Parallelization

Rupesh Nasre.

CS6843 Program Analysis
IIT Madras
July 2025

Learning Outcomes

- Recall control and data dependences
- Recall RAW, WAR and WAW dependences
- Given a loop with array accesses, formulate an ILP to identify dependences
- Write simple codes with atomics and barriers
- Understand the limitations of static parallelization

Speedup

- Speedup = Ts / Tp
- Amdahl's Law: Speedup is limited by the sequential part of the task.

Speedup = $(1 – p + p / S_p)^{-1}$

p is the proportion of the parallel time.
S_p is the speedup of the parallel part.

When infinite resources are available, $p / S_p \rightarrow 0$

Assumes fixed problem size

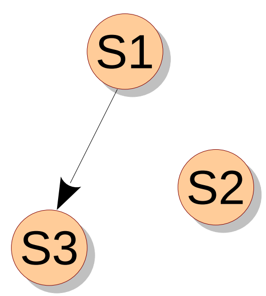
Speedup <= $(1 - p)^{-1}$

- If 20% of the task is sequential, program's speedup is limited to 5 (irrespective of the number of cores or amount of effort).

Instruction Parallel vs. Data Parallel

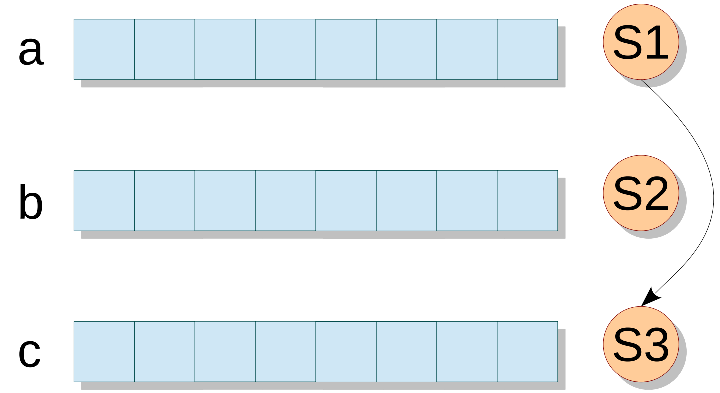
- Parallelism extracted from multiple instructions on the data items.

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



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

- Parallelism extracted from the same operation on different data items.



Task parallelism is a generalization which can have an arbitrary control and data sequence.

Control Dependence

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

Data Dependence

- $pi = 3.142; r = 5.0; area = pi * r * r;$
- Types
  - True / Flow / RAW: S1 δ S2 (x = ...; ... = x;)
  - Anti / WAR: S1 δ^{-1} S2 (... = x; x = ...;)
  - Output / WAW: S1 δ^{o} S2 (x = ...; x = ...;)
  - Input / RAR: S1 δ^{i} S2 (... = x; ... = x;)
  - RAW, WAR, WAW lead to *dataraces*. A datarace may affect correctness.

Program Order vs. Dependence

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

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();

Let's Focus on Loops

- Iteration vector: Sequence of outer loops.
  - $\vec{iv}$ = (ioutermost, ..., imiddle, ..., iinnermost)
  - For instance (i, j, k).
- Iteration space: Set of all possible iteration vectors for a statement.
- Statement instance: $S(\vec{iv})$
- $S(\vec{iv})\ \delta\ S(\vec{jv})$ iff // true dependence

  (a) $\vec{iv} < \vec{jv}$ or ($\vec{iv} == \vec{jv}$ and $S(\vec{iv})$ ⇨⇨⇨ $S(\vec{jv})$ dependence-path in loop-body)

