Uninterpreted Functions: Their use in data dependence analysis

Michelle Mills Strout (University of Arizona)

THE UNIVERSITY OF ARIZONA

Example: Sparse Triangular Solve

Sparse Computation

```
for (i=0; i<N; i++) {
  u[i] = f[i];
  for (j=rowptr[i]; j<diag[i]; j++){
    x[i] = x[i] - A[j]*x[col[j]];
  }
  u[i] = u[i] / A[diag[i]];
}
```

CHiLL Script

```
level_set() = wave-par(<i loop>)
```

CHiLL-I/E compiler

Compile time

Index Arrays

INSPECTOR

(1) Create dependence DAG

(2) Find level sets, or wavefronts

Explicit Functions

EXECUTOR

```
for (l=0; l<M; l++){
  #omp parallel for
  for (i in level_set(l)){
    u[i] = f[i];
    for (j ...
```

Run time

Data Dependence Analysis is Challenging

- Need data dep analysis for FULL parallelism and PARTIAL
- Array indexing and/or loop bounds are non-affine
- Loop `i` marked as sequential by current compilers

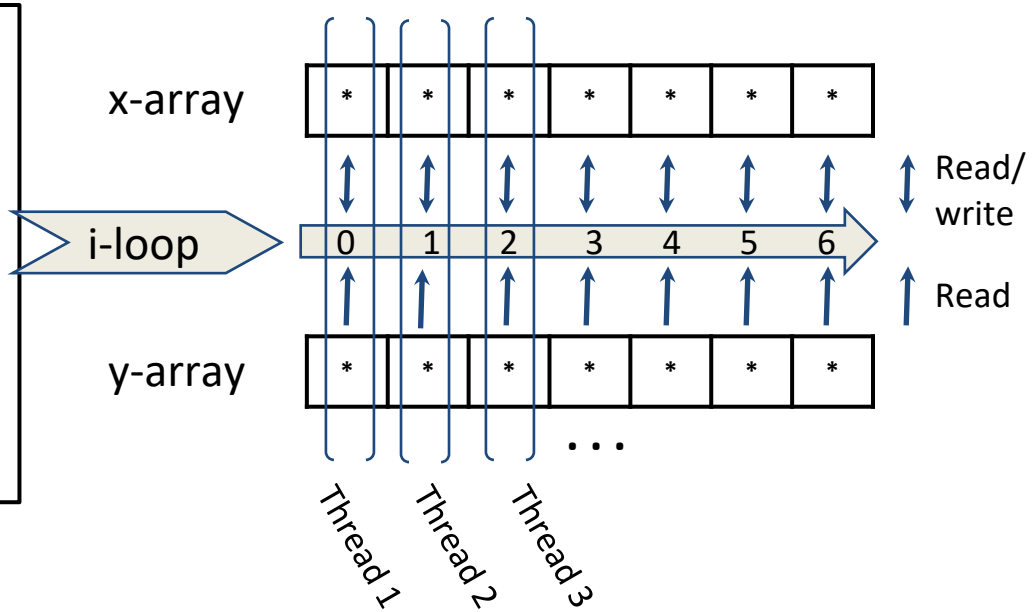
Sparse Triangular Solve Code

```
for (i=0; i<n; i++) {
  u[i] = f[i];
  for (j = rowptr[i]; j < diag[i]; j++){
    u[i] -= A[j]*u[ col[j] ];
  }
  u[i] = u[i] / A[ diag[i] ];
}
```

