[\langle 2,1 \rangle] = 0.845$

... ..

Otherwise similar (except for interleaving with NOPs)

Scheduling `whilelem` ($t; t \in T$)

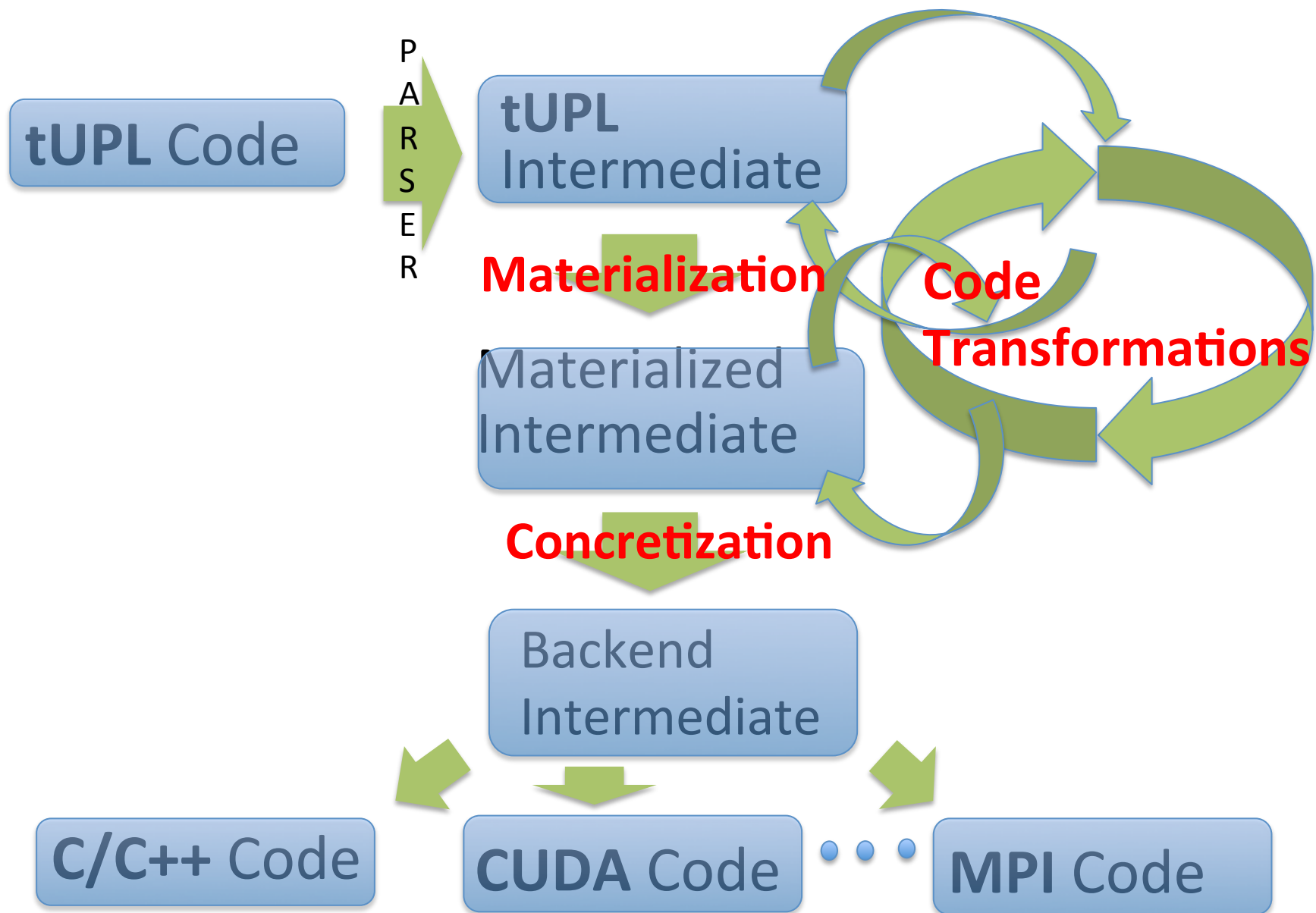
- For each execution of a tuple exactly one of the tuples with a valid conditional serial code is chosen.
- If there are no tuples left with a valid conditional serial code, then the `whilelem` loop terminates.
- Any loop scheduling for a `whilelem` loop must guarantee that every tuple with a valid conditional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

Scheduling `forelem` ($t; t \in T$)

