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.

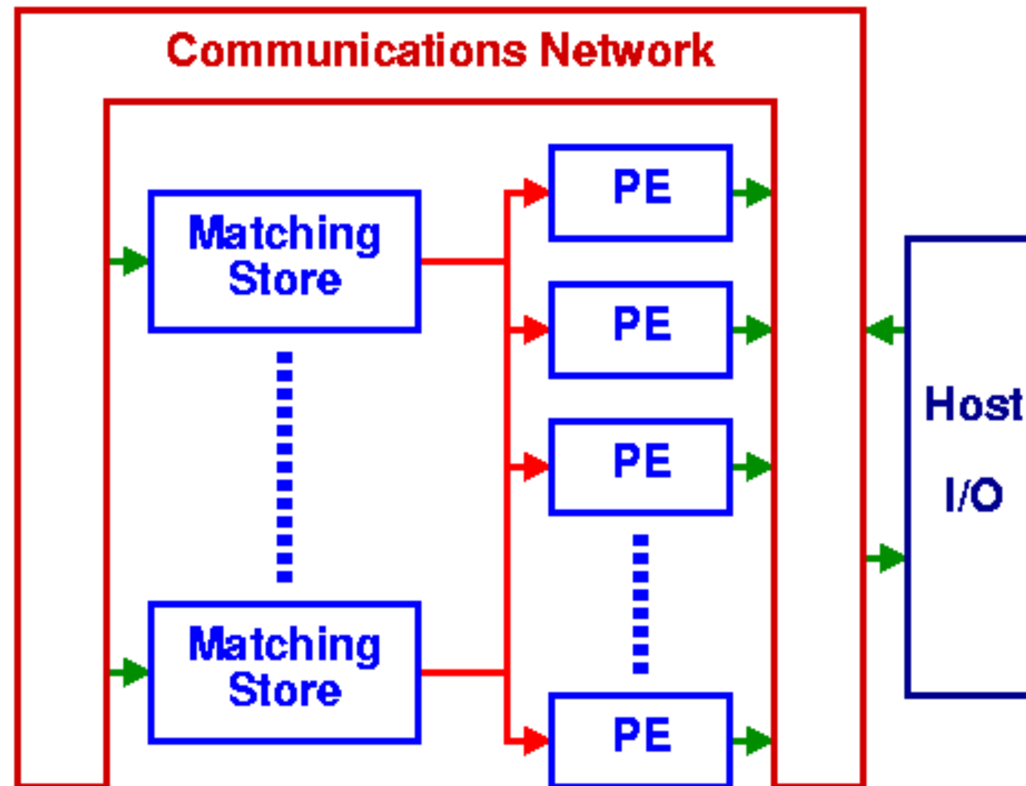
A dataflow compiler records these dependencies by creating unique tags for each dependency instead of using variable names. By giving each dependency a unique tag, it allows the non-dependent code segments in the binary to be executed *out of order* and in parallel.

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

$< \text{token} >_A$

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(ddr)

A(ddr) can be any invertible function, but mostly it is an affine function:

$$Z^n \to Z^k$$

So Addr 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 at location @A [I, J], or that
the data value of $< I, J >_A$ becomes 5.0, or that
$< I, J, data > = < I, J, 5.0 >$*

* Note that tokens can be more dimensional: token tuples t
In this case t.i represents the ith field of 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

tUPL Loop Bodies: Two Cases

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

`Serial_Code_i` $(< t_k >)$ and
`Serial_Code_j` $(< t_m >)$

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

`Serial_Code_i` $(< t_k >)$ and
`Serial_Code_j` $(< t_m >)$

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 (< t`$_k$` >)`

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

  `Serial_Code_j (< t`$_m$` >)`
- ➔ Use of

  `forelem` `( t; t ε T )`
- ➔ As a result each `Serial_Code_i (< t`$_k$` >)` 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 ε 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 ε 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 ε 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 ε T )
{
    if ( ready(q)[q < t] )
    {
        …
    }
}
```

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

```
for (k; kεN+100)*
    {
        …
    }
```

* $k \varepsilon \mathbf{N}^{+}_{100}$ is used for `kεT={<1>,<2>, …,<100>}`,
$k \varepsilon \mathbf{N}_{100}$ is used for `kεT={<0>,<1>, …,<100>}`

Case IV

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

```
whilelem ( q; q ε 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 : $X[t_1]$ = …

for tuple t_2 : y = … $X[t_2]$ …

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 = … X[t] … ; $Old_X[t] = X[t]

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

$X[t_1] = 5; y = X[t_2];$ # $@X[t_1]==@X[t_2]$

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

Different Execution Order

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 ($X[t_1]$!= \$Old_$X[t_1]$) $X[t_1] = 5$; \$Old_$X[t_1] = X[t_1]$
while ($X[t_2]$!= \$Old_$X[t_2]$) $X[t_2] = 10$; \$Old_$X[t_2] = X[t_2]$
while ($X[t_1]$!= \$Old_$X[t_1]$) $X[t_1] = 5$; \$Old_$X[t_1] = X[t_1]$
while ($X[t_2]$!= \$Old_$X[t_2]$) $X[t_2] = 10$; \$Old_$X[t_2] = X[t_2]$
……

Etc. etc.

