Uninterpreted Functions: Their use in data dependence analysis

Michelle Mills Strout (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

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

INSPECTOR
(1) Create dependence DAG
(2) Find level sets, or wavefronts

Explicit Functions

Index Arrays
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

```c
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

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

Diagram showing data dependence with arrows indicating read/write operations.
Data Dependence Analysis of Sparse Computations

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

```c
// 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

```c
// 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

Monotonically Strictly Increasing:
\[(\forall x_1, x_2)(x_1 < x_2 \Leftrightarrow \text{colPtr}(x_1) < \text{colPtr}(x_2)).\]

Functional Consistency
Domain and Range
Injectivity
Periodic Monotonicity
Index-Array Properties as Universally Quantified Assertions

Lower Triangularity:

\[(\forall x_1, x_2)(\text{colPtr}(x_1) < x_2 \Rightarrow x_1 < \text{row}(x_2))\].

Location of nonzero ‘e’ is 4

\[\text{colPtr}(1) < 4 \Rightarrow 1 < \text{row}(4)\]

\[3 < 4 \Rightarrow 1 < 3\]
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];
    }
}
Example: Using Triangularity Property to Show Dependence is UNSAT

\[ \{ [j, p] \rightarrow [j', p'] : j = j' \land p < p' \land \text{row}(p) = j' \land 0 \leq j, j' < n \land \text{colPtr}(j) < p < \text{colPtr}(j + 1) \land \text{colPtr}(j') < p' < \text{colPtr}(j' + 1) \} \]
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