- 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  $\in$  T )
```

```
{
```

```
    ... t ...
```

```
}
```

```
whilelem ( t; t  $\in$  T )
```

```
{
```

```
    ... t ...
```

```
}
```



```
forelem ( i; i  $\in$  pT )
```

```
{
```

```
    ... T[i] ...
```

```
}
```

```
whilelem ( i; i  $\in$  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

Tuple Selectors

- $(i; i \in pT. (row, col) [<10, 100 >])$ means choose only these tuples from the reservoir T for which the row field equals 10 and col field equals 100.
- $IDX_T_{<row, col> [<10, 100 >] } ()$ within the loop body refers to the tuple i, for which row equals 10 and col equals 100.
- $[<10, (100, 1000) >]$ refers to the second field to be of a value between 10 and 1000.

Some Code Transformations*

Orthogonalization

```
forelem (i; i  $\in$  pA)  
  ... A[i]...
```



```
forelem (ii; ii  $\in$  A.field1)  
  forelem (i; i  $\in$  pA.field1[ii])  
    ... A[i]...
```

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

Encapsulation

```
forelem (i; i  $\in$  pA.field1)  
  ... ..
```



```
forelem (i; i  $\in$  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  $\in$  pA)  
  forelem (j; j  $\in$  pB.field_b[A[i].field_a])  
    ... A[i].field_c ... B[j].field_d ...
```



```
forelem (i; i  $\in$  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: $\{ \langle i, j \rangle \mid i \in A \text{ and } j \in B \}$

Some Code Transformations (3)

Loop Interchange


```
forelem (i; i ∈ pA)
  forelem (j; j ∈ pB)
    ... A[i] ... B[j] ...
```



```
forelem (j; j ∈ pB)
  forelem (i; i ∈ pA)
    ... A[i] ... B[j] ...
```

Horizontal Iteration Space Reduction

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



```
forelem (i; i ∈ pA')
```

```
  ... A'[i].field2 ... A'[i].field3 ...
```

With $A' = \{ \langle \text{field2}, \text{field3} \rangle \mid \langle \text{field1}, \text{field2}, \text{field3} \rangle \in A \}$

Materialization

```
forelem (i; i  $\in$  pA.field[X])  
... A[i]...
```



```
forelem (i; i  $\in$  N*)  
... PA[i]...
```

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

$$\{ i \mid i \in A \text{ and } i.\text{field} == X \}$$

DO NOT CONFUSE PA with a linear array data structure

Some more code transformations

Tuple Splitting

```
forelem (i; i  $\varepsilon$  A.field)
```

```
  forelem (k; k  $\varepsilon$  pB.field[i])
```

```
    ... B[k].value ...
```



```
forelem (i; i  $\varepsilon$   $\mathbf{N}_{10}$ )
```

```
  forelem (k; k  $\varepsilon$  pB.field[i])
```

```
    ... B[k].value ...
```



```
forelem (i; i  $\varepsilon$   $\mathbf{N}_{10}$ )
```

```
  forelem (k; k  $\varepsilon$   $\mathbf{N}^*$ )
```

```
    ... B[i][k].value ...
```



```
forelem (i; i  $\varepsilon$   $\mathbf{N}_{10}$ )
```

```
  forelem (k; k  $\varepsilon$   $\mathbf{N}^*$ )
```

```
    ... B[i].value[k] ...
```

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

Some more code transformations (2)

N* Materialization

```
forelem (i; i  $\in$   $\mathbf{N}_{10}$ )  
  forelem (k; k  $\in$   $\mathbf{N}^*$ )  
    ... A[i][k] ...
```