Data Dependence Analysis in Regular/Dense Codes

- Affine index expressions can be analyzed for dependences at compile time

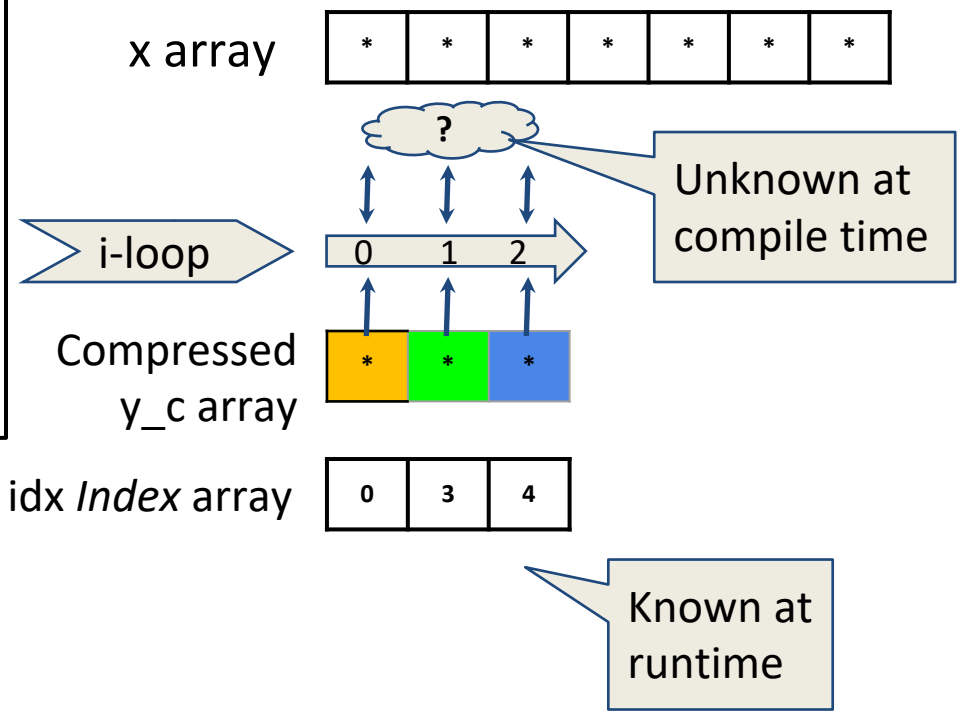
```
// dense vector add
#pragma omp parallel for
for (i=0; i<n; i++) {
    x[i] = x[i] + y[i];
}
```



Data Dependence Analysis of Sparse Computations

- Affine index expressions can be analyzed for dependences at compile time

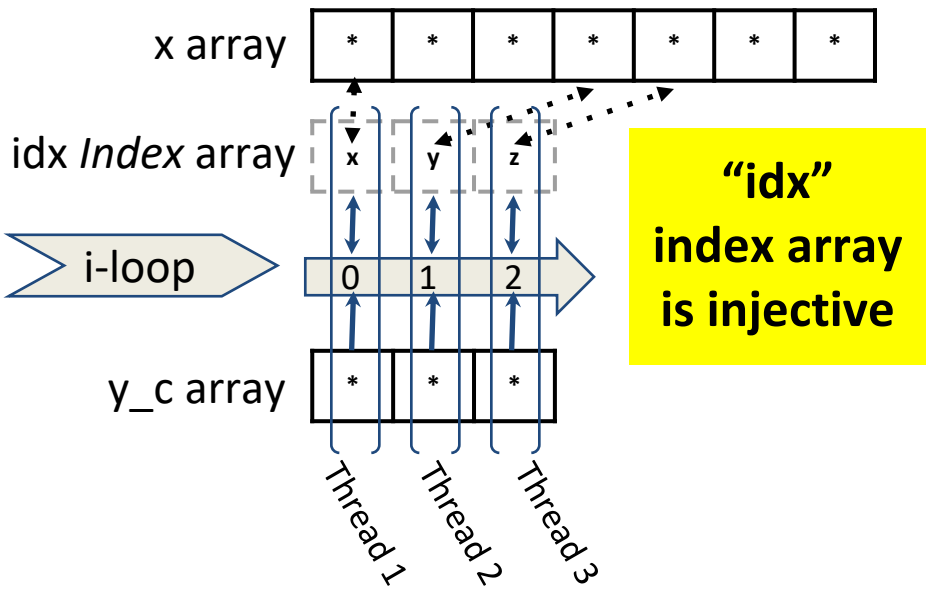
```
// sparse + dense vector add
for (i=0; i<nnz_y; i++){
  x[idx[i]] = x[idx[i]]
              + y_c[i];
}
```