➔ **RACE CONDITION**

Example

```
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;

whilelem ( t; t ε 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 <1,3> : **X[1]** = (B[1]-X[3]*A[1,3])/A[1,1]

<2,1> : 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 ε 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

```
<1,3> # X[t.j] = X[3] = 0 and $Old_X[t] = NULL
        X[1] = (8-X[3]*2)/6 = 8/6 = 1.333
        $Old_X[<1,3>] = 0.000
<2,1> # 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[<2,1>] = 1.333
<3,2> # 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[<3,2>] = 2.067
<1,3> # 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[<1,3>] = 1.466
<2,1> # 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[<2,1>] = 0.845
<3,2> # 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[<3,2>] = 1.969
<1,3> # 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[<1,3>] = 1.015
<2,1> # 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[<2,1>] = 0.995
<3,2> # 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[<3,2>] = 1.999
<1,3> # 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[<1,3>] = 1.000
<2,1> # 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[<2,1>] = 1.000
<3,2> # 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[<3,2>] = 2.000
```

OR

```
<2,1> # X[t.j] = X[1] = 0 and $Old_X[t] = NULL
        X[2] = (9-X[1]*-1)/5 = 9/5 = 1.800
        $Old_X[<2,1>] = 0.000
<1,3> # X[t.j] = X[3] = 0 and $Old_X[t] = NULL
        X[1] = (8-X[3]*2)/6 = 8/6 = 1.333
