

Parallelization

Rupesh Nasre.

CS6843 Program Analysis
IIT Madras
Jan 2014

Control Dependence

- if (x == 4) y = 10; else y = 1;

5

Speedup

- Speedup = T_s / T_p
- **Amdahl's Law**: Speedup is limited by the sequential part of the task.
- If 20% of the task is sequential, program's speedup is limited to 5 (irrespective of the number of cores or amount of effort).

3

Data Dependence

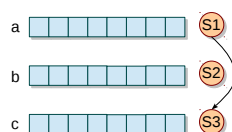
- $\pi = 3.142$; $r = 5.0$; $\text{area} = \pi * r * r$;
- Types
 - **True / Flow**: $S1 \delta S2 (x = \dots; \dots = x;)$
 - **Anti**: $S1 \delta^{-1} S2 (\dots = x; x = \dots)$
 - **Output**: $S1 \delta^0 S2 (x = \dots; x = \dots)$

6

Task Parallel vs. Data Parallel

- Parallelism extracted from multiple instructions on the data items.
- Parallelism extracted from the same task on different data items.

S1: a[i] = 2
S2: b[i] = 4
S3: c[i] = a[i]



S1 is the source and S3 is the sink of the dependence.

4

Program Order vs. Dependence

- **Sequential** order imposed by the program is too restrictive.
- Only the **partial order** of all dependences need to be maintained by the compiler to guarantee program correctness.
- So, reorder flow; maintain dependence.

7

Advantages of Reordering

- Improved **locality**
 - Spatial: matrix operations
 - Temporal: xinit(); yinit(); xcompute(); ycompute();
- Improved **load balance**
 - small1(); big1(); small2(); big2();
- Improved **parallelism**
 - xuse(); xdef(); yuse(); ydef();

8

Reordering Transformations

- A **reordering transformation** is any program transformation that merely changes the execution order of the code, without adding or deleting any executions of any statements.
- A reordering transformation preserves a dependence if it preserves the relative execution order of the source and the sink of that dependence.
- **Theorem:** Any reordering transformation that preserves every dependence in a program leads to an equivalent computation.

11

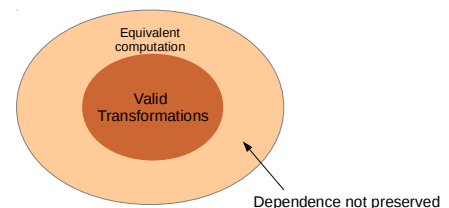
Let's Focus on Loops

- **Iteration vector:** Sequence of outer loops.
 - $\vec{iv} = (i_{outermost}, \dots, i_{middle}, \dots, i_{innermost})$
 - For instance (i, j, k).
- **Iteration space:** Set of all possible iteration vectors for a statement.
- **Statement instance:** $S(\vec{i})$
- $S(\vec{i}) \delta S(\vec{j})$ iff
 - (a) $i < j$ or ($i == j$ and $S1 \Rightarrow \Rightarrow S2$ path in loop-body)
 - (b) both access same memory location
 - (c) at least one of the accesses is a write

9

Valid Transformations

- A transformation is valid for the program to which it applies if it preserves all the dependences in the program.



12

Safe Transformations

- **Loop Dependence Theorem**
 - There exists a dependence from statement S1 to statement S2 in a common nest of loops iff there exist two iteration vectors \vec{i} and \vec{j} for the nest, such that $S1(\vec{i}) \delta S2(\vec{j})$.
- Two computations are **equivalent** if on the same inputs they produce the same output.
- A transformation is **safe** if it leads to an equivalent program.

10

Loop Parallelization

- **Theorem:** It is valid to convert a sequential loop to a parallel loop if the loop carries no dependence.

```
for (k = 0; k < n; ++k) {
  S1: a[k] = b[k];
  S2: b[k] = a[k] + 1;
}
```



```
for (k = 0; k < n; ++k) {
  S1: a[k] = a[k + 1];
}
```



14

General Strategy

```
for (ii = 0; ii < n; ++ii) {
  for (jj = 0; jj < m; ++jj) {
    a[f(ii, jj)][g(ii, jj)] = ...
    ... = ... a[h(ii, jj)][k(ii, jj)]...
  }
}
```

Conditions for flow dependence from iteration (ii_w, jj_w) to (ii_r, jj_r) :

$$\begin{aligned} 0 &\leq ii_w < n \\ 0 &\leq jj_w < m \\ 0 &\leq ii_r < n \\ 0 &\leq jj_r < m \\ (ii_w, jj_w) &\leq (ii_r, jj_r) \\ f(ii_w, jj_w) &= h(ii_r, jj_r) \\ g(ii_w, jj_w) &= k(ii_r, jj_r) \end{aligned}$$

If f, g, h, k are affine functions of loop variables, then dependence testing can be formulated as an ILP.

16

ILP Formulation

```
for (ii = 0; ii < 10; ++ii) {
  a[2 * ii] = ... a[ii + 1] ...
}
```

Is there an anti-dependence between different iterations?