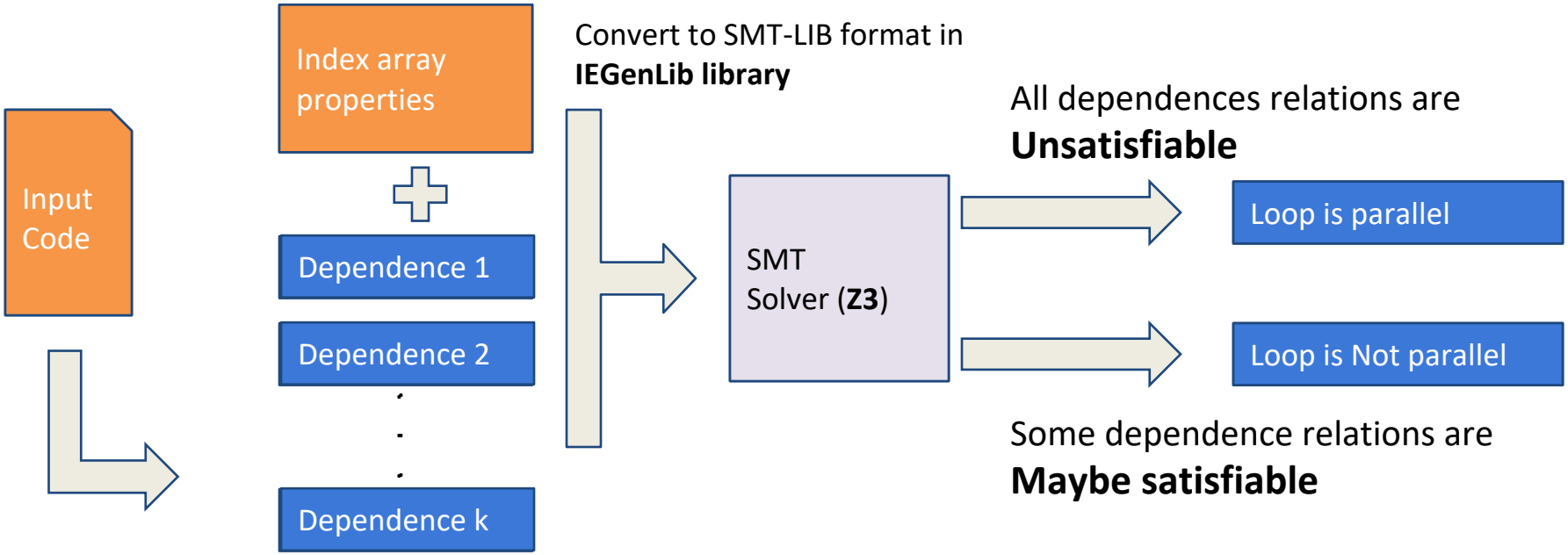
Approach: Use Index Array Properties

- Using index array properties such as injectivity and/or monotonicity make compile-time analysis more precise

```
// sparse + dense vector add
#pragma omp parallel for
for (i=0; i<nnz_y; i++){
  x[idx[i]] = x[idx[i]]
              + y_c[i];
}
```



Automatically Utilizing Variety of Index Array Properties to Find Parallelism

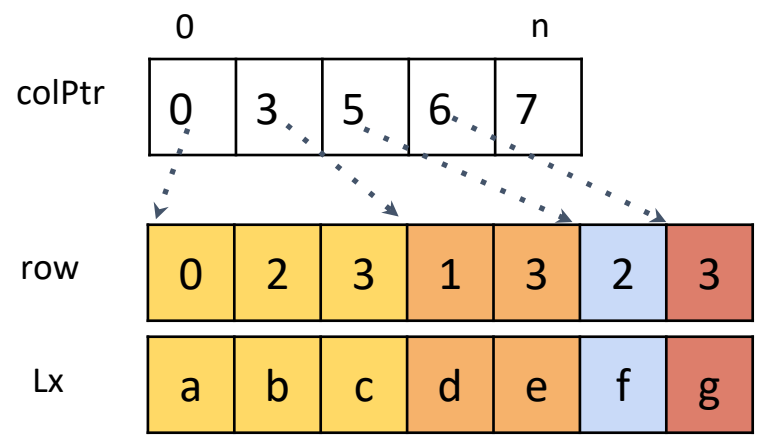


Extracting Constraint based dependencies for a loop in **CHILL compiler framework**.