        $Old_X[<1,3>] = 0.000
<2,1> # 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[<2,1>] = 1.333
<1,3> # X[t.j] = X[3] = 0 and $Old_X[t] = 0.000
        NOP
<2,1> # X[t.j] = X[1] = 1.333 and $Old_X[t] = 1.333
        NOP
<3,2> # 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[<3,2>] = 2.067
<1,3> # 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[<1,3>] = 1.466
<3,2> # X[t.j] = X[2] = 2.067 and $Old_X[t] = 2.067
        NOP
<2,1> # 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[<2,1>] = 0.845
… …
```

Otherwise similar (except for interleaving with NOPs)

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

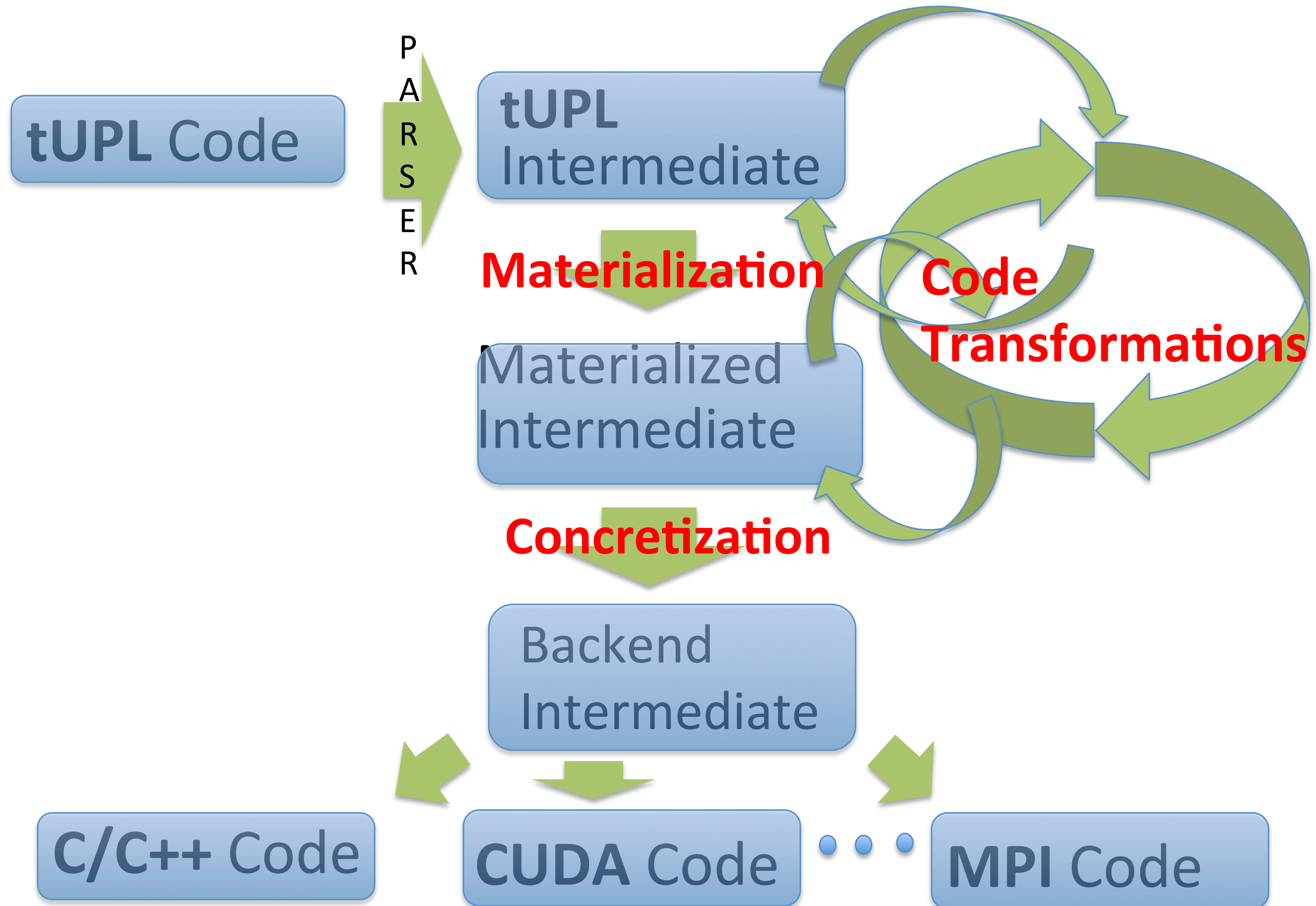
- For each execution of a tuple exactly one of the tuples with a valid conditional serial code is chosen.
- If there are no tuples left with a valid conditional serial code, then the **`whilelem`** loop terminates.
- Any loop scheduling for a **`whilelem`** loop must guarantee that every tuple with a 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.

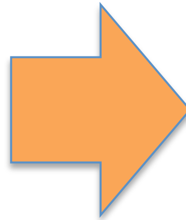
Note that if the conditions are not carefully chosen it can happen that the **`forelem`** loop terminates before all tuples have been executed.

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

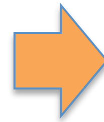
Tuple Selectors

- `(i;i ε 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.
- $\texttt{IDX_T}_{\texttt{<row,col>[<10,100>]}}\texttt{()}$ 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 ε pA)
    … A[i]…
```