  (b) both access the same memory location

  (c) at least one of the accesses is a write

Safe Transformations

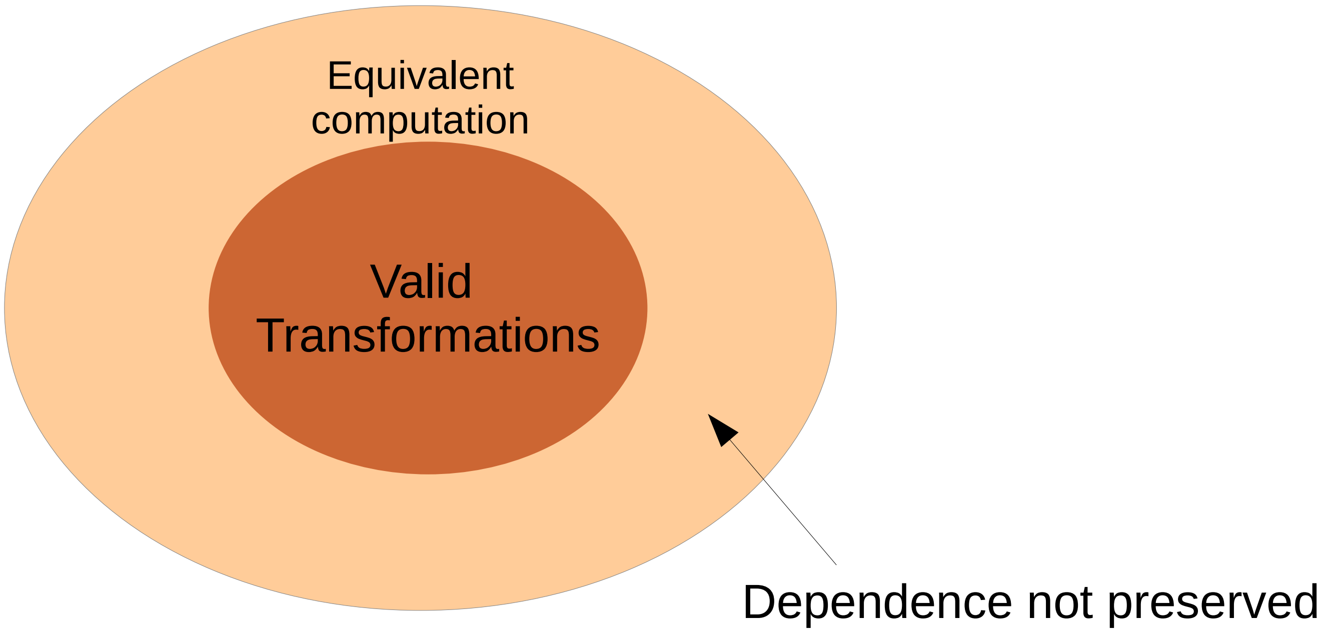
- **Loop Dependence Analysis**
  - There exists a dependence from statement S1 to statement S2 in a common nest of loops iff there exist two iteration vectors $\vec{i}$ and $\vec{j}$ for the nest, such that S1($\vec{i}$) δ S2($\vec{j}$).
- Two computations are equivalent if on the same inputs they produce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

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.

Valid Transformations

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



Classwork: Write a simple transformation that maintains computation equivalence but does not preserve dependence.

Loop Parallelization

- Usually, iterations of a loop are distributed across threads.
- **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;
}
```

k 0 1 2 ... n-1

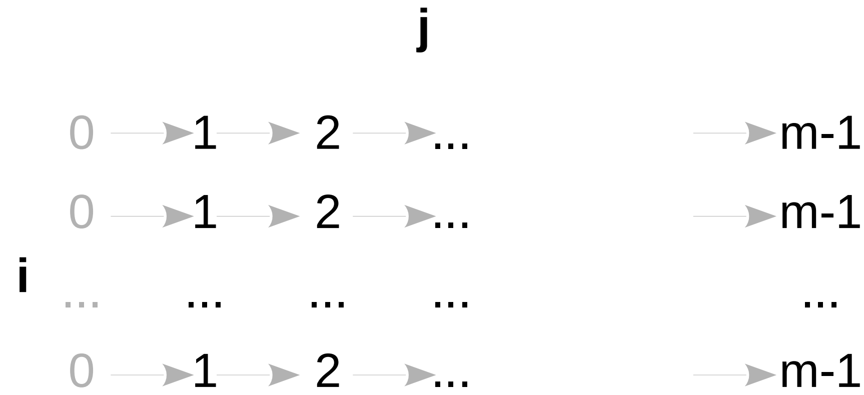
```
for (k = 0; k < n; ++k) {
  S1: a[k] = a[k + 1];
}
```

k 0 ← 1 ← 2 ← ... ← n-1

Loop Parallelization

```
for (i = 0; i < n; ++i)
    for (j = 1; j < m; ++j)
        S1: a[i][j] = a[i][j-1] * pi;
```

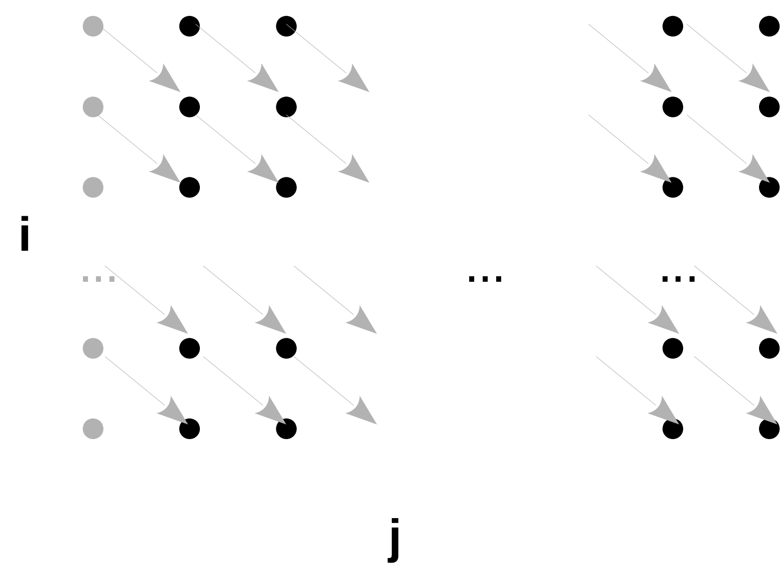
Note: No loop-carried dependence across rows.
So we can parallelize the outer loop.



Loop-Carried Dependence Graph