```
forelem (i; i  $\in$   $\mathbf{N}_{10}$ )  
  forelem (k; k  $\in$  PA_len[i])  
    ... A[i][k] ...
```

Some more code transformations (3)

Data Localization

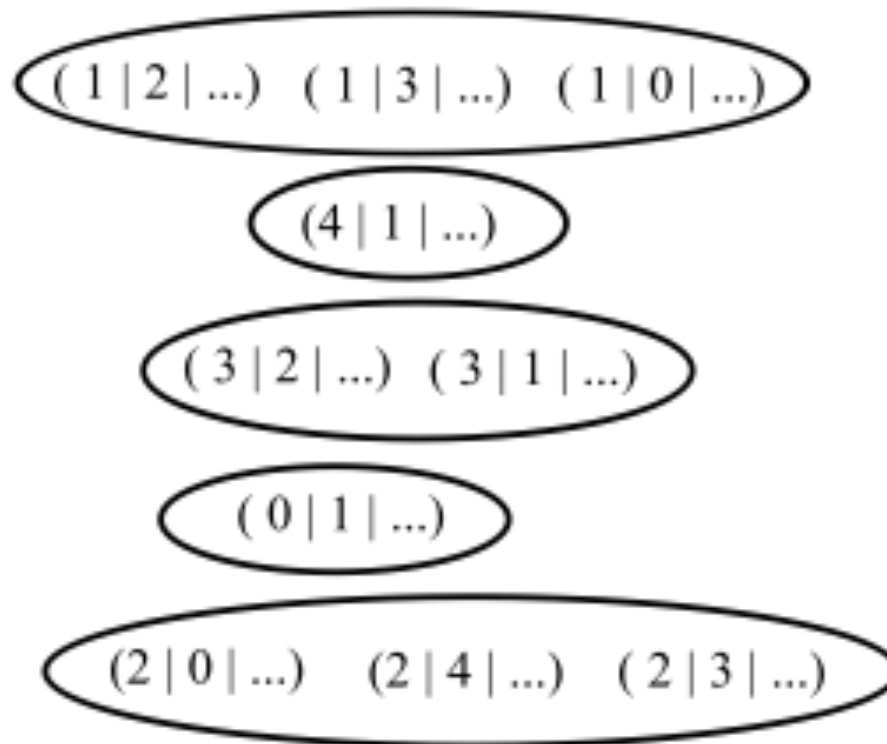
```
forelem (i; i  $\in$  pA)  
... B [ A[i] ] ...
```