Universally Quantified Assertions

- Index arrays are represented as uninterpreted functions
- Index-array properties as universally quantified assertions

Lx	0	1	2	3
0	a			
1		d		
2	b		f	
3	c	e		g

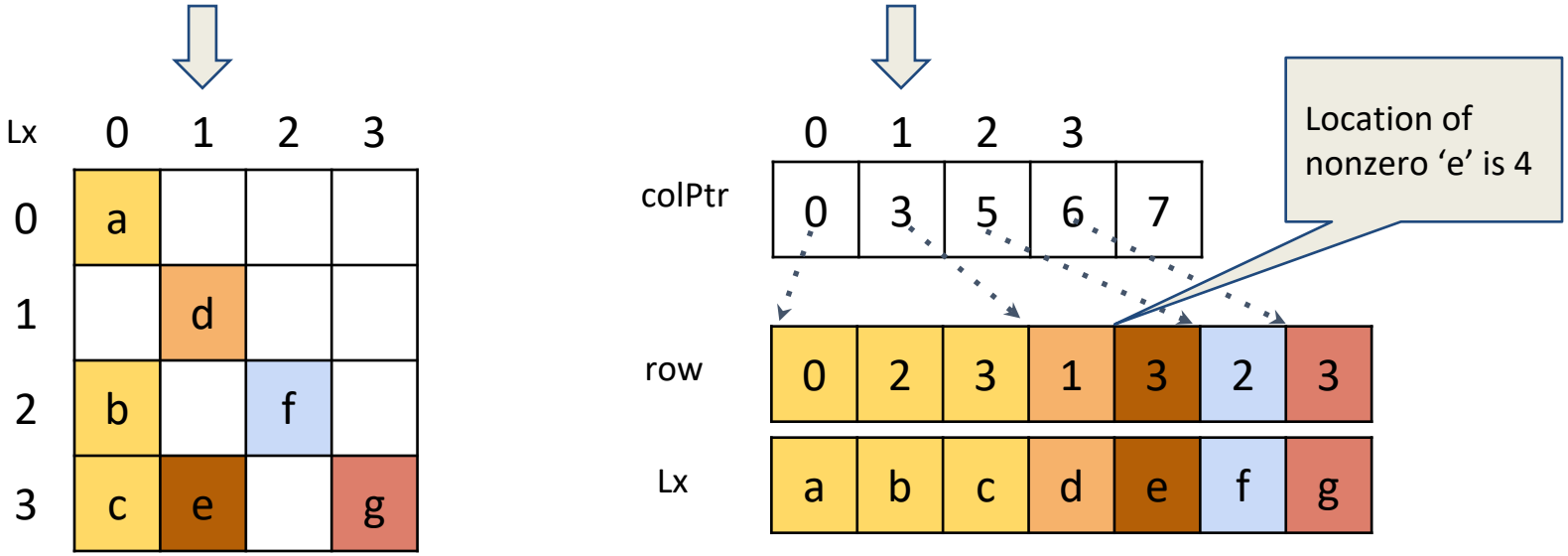


Monotonically Strictly Increasing:

$(\forall\, x_1, x_2)(x_1 < x_2 \Leftrightarrow colPtr(x_1) < colPtr(x_2))$.

Functional Consistency
Domain and Range
Injectivity
Periodic Monotonicity

Index-Array Properties as Universally Quantified Assertions



Lower Triangularity:

$(\forall\, x_1, x_2)(\ colPtr(x_1) < x_2 \Rightarrow x_1 < row(x_2)\ ).$

$colPtr(1) < 4 \Rightarrow 1 < row(4)$

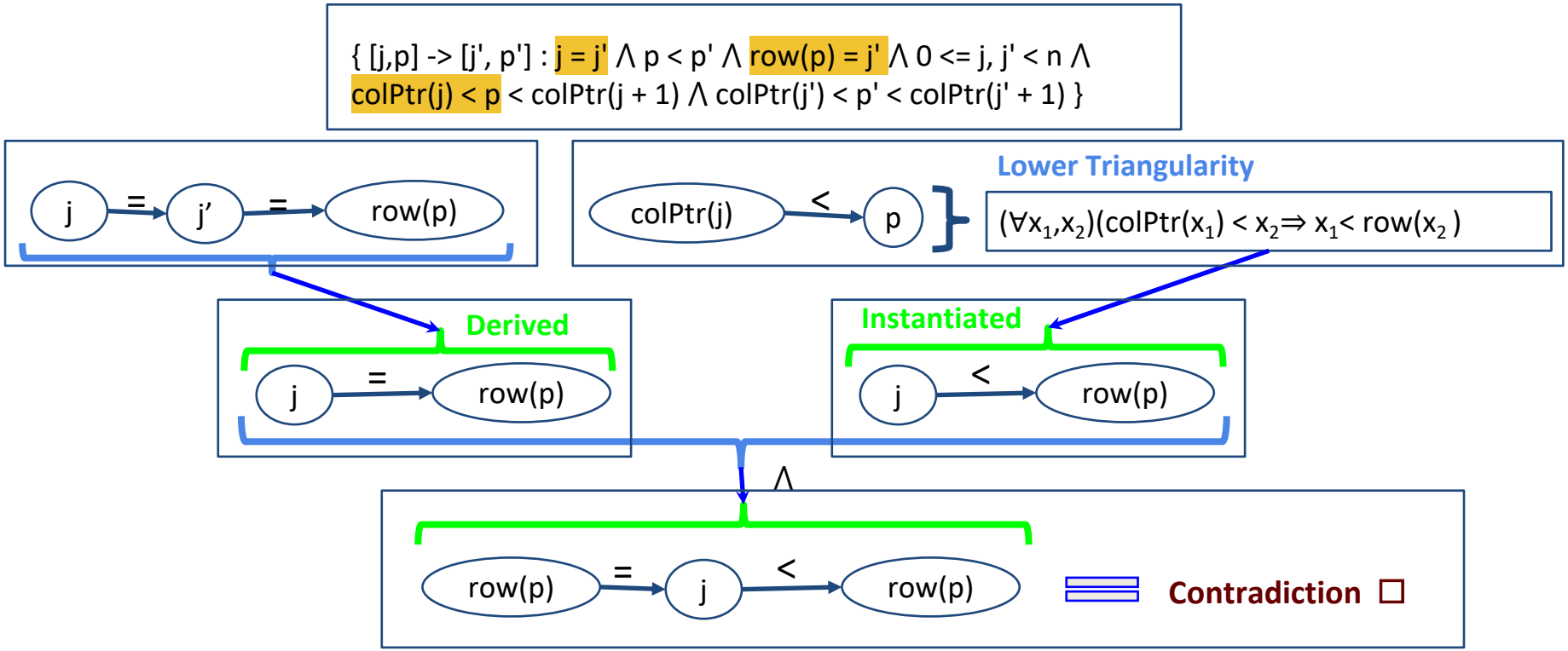
$3 < 4 \Rightarrow 1 < 3$

Constraint-Based Data Dependence Analysis of Sparse Computation

```
for (int j=0; j<n; j++){
  x[j] = x[j] / Lx[colPtr[j]];
  for(int p=colPtr[j]+1; p<colPtr[j+1]; p++){
    x[row[p]] = x[[row[p]] - Lx[p] * x[j];
```

$$\{[j,p] \rightarrow [j',p'] : \overbrace{j = j' \wedge p < p'}^{\textit{lexicographical Ordering}} \wedge \overbrace{row(p) = j'}^{\textit{Array Access Equality}} \wedge$$
$$\underbrace{0 \leq j, j' < n \wedge colPtr(j) < p < colPtr(j+1) \wedge colPtr(j') < p' < colPtr(j'+1)}_{\textit{Loop Bounds}}\}$$

Example: Using Triangularity Property to Show Dependence is UNSAT



Summary for Sparse Data Dependence Analysis

- Represent index arrays with uninterpreted functions
- Encode index array properties as universally quantified assertions
- Leverage SMT solvers to find ***full*** parallelism when possible and to simplify ***partial*** parallelism
- See Mahdi Soltan Mohammadi's poster for more details