Dependence equations
 $0 \leq ii_r < ii_w < 10$
 $2 * ii_w = ii_r + 1$

which can be written as

$$\begin{aligned} 0 &\leq ii_r \\ ii_r &\leq ii_w - 1 \\ ii_w &\leq 9 \\ 2 * ii_w &\leq ii_r + 1 \\ ii_r + 1 &\leq 2 * ii_w \end{aligned} \quad \left\{ \begin{array}{cc} -1 & 0 \\ 1 & -1 \\ 0 & 1 \\ -1 & 2 \\ 1 & -2 \end{array} \right\} \begin{bmatrix} ii_r \\ ii_w \end{bmatrix} \leq \begin{bmatrix} 0 \\ -1 \\ 9 \\ 1 \\ -1 \end{bmatrix}$$

The system is not satisfiable, so anti-dependence does not exist.

19

ILP Formulation

```
for (ii = 0; ii < 10; ++ii) {
  a[2 * ii] = ... a[2 * ii + 1] ...
}
```

Is there a flow dependence between different iterations?

Dependence equations
 $0 \leq ii_w < ii_r < 10$
 $2 * ii_w = 2 * ii_r + 1$

which can be written as

$$\begin{aligned} 0 &\leq ii_w \\ ii_w &\leq ii_r - 1 \\ ii_r &\leq 9 \\ 2 * ii_w &\leq 2 * ii_r + 1 \\ 2 * ii_r + 1 &\leq 2 * ii_w \end{aligned} \quad \left\{ \begin{array}{cc} -1 & 0 \\ 1 & -1 \\ 0 & 1 \\ 2 & -2 \\ -2 & 2 \end{array} \right\} \begin{bmatrix} ii_w \\ ii_r \end{bmatrix} \leq \begin{bmatrix} 0 \\ -1 \\ 9 \\ 1 \\ -1 \end{bmatrix}$$

Dependence exists if the system has a solution.

17

ILP Formulation

```
for (ii = 0; ii < 10; ++ii) {
  a[2 * ii] = ... a[ii + 1] ...
}
```

Is there a true dependence between different iterations?

Dependence equations
 $0 \leq ii_w < ii_r < 10$
 $2 * ii_w = ii_r + 1$

which can be written as

$$\begin{aligned} 0 &\leq ii_w \\ ii_w &\leq ii_r - 1 \\ ii_r &\leq 9 \\ 2 * ii_w &\leq ii_r + 1 \\ ii_r + 1 &\leq 2 * ii_w \end{aligned} \quad \left\{ \begin{array}{cc} 0 & -1 \\ -1 & 1 \\ 1 & 0 \\ -1 & 2 \\ 1 & -2 \end{array} \right\} \begin{bmatrix} ii_r \\ ii_w \end{bmatrix} \leq \begin{bmatrix} 0 \\ -1 \\ 9 \\ 1 \\ -1 \end{bmatrix}$$

The system is satisfiable, so true dependence exists.

| ii_r | ii_w |
|--------|--------|
| 0 | -- |
| 1 | -- |
| 2 | -- |
| 3 | 2 |
| 4 | -- |
| 5 | 3 |
| 6 | -- |
| 7 | 4 |
| 8 | -- |
| 9 | 5 |

20

ILP Formulation

```
for (ii = 0; ii < 10; ++ii) {
  a[2 * ii] = ... a[2 * ii + 1] ...
}
```

Is there an anti-dependence between different iterations?

Dependence equations
 $0 \leq ii_r < ii_w < 10$
 $2 * ii_w = 2 * ii_r + 1$

which can be written as

$$\begin{aligned} 0 &\leq ii_r \\ ii_r &\leq ii_w - 1 \\ ii_w &\leq 9 \\ 2 * ii_w &\leq 2 * ii_r + 1 \\ 2 * ii_r + 1 &\leq 2 * ii_w \end{aligned} \quad \left\{ \begin{array}{cc} -1 & 0 \\ 1 & -1 \\ 0 & 1 \\ -2 & 2 \\ 2 & -2 \end{array} \right\} \begin{bmatrix} ii_r \\ ii_w \end{bmatrix} \leq \begin{bmatrix} 0 \\ -1 \\ 9 \\ 1 \\ -1 \end{bmatrix}$$

The system is not satisfiable, so anti-dependence does not exist.

18

Managing Races

- Data-race between iterations p and q for element $a[f(i)]$.
- Critical section
 - Locks
 - Atomics
 - Barriers

23

Inserting Locks

- Data-race between iterations p and q for element $a[f(i)]$.

```
if (i == p || i == q) {
    lock(f(i));
    ... perform operation ...
    unlock(f(i));
}
```

This operation could be same or different for the involved threads.

