Lecture on Parallelization

I. Basic Parallelization
II. Data dependence analysis
III. Interprocedural parallelization

Chapter 11.1-11.1.4

Parallelization of Numerical Applications

- DoAll loop parallelism
  - Find loops whose iterations are independent
  - Number of iterations typically scales with the problem
  - Usually much larger than the number of processors in a machine
  - Divide up iterations across machines

Basic Parallelism

Examples:

```
FOR i = 1 to 100
    A[i] = B[i] + C[i]

FOR i = 11 TO 20
    a[i] = a[i-1] + 3

FOR i = 11 TO 20
    a[i] = a[i-10] + 3
```

- Does there exist a data dependence edge between two different iterations?
- A data dependence edge is loop-carried if it crosses iteration boundaries
- DoAll loops: loops without loop-carried dependences

Recall: Data Dependences

- True dependence:

```
    a =
  = a
```

- Anti-dependence:

```
       = a
a =
```

- Output dependence

```
    a =
    a =
```

Affine Array Accesses

- Common patterns of data accesses: (i, j, k are loop indexes)

  `A[i], A[j], A[i-1], A[0], A[i+j], A[2*i], A[2*i+1]`

  `A[i,j], A[i-1, j+1]`

- Array indexes are affine expressions of surrounding loop indexes
  - Loop indexes: $i_n, i_{n-1}, \ldots, i_1$
  - Integer constants: $c_n, c_{n-1}, \ldots, c_0$
  - Array index: $c_n i_n + c_{n-1} i_{n-1} + \ldots + c_1 i_1 + c_0$
  - Affine expression: linear expression + a constant term (c_0)

II. Formulating Data Dependence Analysis

```
FOR i := 2 to 5 do
    A[i-2] = A[i]+1;
```

- Between read access `A[i]` and write access `A[i-2]` there is a dependence if:
  - there exist two iterations i_r and i_w within the loop bounds, s.t.
  - iterations i_r & i_w read & write the same array element, respectively

$$\exists \text{ integers } \quad i_w, i_r \qquad 2 \le i_w, i_r \le 5 \qquad i_r = i_w - 2$$

- Between write access `A[i-2]` and write access `A[i-2]` there is a dependence if:

$$\exists \text{ integers } \quad i_w, i_v \qquad 2 \le i_w, i_v \le 5 \qquad i_w - 2 = i_v - 2$$

  - To rule out the case when the same instance depends on itself:
    - add constraint $i_w \ne i_v$

Memory Disambiguation

is

Undecidable at Compile Time

```
read(n)
For i =
    a[i] = a[n]
```

Domain of Data Dependence Analysis

- Only use loop bounds and array indexes that are affine functions of loop variables

```
for i = 1 to n
   for j = 2i to 100
      a[i+2j+3][4i+2j][i*i] = …
                         … = a[1][2i+1][j]
```

- Assume a data dependence between the read & write operation if there exists:
  - a read instance with indexes i_r, j_r and
  - a write instance with indexes i_w, j_w

$$\exists \text{ integers } \quad i_r, j_r, i_w, j_w \qquad 1 \le i_w, i_r \le n \qquad 2i_w \le j_w \le 100$$

$$2i_r \le j_r \le 100$$

$$i_w + 2j_w + 3 = 1 \qquad 4i_w + 2j_w = 2i_r + 1$$

- Equate each dimension of array access; ignore non-affine ones
  - No solution → No data dependence
  - Solution → there may be a dependence

Complexity of Data Dependence Analysis

For every pair of accesses not necessarily distinct (F_1 , f_1) and (F_2, f_2)

one must be a write operation

Let $B_1 i_1 + b_1 \geq 0$, $B_2 i_2 + b_2 \geq 0$ be the corresponding loop bound constraints,

$$\exists \text{ integers } i_1, i_2 \quad B_1 i_1 + b_1 \geq 0, \; B_2 i_2 + b_2 \geq 0$$

$$F_1 i_1 + f_1 = F_2 i_2 + f_2$$

If the accesses are not distinct, then add the constraint $i_1 \neq i_2$

- Equivalent to integer linear programming

$$\exists \text{ integer } \vec{i} \qquad \vec{A}_1 \vec{i} \leq \vec{b}_1 \qquad \vec{A}_2 \vec{i} = \vec{b}_2$$