```
forelem (i; i  $\in$  pA')  
... A'[i].field_B ...
```

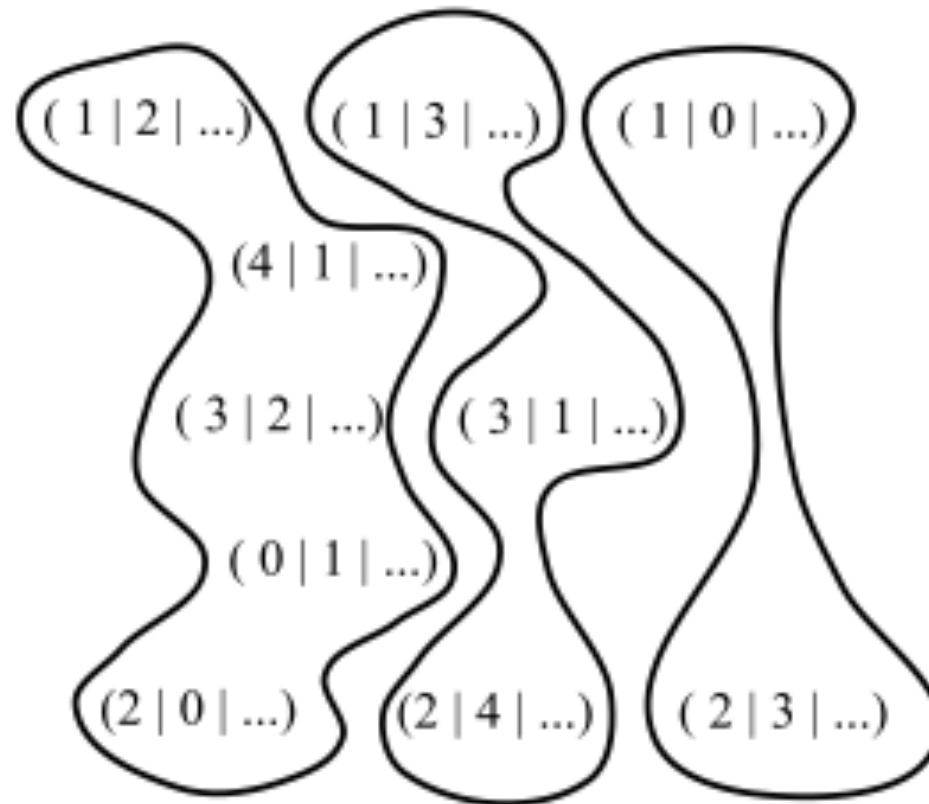
Here the tuples in reservoir A are being extended to include the data at address `@B[A[i].field_k]`. So $A' = \{ \langle t, B[t] \rangle \mid t \in A \}$. By default, this transformation is only allowed for read only data at B.

Regrouping of Single Storage Locations (Tuples)

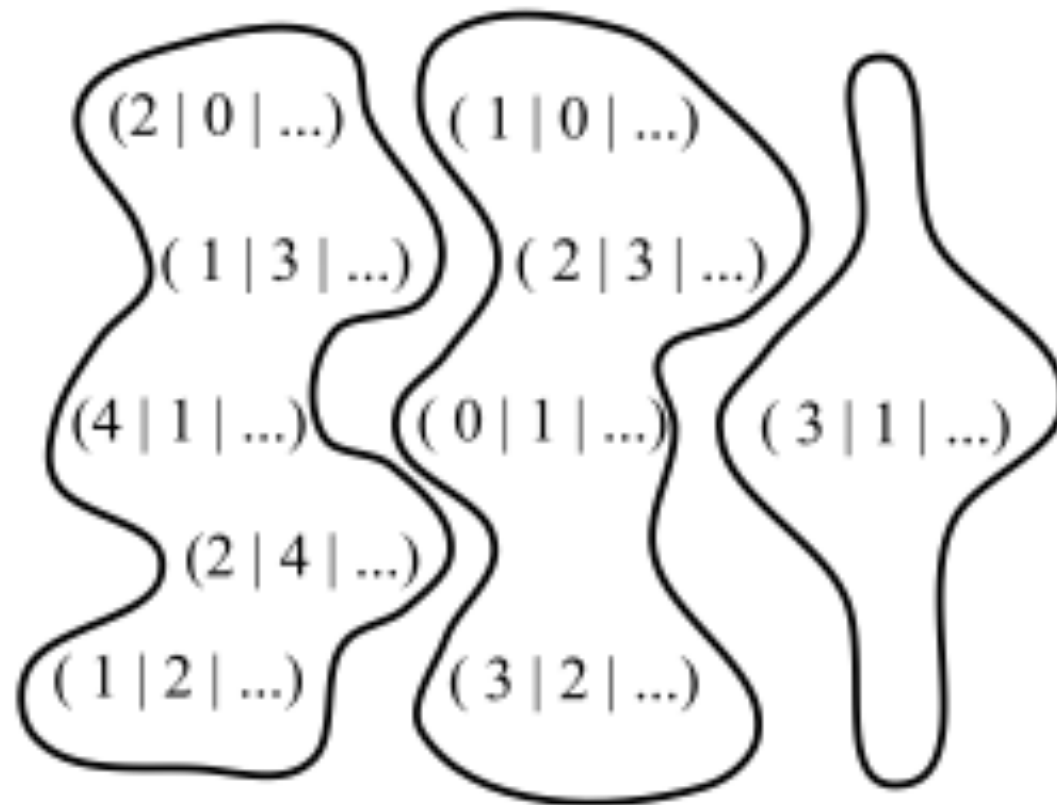


Regrouping as a result of orthogonalization on the first field

Regrouping after Materialization and Loop Interchange



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



Concretization