- e.g., Producer-consumer

```
produce() {
    while (...) {
        items.add(...);
    }
}

consume() {
    e = items.remove();
}
```

24

Inserting Atomics

- If the operation is simple
 - Primitive type
 - Single element
 - Relative update / read-write
- Example
 - Producer-consumer with single element update
- Types
 - increment, decrement
 - add, sub
 - min, max
 - exch, CAS

27

Inserting Locks

- For multiple data items $a[f(i)]$ and $a[g(i)]$
 - Single lock
 - Multiple locks
- Multiple locks may lead to deadlock
 - may allow deadlock if it improves parallelism
- Deadlock avoidance may lead to livelock
 - may allow livelock if rare

25

Inserting Atomics

- **Classwork:** convert the following example from locks to atomics

```
if (i == p || i == q) {
    lock(f(i));
    sum += a[i];
    unlock(f(i));
}
```

- **Classwork:** write parallel slist insertion and deletion routines using atomics
- **Homework:** write parallel dlist insertion routine using atomics

28

Inserting Locks

- Sometimes, a lock may be for a simple operation

```
if (i == p || i == q) {
    lock(f(i));
    sum += a[i];
    unlock(f(i));
}
```

- A simple critical section may be convertible to atomics.

26

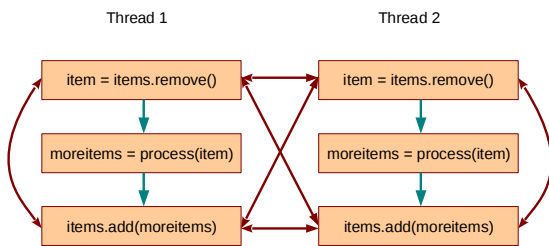
Inserting Locks

```
if (i == 1 || i == 2 || i == 4 || ...) {
    lock(f(i));
    item = items.remove();
    moreitems = process(item);
    items.add(moreitems);
    unlock(f(i));
}
```

- If there are many threads involved in the $\text{if}(\dots)$ condition and the operation is multi-step, overapproximate the dependences.

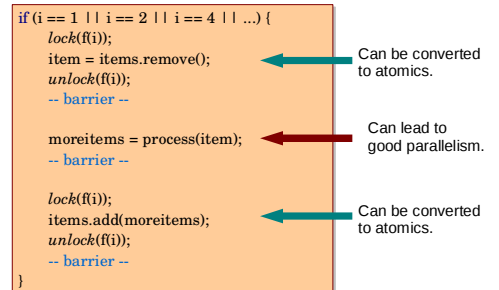
29

Dependences



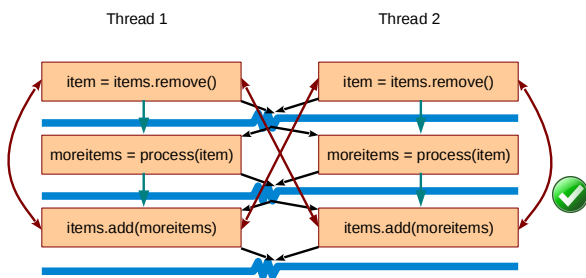
30

Inserting Barriers



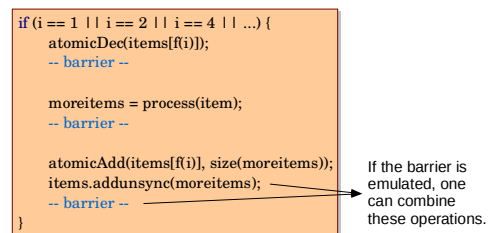
33

Barriers



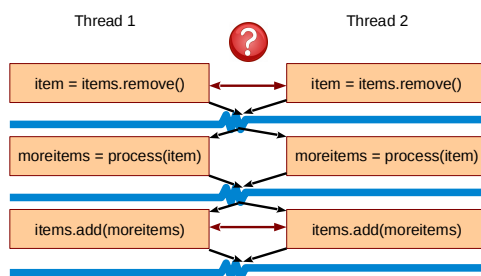
31

Inserting Barriers



34

Barriers



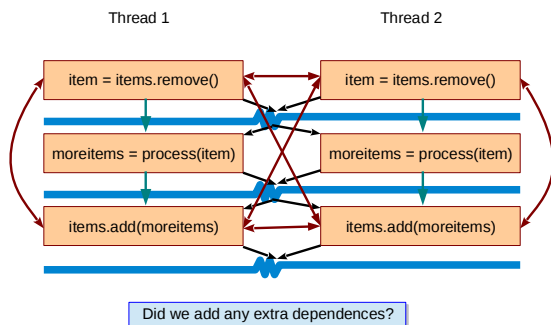
32

Barriers and Dependences

- A barrier may be considered in effect similar to loop distribution.
- If dependences are sparse, use atomics/locks; otherwise barriers work well.
- A barrier may add more dependences than required.
- But it must preserve all the existing dependences.

35

Barriers and Dependences



36

Barriers and Dependences

- **Homework:** Using the mechanisms we studied, construct an example wherein inserting a barrier may lead to incorrect parallel execution.

37