- Integer linear programming is NP-complete
  - O(size of the coefficients) or $O(n^n)$

Data Dependence Analysis Algorithm

- Typically solving many tiny, repeated problems
  - Integer linear programming packages optimize for large problems
  - Use memoization to remember the results of simple tests
- Apply a series of relatively simple tests
  - GCD: 2*i, 2*i+1; GCD for simultaneous equations
  - Test if the ranges overlap
- Backed up by a more expensive algorithm
  - Use Fourier-Motzkin Elimination to test if there is a real solution
    - Keep eliminating variables to see if a solution remains
    - If there is no solution, then there is no integer solution

Fourier-Motzkin Elimination

- To eliminate a variable from a set of linear inequalities.
- To eliminate a variable x_1
  - Rewrite all expressions in terms of lower or upper bounds of x_1
  - Create a transitive constraint for each pair of lower and upper bounds.
- Example: Let L, U be lower bounds and upper bounds resp
  - To eliminate x_1:

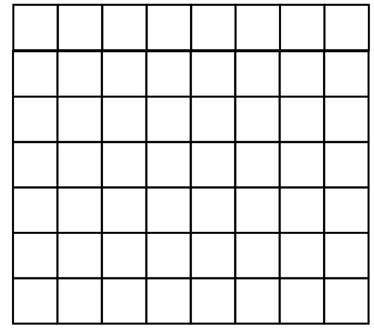
$$L_1(x_2, \dots, x_n) \le x_1 \le U_1(x_2, \dots, x_n)$$
$$L_2(x_2, \dots, x_n) \le x_1 \le U_2(x_2, \dots, x_n)$$

$$\Rightarrow$$

$$L_1(x_2, \dots, x_n) \le U_1(x_2, \dots, x_n)$$
$$L_1(x_2, \dots, x_n) \le U_2(x_2, \dots, x_n)$$
$$L_2(x_2, \dots, x_n) \le U_1(x_2, \dots, x_n)$$
$$L_2(x_2, \dots, x_n) \le U_2(x_2, \dots, x_n)$$

Example

```
FOR i = 1 to 5
    FOR j = i+1 to 5
      A[i,j] = f(A[i,i], A[i-1,j])
```



$$\begin{aligned} 1 &\le i \\ i &\le 5 \\ i+1 &\le j \\ j &\le 5 \end{aligned} \qquad \begin{aligned} 1 &\le i' \\ i' &\le 5 \\ i'+1 &\le j' \\ j' &\le 5 \end{aligned}$$

Data Dependence Analysis Algorithm

- Typically solving many tiny, repeated problems
  - Integer linear programming packages optimize for large problems
  - Use memoization to remember the results of simple tests
- Apply a series of relatively simple tests
  - GCD: 2*i, 2*i+1; GCD for simultaneous equations
  - Test if the ranges overlap
- Backed up by a more expensive algorithm
  - Use Fourier-Motzkin Elimination to test if there is a real solution
    - Keep eliminating variables to see if a solution remains
    - Add heuristics to encourage finding an integer solution.
  - Create 2 subproblems if a real, but not integer, solution is found.
    - For example, if $x = .5$ is a solution,
      create two problems,
      by adding $x \leq 0$ and $x \geq 1$ respectively to original constraint.

Relaxing Dependences

Privatization:

