Lecture 10

Parallelization

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

Chapter 11.1-11.1.4

Why?

- Automatic parallelization is the holy grail
- Numerical applications, signal processing
  - A simpler but very useful domain
  - Has dense matrices
  - Lots of parallelism, ways to parallelize
  - But still hard to get good performance
- Understanding parallelization makes you a better programmer for parallel machines
- Beautiful abstraction: linear algebra, integer linear programming

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: $i_n, i_{n-1}, \dots, i_1$
  - Integer constants: $c_n, c_{n-1}, \dots, c_0$
  - Array index: $c_n i_n + c_{n-1} i_{n-1} + \dots + c_1 i_1 + c_0$
  - 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

    $\exists$integers i_w, i_r $2 \le i_w, i_r \le 5$ $i_r = i_w - 2$

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

    $\exists$integers i_w, i_v $2 \le i_w, i_v \le 5$ $i_w - 2 = i_v - 2$

  - 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:
  - Let a read instance be denoted with indexes i_r, j_r and
  - a write instance be denoted with indexes i_w, j_w

∃ Integers i_r, j_r, i_w, j_w, n

$$\begin{bmatrix} 1 & 0 & 0 \\ -1 & 0 & 1 \\ -2 & 1 & 0 \\ 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} i_w \\ j_w \\ n \end{bmatrix} + \begin{bmatrix} -1 \\ 0 \\ 0 \\ 100 \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \qquad \begin{bmatrix} 1 & 0 & 0 \\ -1 & 0 & 1 \\ -2 & 1 & 0 \\ 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} i_r \\ j_r \\ n \end{bmatrix} + \begin{bmatrix} -1 \\ 0 \\ 0 \\ 100 \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 \\ 4 & 2 \end{bmatrix} \begin{bmatrix} i_w \\ j_w \end{bmatrix} + \begin{bmatrix} 3 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 2 & 0 \end{bmatrix} \begin{bmatrix} i_r \\ j_r \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

Domain of Data Dependence Analysis

- 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,

$\exists$ 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

$\exists$ integer $\vec{i}$ $\quad A_1 \vec{i} \leq \vec{b}_1 \quad A_2 \vec{i} = \vec{b}_2$

- 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

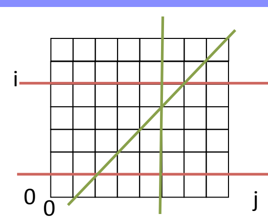
- To eliminate a variable from a set of linear inequalities.
- To eliminate a variable x_1
  - Rewrite all expressions in terms of lower or upper bounds of x_1
  - Create a transitive constraint for each pair of lower and upper bounds.
- Example: Let L, U be lower bounds and upper bounds resp
  - To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

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

$\Rightarrow$

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

Example




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

write	read
1 ≤ i	1 ≤ i'
i ≤ 5	i' ≤ 5
i + 1 ≤ j	i' + 1 ≤ j'
j ≤ 5	j' ≤ 5

1: Data dep between A[i,j], A[i',i']
- i = i'
- j = i'
- i'+1 ≤ i'

2: Data dep between A[i,j] and A[i'-1,j']
- i = i' – 1 => i+1 = i'
- j = j'

