Lecture 12
Parallelization

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

Chapter 11.1-11.1.4

Why?

• Automatic parallelization is the holy grail
• 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:** \((i, j, k \text{ are loop indexes})\)
  \[ A[i,j], A[i-1, j+1] \]

- **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

• 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 $i_r$ and $i_w$ read & write the same array element, respectively
      $\exists$ integers $i_w, i_r \ 2 \leq i_w, i_r \leq 5 \ i_r = i_w - 2$

• Between write access A[i-2] and write access A[i-2] there is a dependence if:
    $\exists$ integers $i_w, i_r \ 2 \leq i_w, i_r \leq 5 \ i_w - 2 = i_r - 2$

– To rule out the case when the same instance depends on itself:
  • add constraint $i_w \neq i_r$

Memory Disambiguation

is

Undecidable at Compile Time

read(n)
For i =
a[i] = a[n]
Only use loop bounds and array indexes that are affine functions of loop variables

\[
\text{for } i = 1 \text{ to } n \\
\text{for } j = 2i \text{ to } 100 \\
a[i+2j+3][4i+2j][i+1] = … \\
… = a[1][2i+1][j]
\]

Assume a data dependence between the read & write operation if:
– Let a read instance be denoted with indexes \(i_r, j_r\)
– A write instance be denoted with indexes \(i_w, j_w\)

\[
\exists \text{Integers } i_r, j_r, i_w, j_w \text{ such that } \\
\begin{bmatrix}
1 & 0 & 0 \\
-1 & 0 & 1 \\
-2 & 1 & 0 \\
0 & -1 & 0
\end{bmatrix}
\begin{bmatrix}
i_r \\
j_r \\
i_w \\
j_w
\end{bmatrix}
\begin{bmatrix}
-1 \\
0 \\
-2 \\
0
\end{bmatrix}
\begin{bmatrix}
0 \\
0 \\
0 \\
100
\end{bmatrix}
\begin{bmatrix}
i_r \\
j_r \\
i_w \\
j_w
\end{bmatrix}
\begin{bmatrix}
1 \\
2 \\
3 \\
1
\end{bmatrix}
\]

\[
\text{Equate each dimension of array access; ignore non-affine ones} \\
\begin{align*}
\text{No solution} & \Rightarrow \text{No data dependence} \\
\text{Solution} & \Rightarrow \text{there may be a dependence}
\end{align*}
\]
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

\[\exists \text{ integer } \tilde{i} \quad A_1 \tilde{i} \leq \tilde{b}_1 \quad A_2 \tilde{i} \leq \tilde{b}_2\]

• Integer linear programming is NP-complete
  – \(O(\text{size of the coefficients})\) or \(O(n^2)\)

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: \(2^\ast i, 2^\ast i+1\); 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 $x_i$.
  - Rewrite all expressions in terms of lower or upper bounds of $x_i$.
  - 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 $x_i$:

$$
L_i(x_2, \ldots, x_n) \leq x_1 \leq U_i(x_2, \ldots, x_n)
\quad L_2(x_2, \ldots, x_n) \leq x_1 \leq U_2(x_2, \ldots, x_n)
\Rightarrow L_1(x_2, \ldots, x_n) \leq U_1(x_2, \ldots, x_n)
\quad L_2(x_2, \ldots, x_n) \leq U_2(x_2, \ldots, x_n)
$$

Example

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

1: Data dep between $A[i,j], A[i,i]$
\[ i = i' \]
\[ j = j' \]
\[ i' + 1 = i' \]

2: Data dep between $A[i,j]$ and $A[i-1,j]$
\[ i = i' - 1 \Rightarrow i+1 = i' \]
\[ j = j' \]

Substituting
\[ 1 \leq i + 1 \]
\[ i + 1 \leq j \]
\[ j \leq 5 \]

Combining
\[ 1 \leq i \leq 4 \]
\[ i \leq j - 2 \]
\[ j \leq 5 \]

Eliminating $i$:
\[ 1 \leq 4 \leq 1 \leq j - 2 \leq 5 \]
\[ 3 \leq j \]

Eliminating $j$:
\[ 3 \leq 5 \]
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 ≤ 0 and x ≥ 1 respectively to original constraint.

Relaxing Dependences

Privatization:

• Scalar

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

• Array

```plaintext
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:

```plaintext
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

\[
\begin{array}{ccc}
\begin{array}{c}
1000 \\
\end{array} & = & \begin{array}{c}
1000 \\
\end{array} \\
\begin{array}{c}
32 \\
\end{array} & = & \begin{array}{c}
1000 \\
\end{array}
\end{array}
\times \begin{array}{c}
1000 \\
\end{array}
\]

Data Accessed

\[
\begin{array}{ccc}
\begin{array}{c}
1000 \\
\end{array} & = & \begin{array}{c}
1000 \\
\end{array} \\
\begin{array}{c}
32 \\
\end{array} & = & \begin{array}{c}
1000 \\
\end{array}
\end{array}
\times \begin{array}{c}
1000 \\
\end{array}
\]

1002000

65024
Experimental Results

![Graph showing MFlops with and without blocking](image)

<table>
<thead>
<tr>
<th>Processors</th>
<th>With Blocking</th>
<th>Without Blocking</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>50</td>
<td>45</td>
</tr>
<tr>
<td>2</td>
<td>55</td>
<td>50</td>
</tr>
<tr>
<td>3</td>
<td>60</td>
<td>55</td>
</tr>
<tr>
<td>4</td>
<td>65</td>
<td>60</td>
</tr>
</tbody>
</table>

**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];
            }
          }
        }
      }
    }
  }
  ```