- Scalar

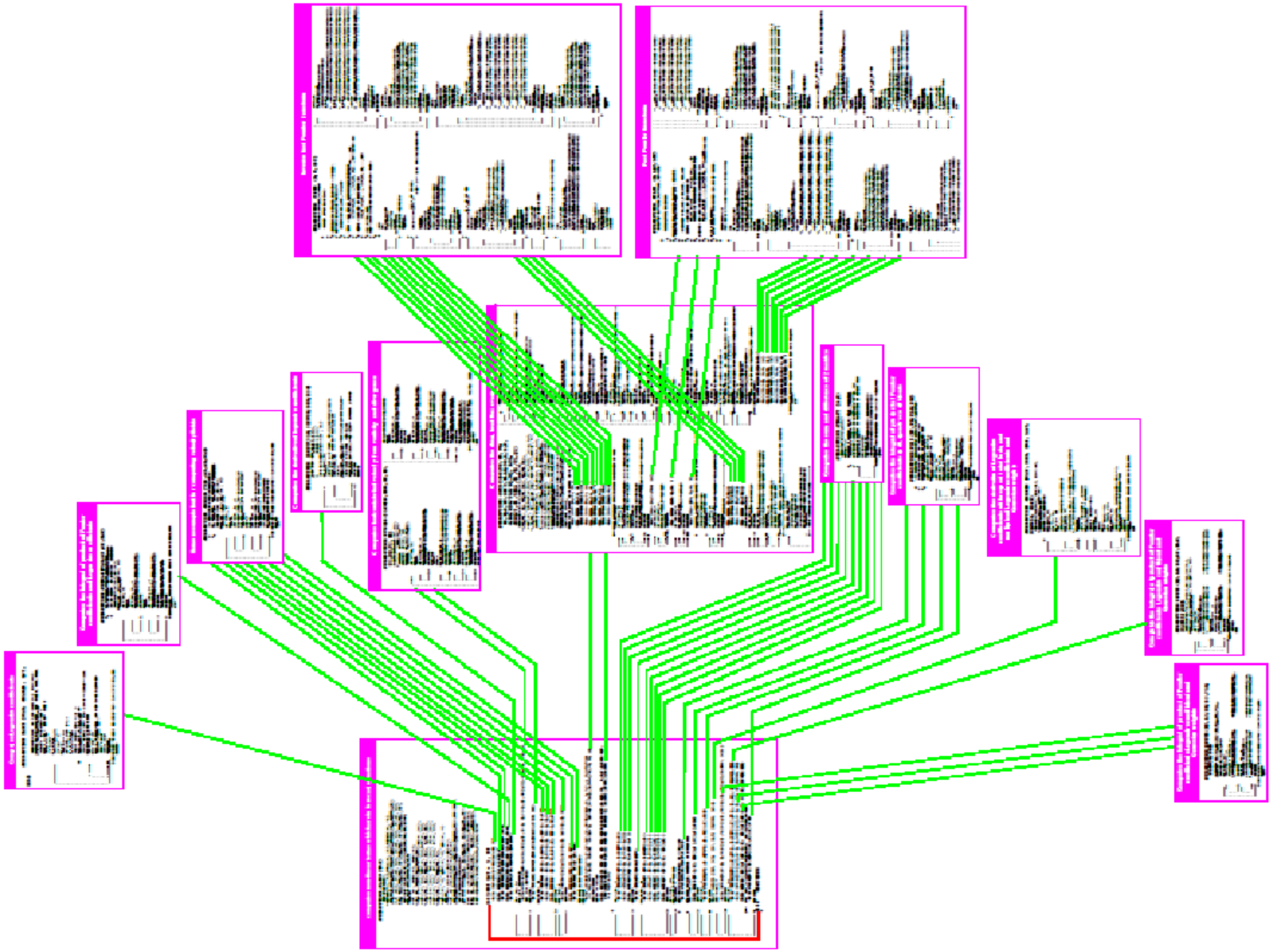
```
for i = 1 to n
   t = (A[i] + B[i]) / 2;
   C[i] = t * t;
```

- Array

```
for i = 1 to n
   for j = 1 to n
      t[j] = (A[i,j] + B[i,j]) / 2;
   for j = 1 to n
      C[i,j] = t[j] * t[j];
```

Reduction:

```
for i = 1 to n
   sum = sum + A[i];
```



Interprocedural Parallelization

- Why? Amdahl's Law
- Interprocedural symbolic analysis
  - Find interprocedural array indexes
    which are affine expressions of outer loop indices
- Interprocedural parallelization analysis
  - Data dependence based on summaries of array regions accessed
    - If the regions do not intersect, there is parallelism
  - Find privatizable scalar variables and arrays
  - Find scalar and array reductions

Conclusions

- Basic parallelization
  - Doall loop: loops with no loop-carried data dependences
  - Data dependence for affine loop indexes = integer linear programming

- Coarse-grain parallelism because of Amdahl's Law
  - Interprocedural analysis is useful for affine indices
  - Ask users for help on unresolved dependences