Substituting
- 1 ≤ i + 1, i + 1 ≤ 5
- i + 2 ≤ j j ≤ 5

Combining
- 1 ≤ i; i ≤ 4 i ≤ j -2; j ≤ 5

Eliminating i:
- 1 ≤ 4; 1 ≤ j -2; j ≤ 5
- 3 ≤ j; j ≤ 5

Eliminating j:
- 3 ≤ 5

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

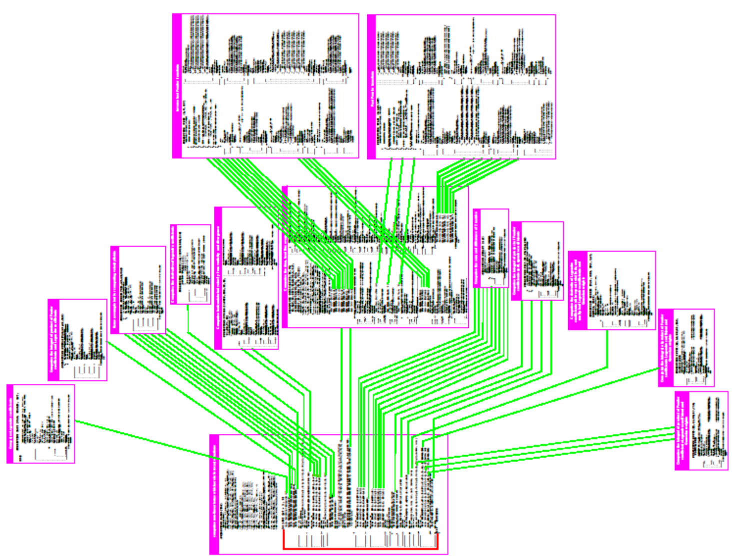
```
for i = 1 to n
   for j = 1 to n
      t[j] = (A[i,j] + B[i,j]) / 2;
   for j = 1 to n
      C[i,j] = t[j] * t[j];
```

Reduction:

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

Interprocedural Parallelization

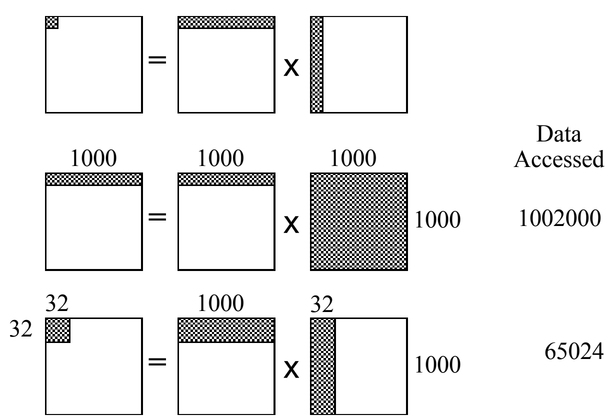
- Why? Amdahl's Law
- Interprocedural symbolic analysis
  - Find interprocedural array indexes which are affine expressions of outer loop indices
- Interprocedural parallelization analysis
  - Data dependence based on summaries of array regions accessed
    - If the regions do not intersect, there is no parallelism
  - Find privatizable scalar variables and arrays
  - Find scalar and array reductions



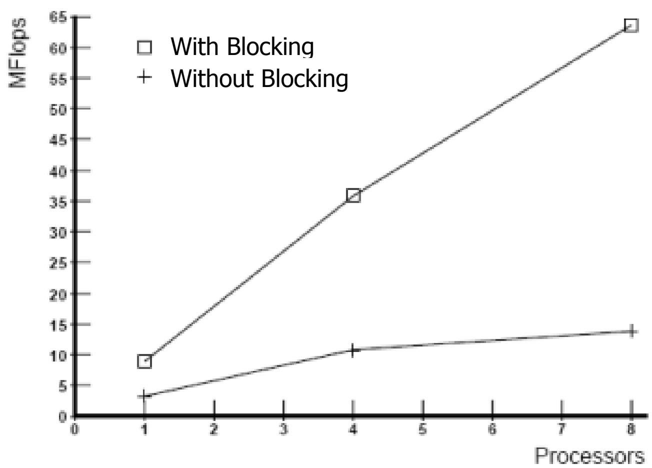
Conclusions

- 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

1. Blocking Example: Matrix Multiplication



Experimental Results



Code Transform

- Before

```
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++) {
     for (k = 0; k < n; k++) {
        Z[i,j] = Z[i,j] + X[i,k]*Y[k,j];
     }}}
```

- After

```
for (ii = 0; ii < n; ii = ii+B) {
  for (jj = 0; jj < n; jj = jj+B) {
     for (kk = 0; kk < n; kk = kk+B) {
       for (i = ii; i < min(n,kk+B); i++) {
         for (j = jj; j < min(n,kk+B); j++) {
            for (k = kk; k < min(n,kk+B); k++) {
              Z[i,j] = Z[i,j] + X[i,k] * Y[k,j];
           }}}}}}
```