```
forelem (i; i  $\in$   $\mathbf{N}^*$ )  
... PA[i]...
```



```
forelem (i; i  $\in$  PA_len[i])  
... PA[i] ...
```



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

Some Concretization Steps

tUPLE loop construct	Concretization
<pre>forelem (i; i ∈ pA) ... A[i]...</pre>	Linked list of struct's
<pre>forelem (i; i ∈ N₁₀) ... A[i]...</pre>	An array of struct's
<pre>forelem (i; i ∈ N₁₀) forelem (k; k ∈ PA_len[i]) ... A[i][k] ...</pre>	An array of arrays of struct's
<pre>forelem (i; i ∈ N₁₀) forelem (k; k ∈ PA_len[i]) ... A[i][k].value ...</pre>	An array of arrays of struct's
<pre>forelem (i; i ∈ N₁₀) forelem (k; k ∈ PA_len[i]) ... A[i].value[k] ...</pre>	An array of arrays of values

Example

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

Data Localization

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

Materialization

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

Tuple Splitting

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

Horizontal Iteration Space Reduction

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

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

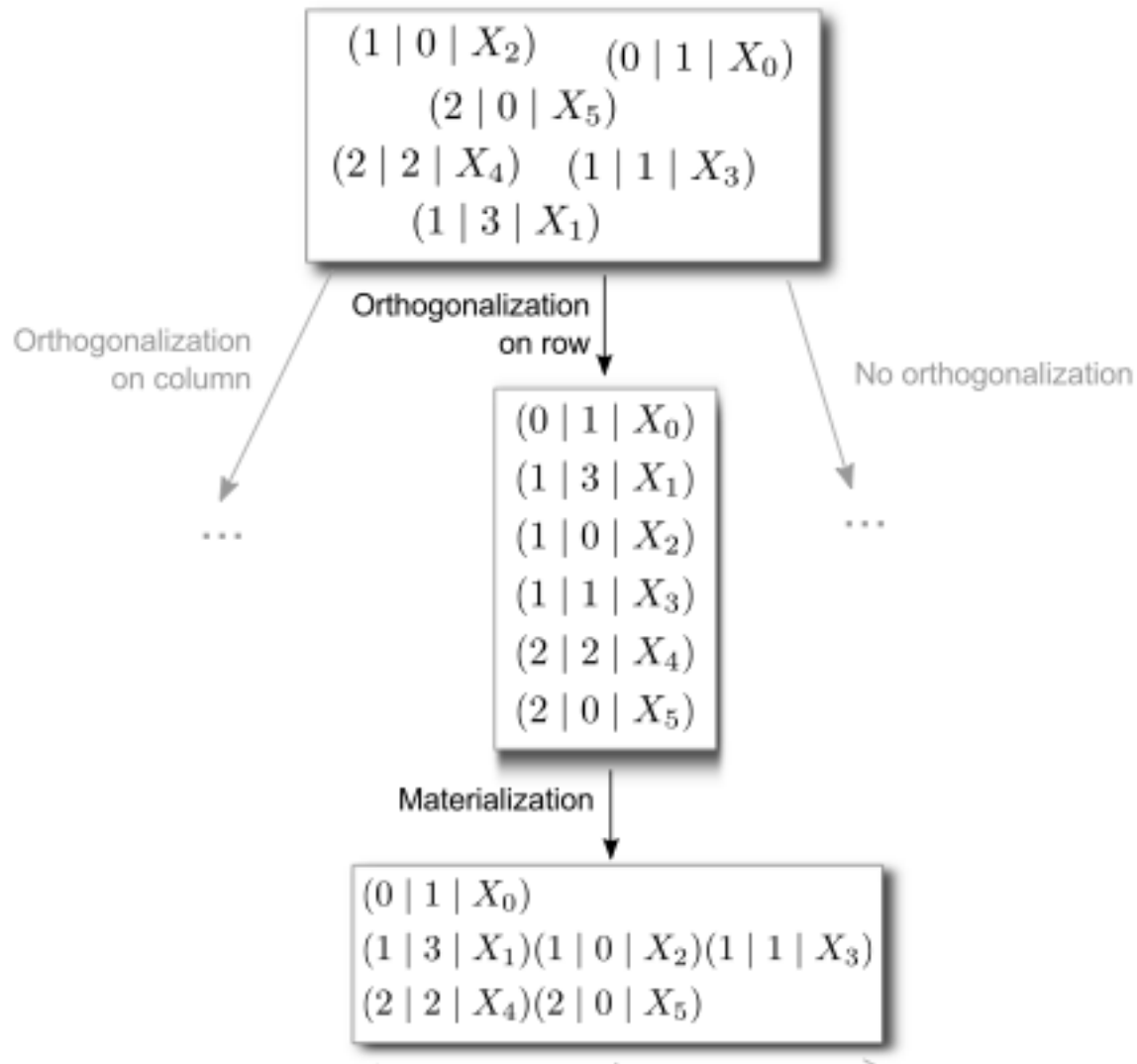
An linked list of struct's: A

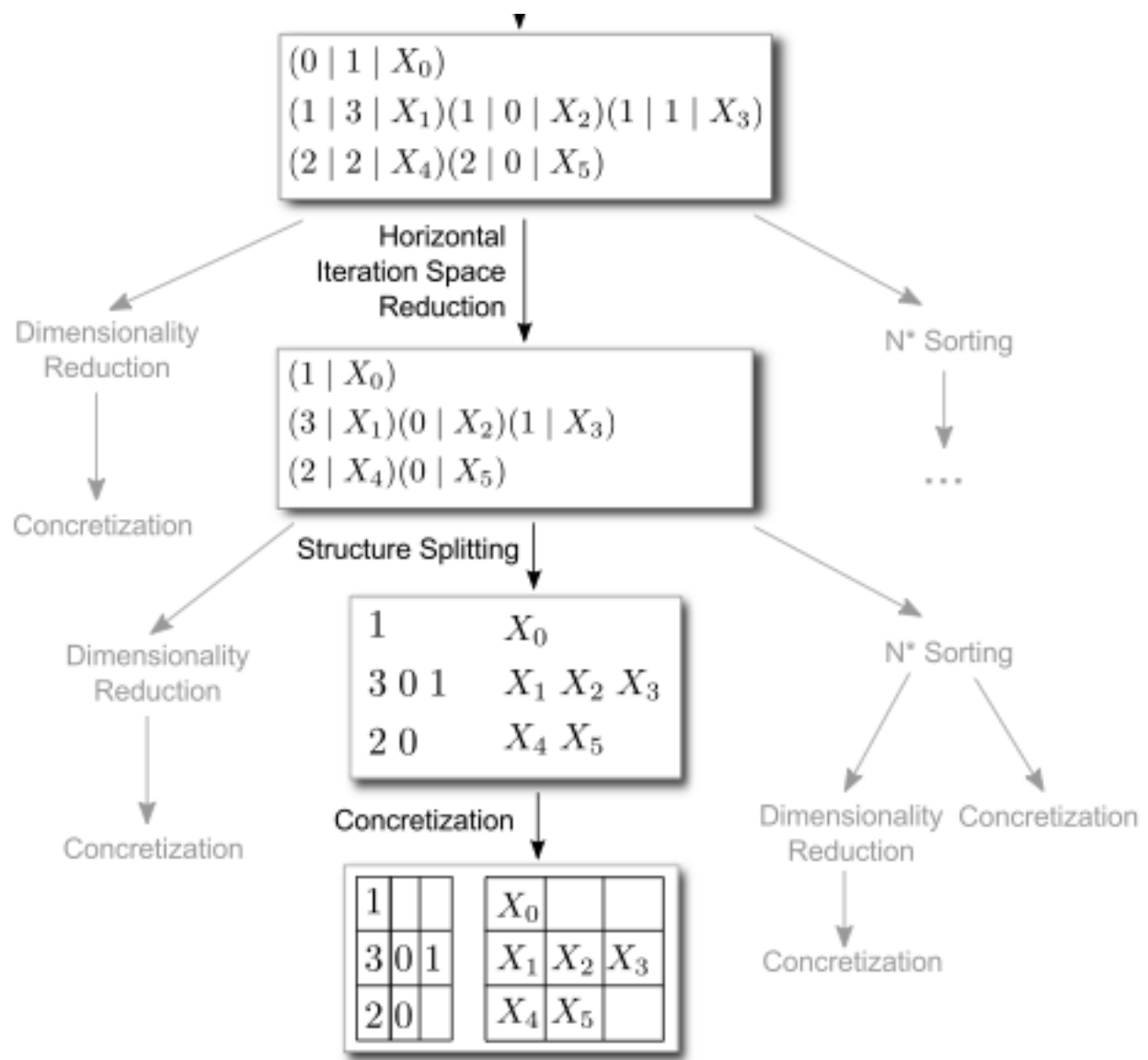
An array of struct's A'

**Several Arrays for each field
of A'**

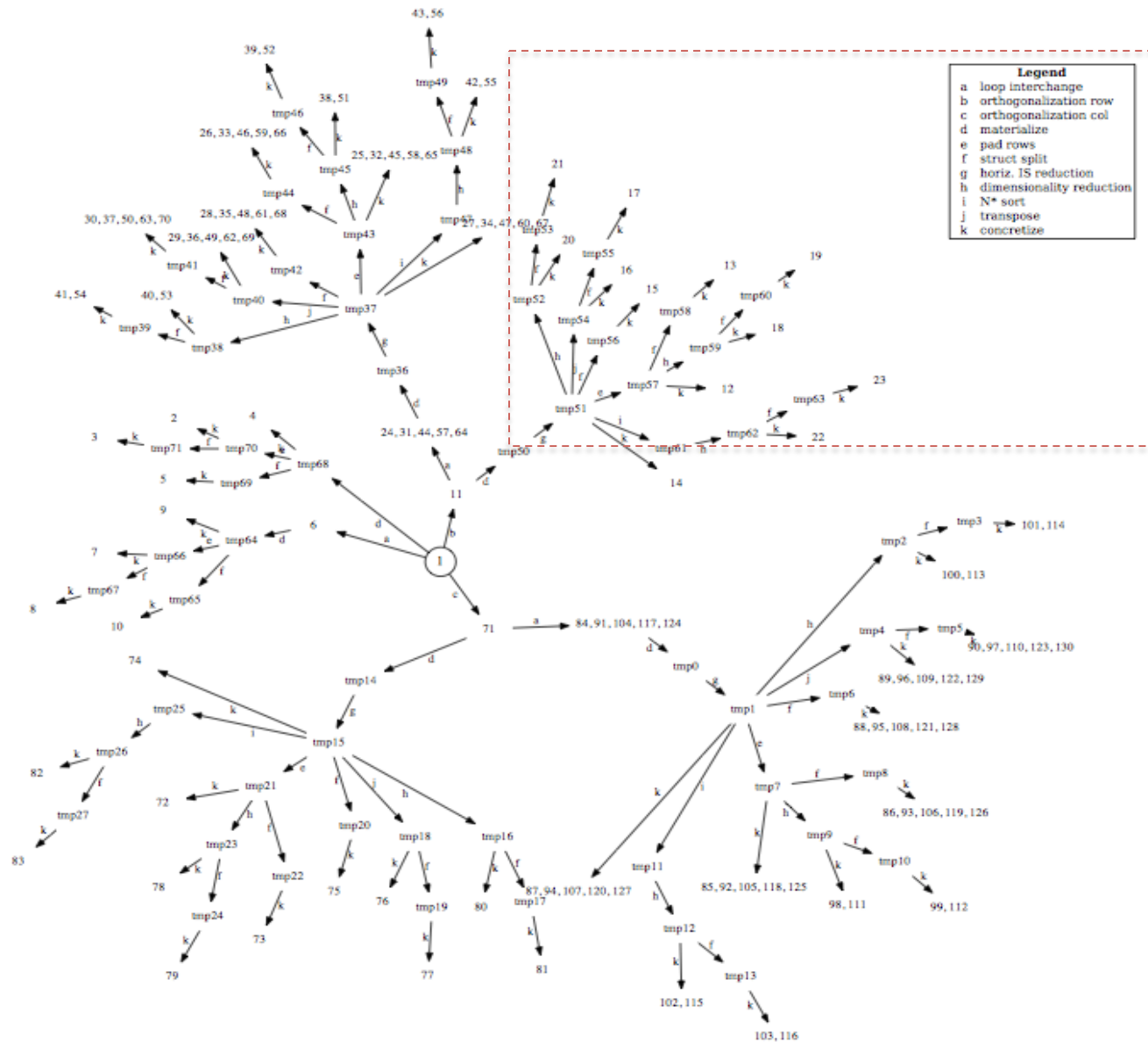
**Just one array of field_B
values**

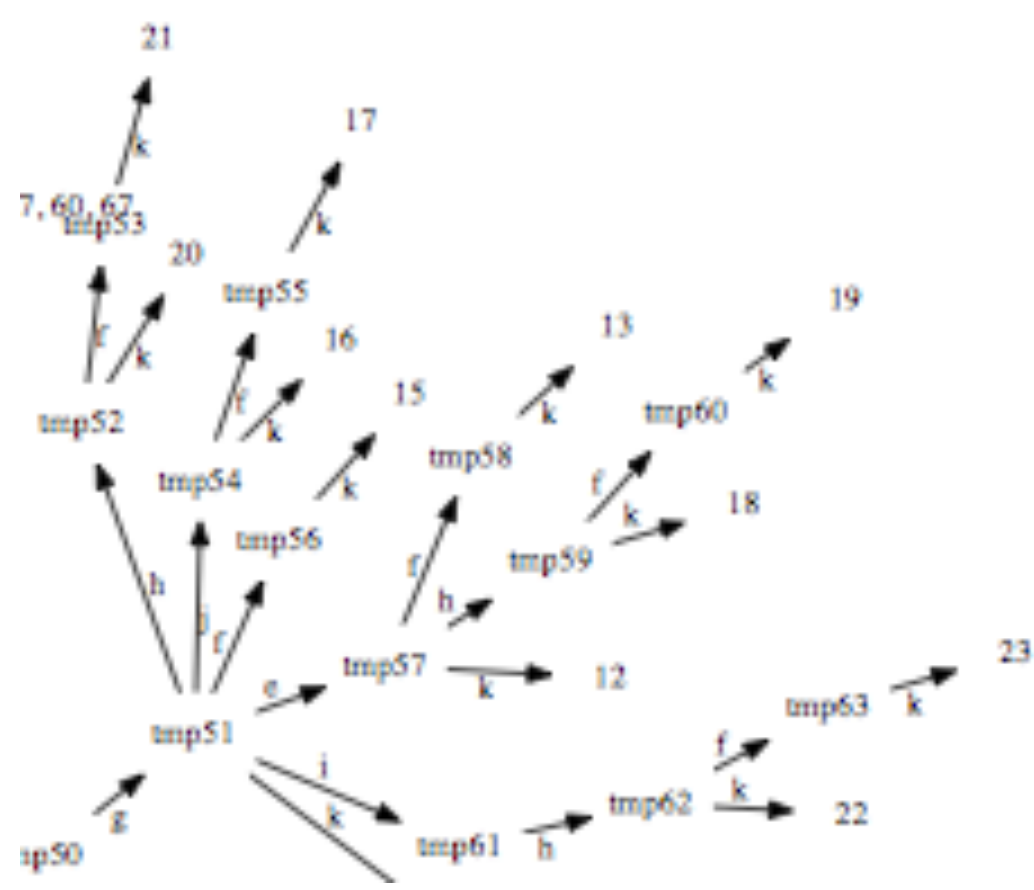
The automatic generation of ITPACK data structure





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:

$(t_k, \text{Serial_Code_}i)^*$

representing

$\text{Serial_Code_}i (< t_k >)$

*Note that $\text{Cond_}i$ 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:

$(t_1, \text{Serial_Code}_i), (t_2, \text{Serial_Code}_i), \dots$

- **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

$(t_1, \text{Serial_Code}_i), (t_2, \text{Serial_Code}_i), \dots,$
 $(t_k, \text{Serial_Code}_j), (t_{k+1}, \text{Serial_Code}_j), \dots$

- **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 $C = C_1 \parallel C_2$ 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:

$(t_1, \text{Serial_Code_i}), (t_2, \text{Serial_Code_j})$

have a data dependence on an address function