```
for (i = 1; i < n; ++i)
    for (j = 1; j < m; ++j)
        S1: a[i][j] = a[i-1][j-1] * pi;
```

Note: We can parallelize across the diagonals.



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)]...
    }
}
```

Can the nested loop be parallelized?

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

$0 <= ii_w < n$

$0 <= jj_w < m$

$0 <= ii_r < n$

$0 <= jj_r < m$

$(ii_w, jj_w) <= (ii_r, jj_r)$

$\mathbf{f}(ii_w, jj_w) = \mathbf{h}(ii_r, jj_r)$

$\mathbf{g}(ii_w, jj_w) = \mathbf{k}(ii_r, jj_r)$

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

Identifying Dependence

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

Is there a flow dependence between different iterations?

Flow dependence is read-after-write (to the same memory location).
$w \rightarrow\rightarrow\rightarrow\rightarrow r$

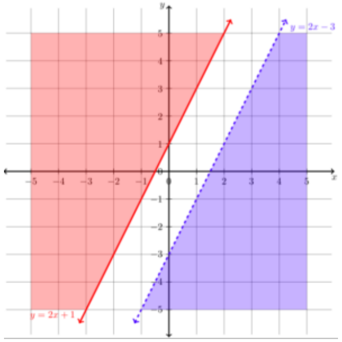
Dependence equations

$0 <= ii_w < ii_r < 10$

$2 * ii_w = 2 * ii_r + 1$

which can be written as

$$\left.\begin{array}{l} 0 <= ii_w \\ ii_w <= ii_r - 1 \\ ii_r <= 9 \\ 2 * ii_w <= 2 * ii_r + 1 \\ 2 * ii_r + 1 <= 2 * ii_w \end{array}\right\} \quad \begin{pmatrix} -1 & 0 \\ 1 & -1 \\ 0 & 1 \\ 2 & -2 \\ -2 & 2 \end{pmatrix} \begin{pmatrix} ii_w \\ ii_r \end{pmatrix} <= \begin{pmatrix} 0 \\ -1 \\ 9 \\ 1 \\ -1 \end{pmatrix}$$



Dependence exists if the system has a solution.

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

$0 <= ii_r < ii_w < 10$

$2 * ii_w = 2 * ii_r + 1$

which can be written as

$$\left.\begin{array}{l} 0 <= ii_r \\ ii_r <= ii_w - 1 \\ ii_w <= 9 \\ 2 * ii_w <= 2 * ii_r + 1 \\ 2 * ii_r + 1 <= 2 * ii_w \end{array}\right\} \quad \begin{pmatrix} -1 & 0 \\ 1 & -1 \\ 0 & 1 \\ -2 & 2 \\ 2 & -2 \end{pmatrix} \begin{pmatrix} ii_r \\ ii_w \end{pmatrix} <= \begin{pmatrix} 0 \\ -1 \\ 9 \\ 1 \\ -1 \end{pmatrix}$$

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

ILP Formulation

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

Is there an anti-dependence between different iterations?

Dependence equations

$0 <= ii_r < ii_w < 10$

$2 * ii_w = ii_r + 1$

which can be written as

$$\left.\begin{array}{l} 0 <= ii_r \\ ii_r <= ii_w - 1 \\ ii_w <= 9 \\ 2 * ii_w <= ii_r + 1 \\ ii_r + 1 <= 2 * ii_w \end{array}\right\} \begin{pmatrix} -1 & 0 \\ 1 & -1 \\ 0 & 1 \\ -1 & 2 \\ 1 & -2 \end{pmatrix} \begin{pmatrix} ii_r \\ ii_w \end{pmatrix} <= \begin{pmatrix} 0 \\ -1 \\ 9 \\ 1 \\ -1 \end{pmatrix}$$

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

ILP Formulation

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

Is there a true dependence between different iterations?