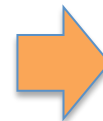


```
forelem (ii; ii ε A.field1)
    forelem (i; i ε pA.field1[ii])
        … A[i]…
```

`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 $\mathbf{N}_{10}$)

… …

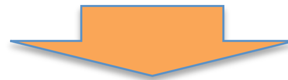
If `A.field1` would be { 0, 1, 3, 4, 7, 9, 10 }, for instance. This transformation only makes sense, if the execution of the inner loop for the other i-value's results into a NOP. i.e. C[i] = C[i] + B[i], and B[i] == 0 for 2, 5, 6 and 8.

*`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)

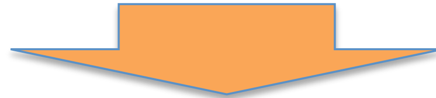
Loop Interchange

Horizontal Iteration Space Reduction

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 }

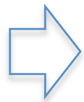
DO NOT CONFUSE PA with a linear array data structure

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 …
```

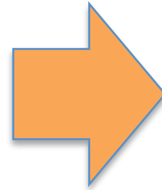
```
forelem (i; i ε N10)
    forelem (k; k ε N*)
        … B[i].value[k] …
```

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)

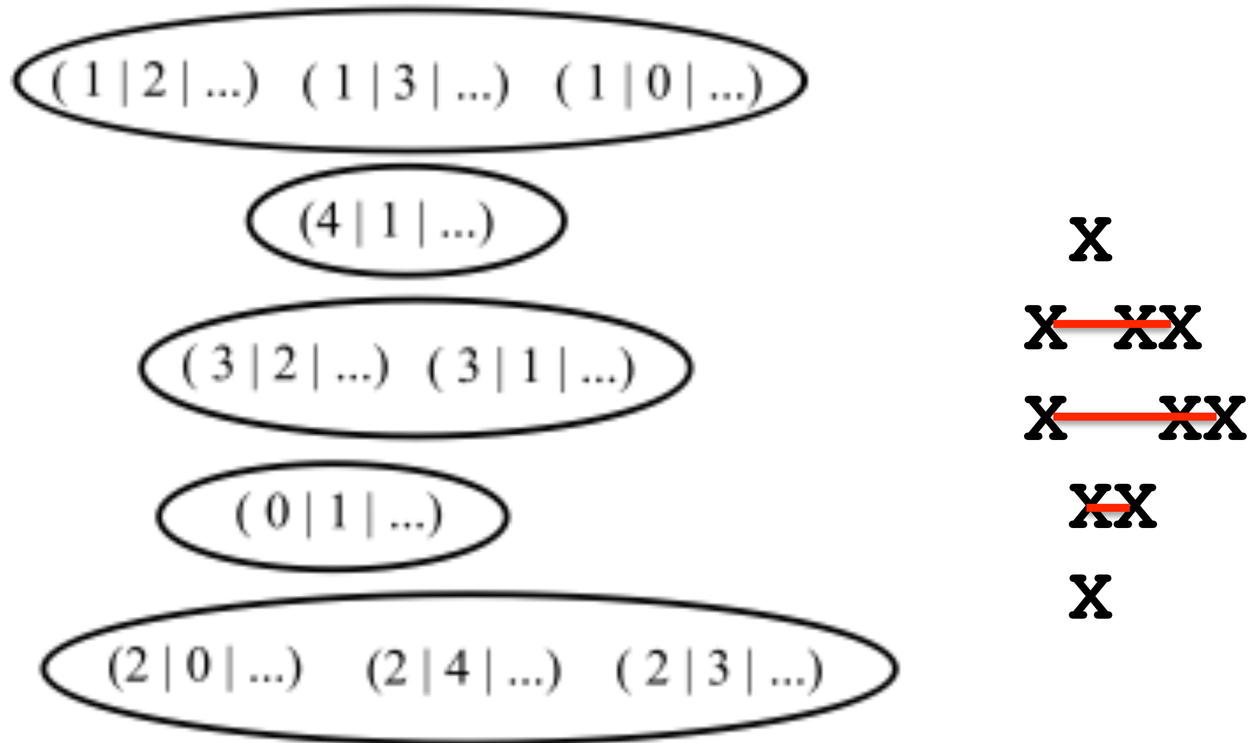
Data Localization

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

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

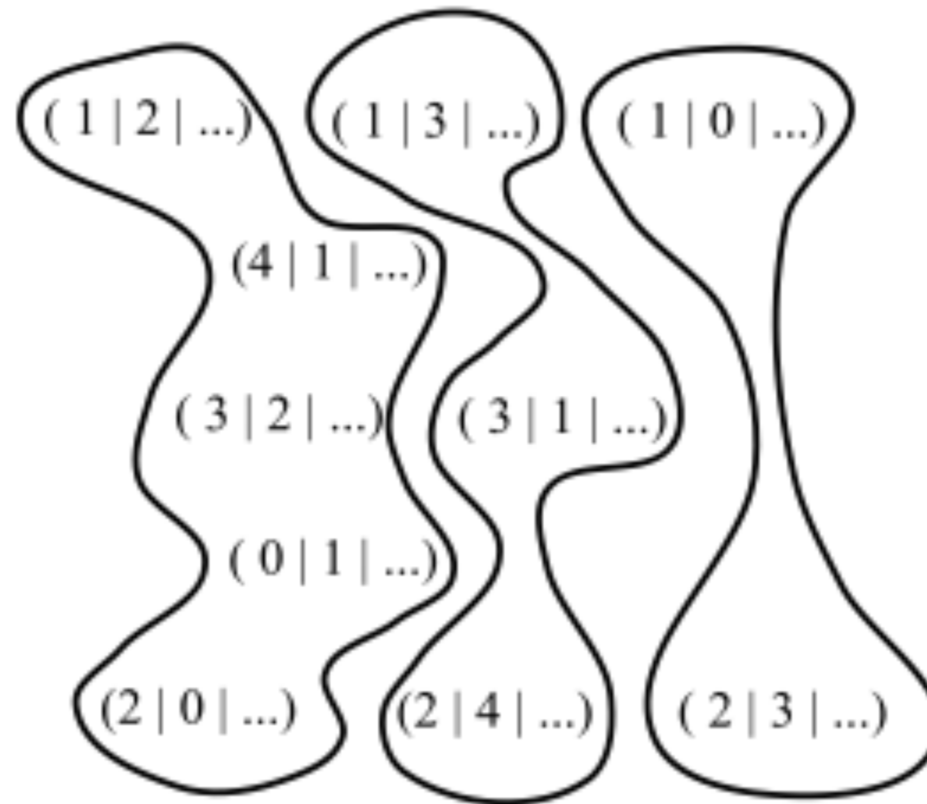
Here the tuples in reservoir A are being extended to include the data at address @`B[A[i].field_k}`. So A' = { < t, B[t] > | t ε A }. By default, this transformation is only allowed for read only data at B.

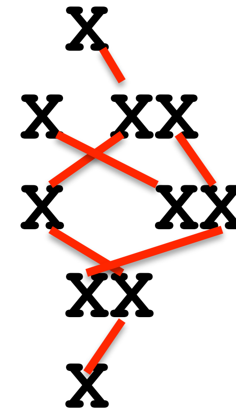
Regrouping of Single Storage Locations (Tuples)



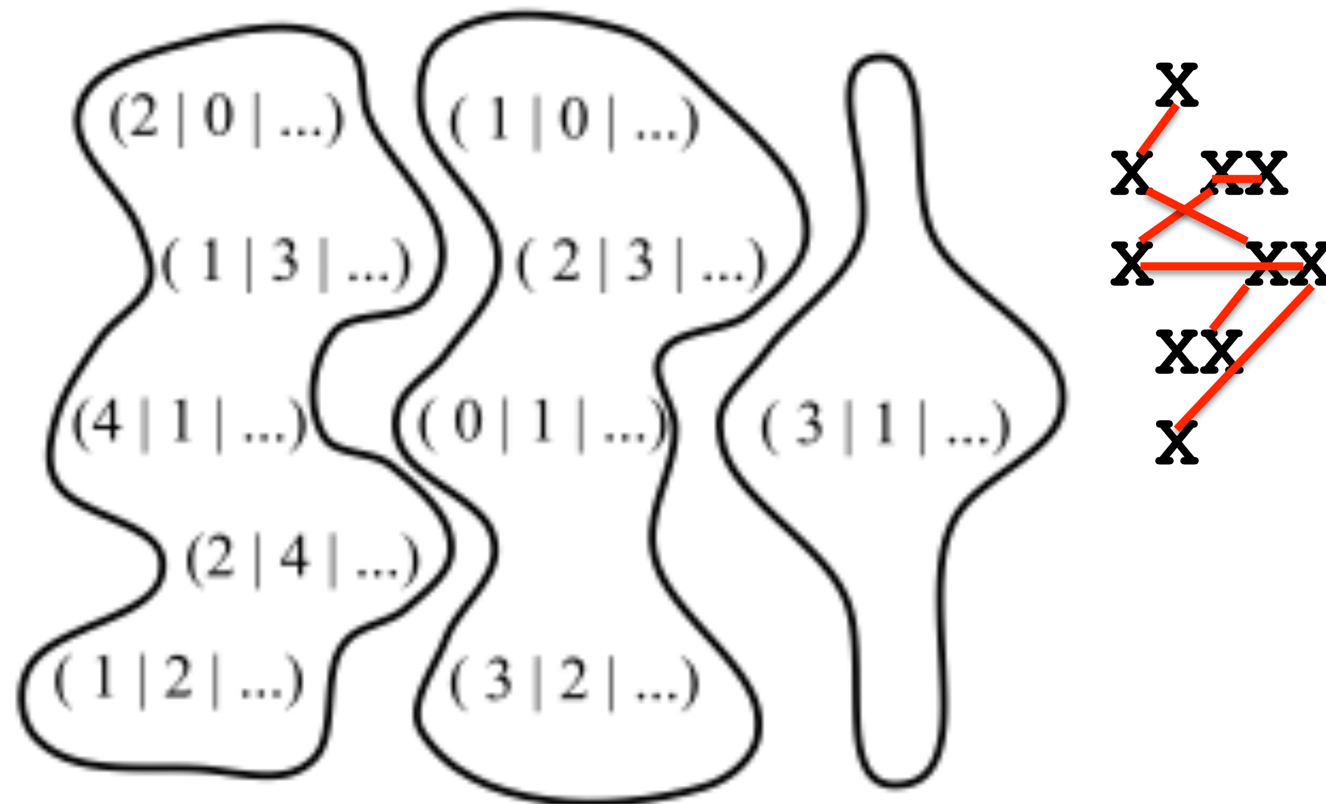
Regrouping as a result of orthogonalization on the first field

Regrouping after Materialization and Loop Interchange



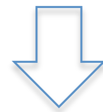


Regrouping after orthogonalization on the second field followed by materialization and loop interchange

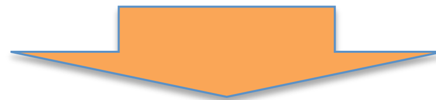


Concretization