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

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

What about `forelem` loop structures?

There are no Read on Write Dependencies, and/or Write on Read Dependencies in a **`forelem`** loop

BUT

there can be input and output dependencies in **`forelem`** loops

SO

Profit chains can only be based on read on read dependencies or write on write dependencies

Profit Chains in SpMxV

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

($\langle 1,1 \rangle$, Serial_Code_1), ($\langle 1,2 \rangle$, 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 $(t_k, \text{Serial_Code_}i)$ there is an i such that

$$(t_k, \text{Serial_Code_}i) \in C_i$$

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 $(t_k, \text{Serial_Code_}i)$ pairs which were not covered, to obtain a covering chain set.

Main Theorem II

If

whilelem ($t; t \varepsilon T$)

is just scheduled, then if

whilelem ($C; C \varepsilon CCS$)

forelem ($t; t \varepsilon C$)

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

forelem ($t; t \varepsilon T$)

and

forelem ($C; C \varepsilon CCS$)

forelem ($t; t \varepsilon C$)

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 ∈ 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)
    {
      ...
      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
      }
    }
  }
}
```

Serial_Code_1

Serial_Code_2

Then $(\langle s,4,6 \rangle, \text{Serial_Code_1}), (\langle 4,6,52 \rangle, \text{Serial_Code_1}), \dots,$
 $(\langle 100,105,107 \rangle, \text{Serial_Code_1}), (\langle 105, 107,111 \rangle, \text{Serial_Code_2}),$
 $(\langle 111,107, 105 \rangle, \text{Serial_Code_1}), \dots (\langle 6,4,s \rangle, \text{Serial_Code_1})$ with
 $\text{Remainder}[4, 6] > 0$, **with** $\text{Remainder}[6, 52] > 0$, ...
etc., and $\text{Remainder}[107, 111] == 0$ is a profit chain.

As well as

$(\langle s,4,6 \rangle, \text{Serial_Code_1}), (\langle 4,6,52 \rangle, \text{Serial_Code_1}), \dots,$
 $(\langle 100,105,107 \rangle, \text{Serial_Code_1}), (\langle 105, 107, t \rangle, \text{Serial_Code_1}),$
with $\text{Remainder}[4, 6] > 0$, **with** $\text{Remainder}[6, 52] > 0$, ...
etc.

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