```
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]…
```

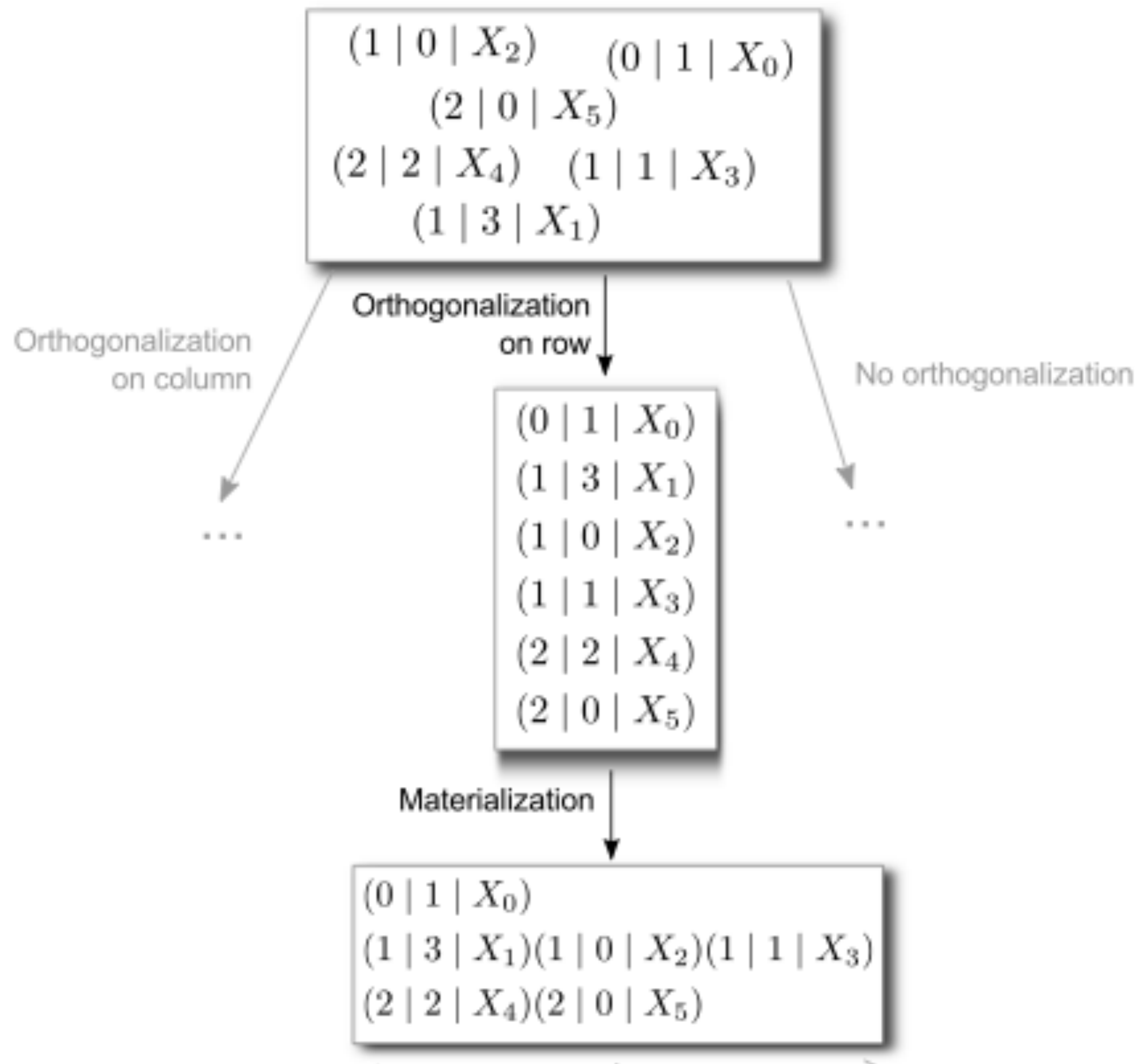
Several Arrays for each field of A'

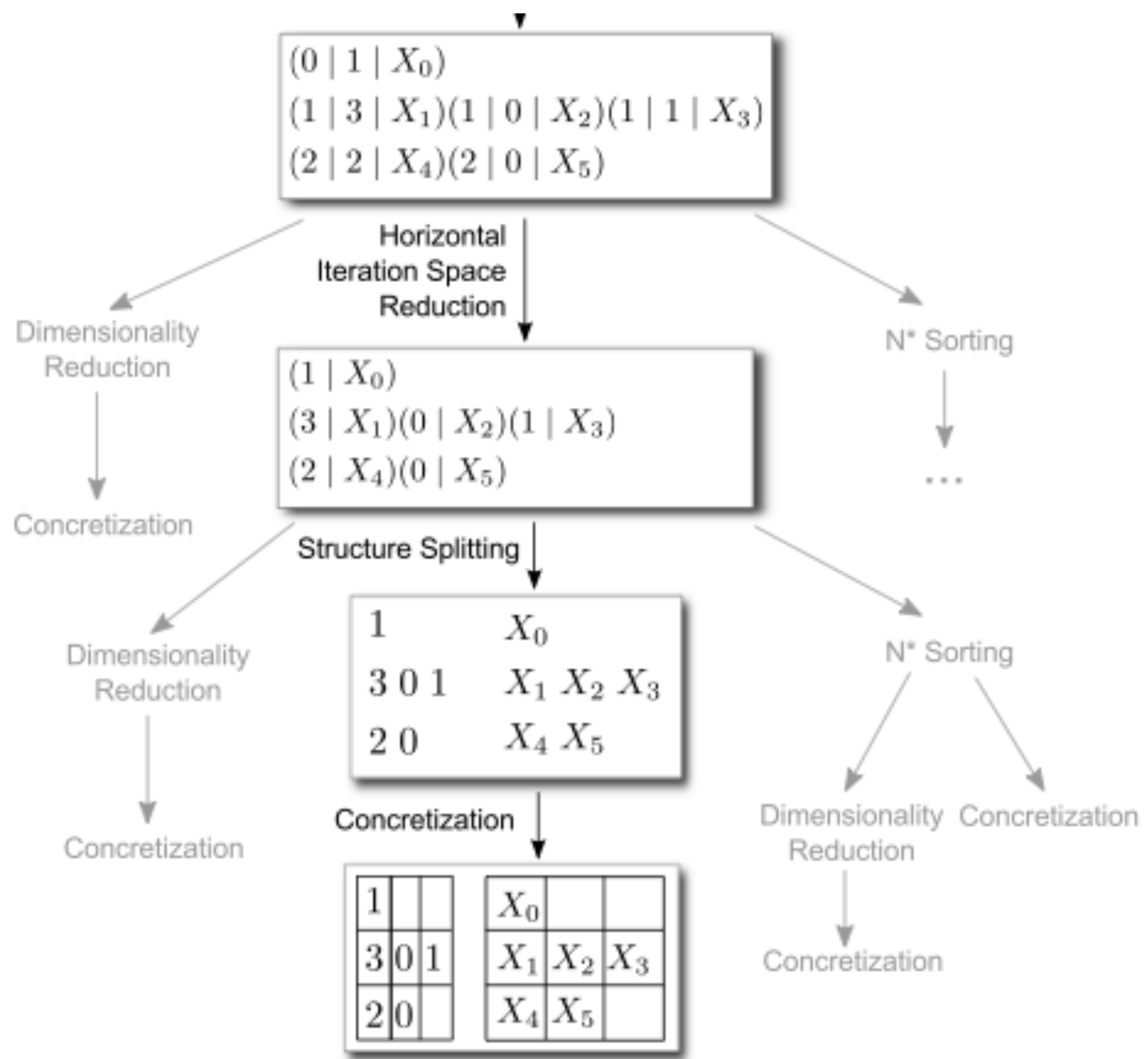
Horizontal Iteration Space Reduction

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

Just one array of field_B values

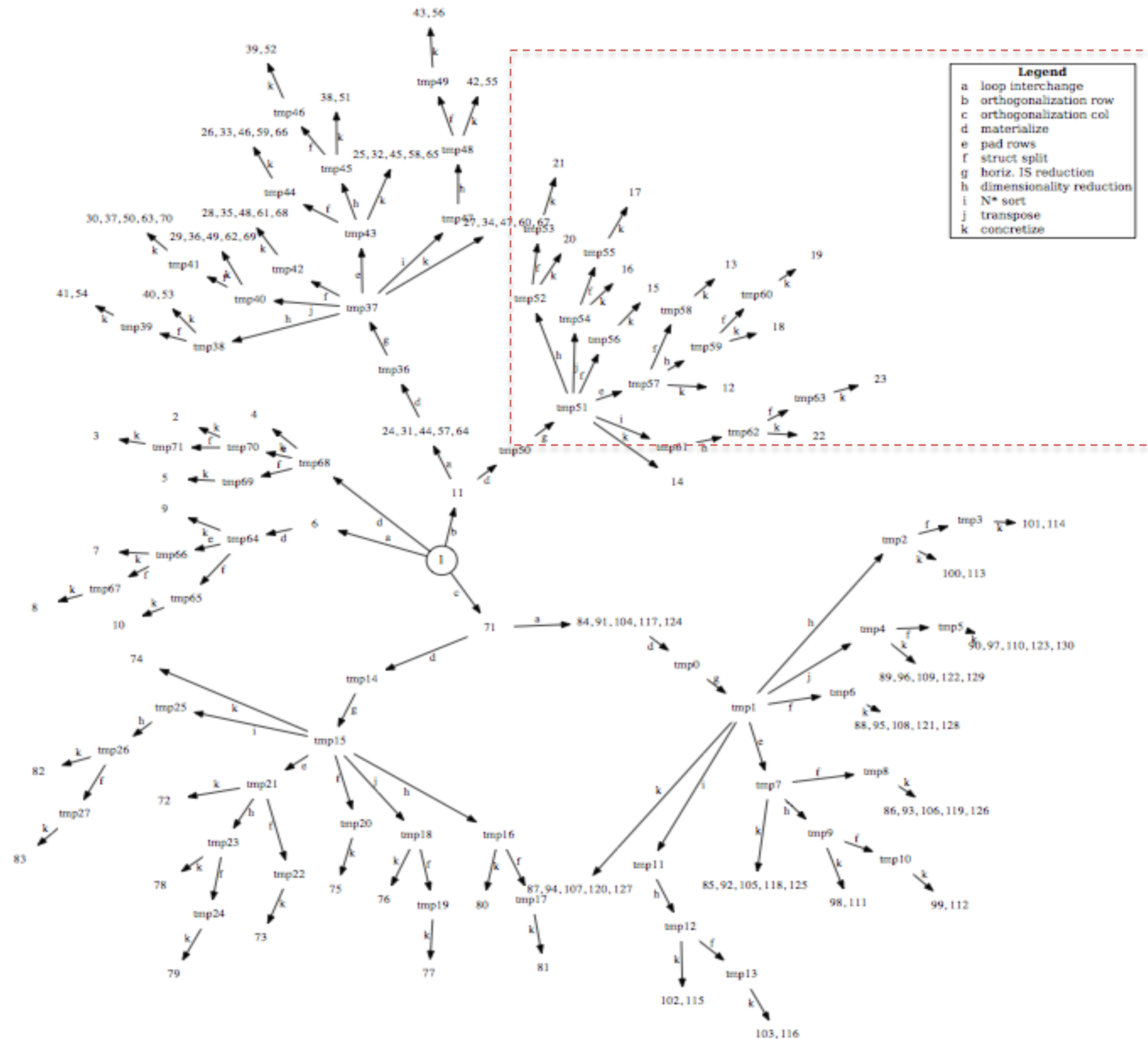
The automatic generation of ITPACK data structure

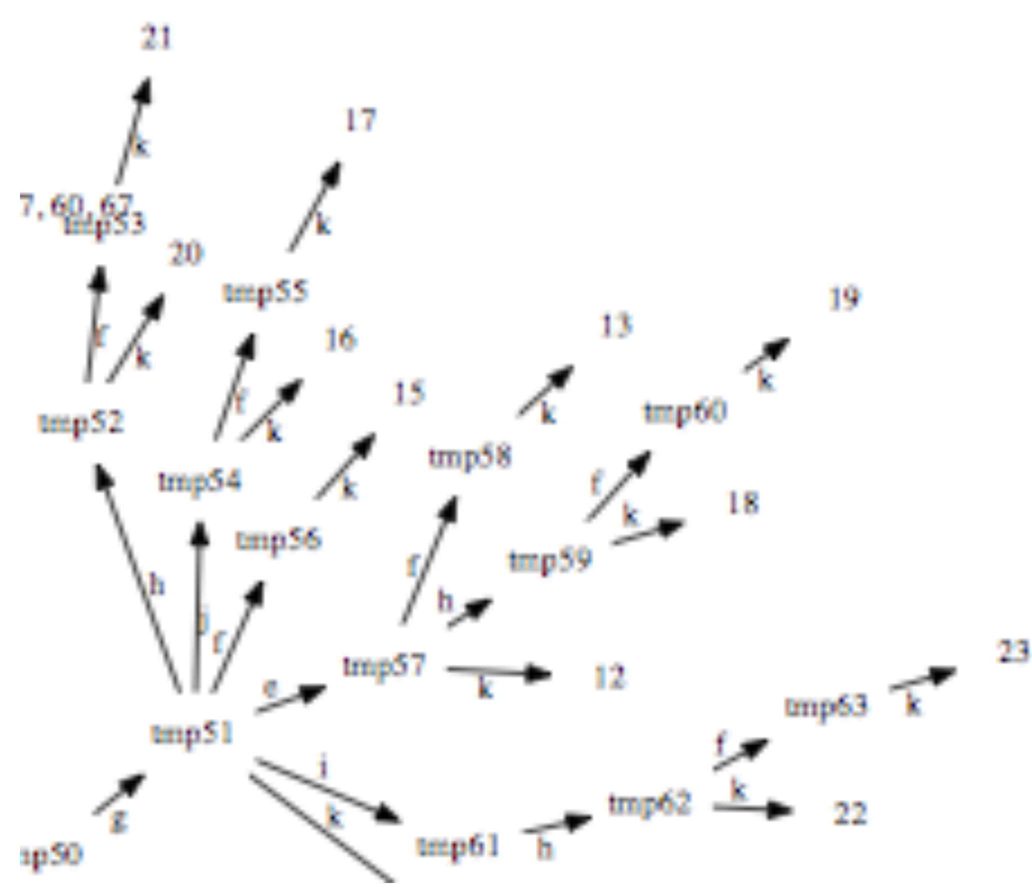




(0 | 1 | X_0)
(1 | 3 | X_1)(1 | 0 | X_2)(1 | 1 | X_3)
(2 | 2 | X_4)(2 | 0 | X_5)
Horizontal Iteration Space Reduction
Dimensionality Reduction
Concretization
N* Sorting
...
(1 | X_0)
(3 | X_1)(0 | X_2)(1 | X_3)
(2 | X_4)(0 | X_5)
Structure Splitting
Dimensionality Reduction
Concretization
N* Sorting
1 X_0
3 0 1 X_1 X_2 X_3
2 0 X_4 X_5
Concretization
Dimensionality Reduction
Concretization
Concretization

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

END OF COURSE

Parallel Programming II (this fall)

- **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.