Dependence equations

$0 <= ii_w < ii_r < 10$

$2 * ii_w = ii_r + 1$

which can be written as

$$\left.\begin{array}{l} 0 <= ii_w \\ ii_w <= ii_r - 1 \\ ii_r <= 9 \\ 2 * ii_w <= ii_r + 1 \\ ii_r + 1 <= 2 * ii_w \end{array}\right\} \begin{pmatrix} 0 & -1 \\ -1 & 1 \\ 1 & 0 \\ -1 & 2 \\ 1 & -2 \end{pmatrix} \begin{pmatrix} ii_r \\ ii_w \end{pmatrix} <= \begin{pmatrix} 0 \\ -1 \\ 9 \\ 1 \\ -1 \end{pmatrix}$$

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

The system is satisfiable, so true dependence exists.

ILP Formulation

```
for (ii = 0; ii < 10; ++ii) {
    a[2 * ii] = ... a[ii + 1] ...
    a[3 + ii] = ... a[5 * ii] ...
}
```

Is there a true dependence between different iterations?

We will have to model equations across **all** inter-iteration pairs of reads/writes.

- 2* ii and ii + 1
- 3 + ii and 5 * ii
- 2 * ii and 5 * ii
- 3 + ii and ii + 1

How about 2 * ii and 3 + ii?
How about ii + 1 and 5 * ii?

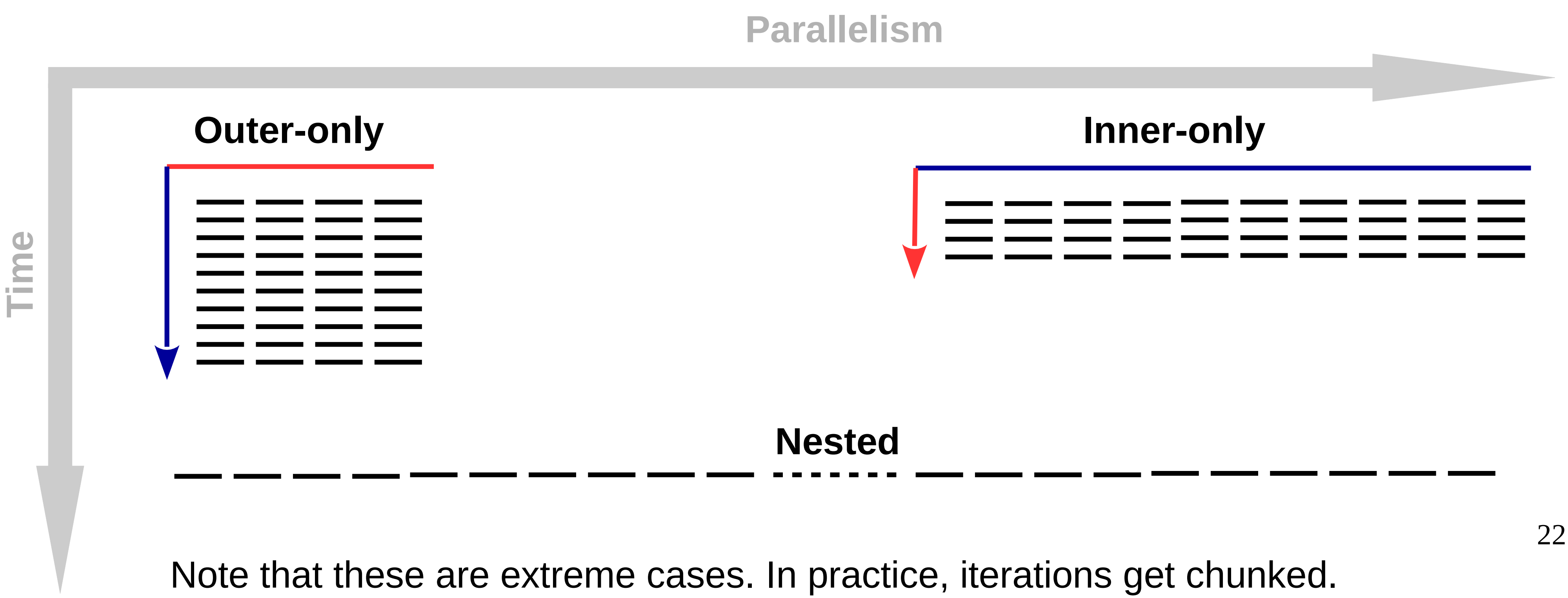
If **any** of the systems is satisfiable, then true dependence exists.

Nested Loops

```
for (ii = 0; ii < 4; ++ii)
    for (jj = 0; jj < 10; ++jj)
        a[2 * ii + 3][jj + 4] = … a[ii – 1][2 * jj - 5] …
```

Is there an anti-dependence between different iterations?

Three ways to parallelize:



Note that these are extreme cases. In practice, iterations get chunked.

Nested Loops

```
for (ii = 0; ii < 4; ++ii)
  for (jj = 0; jj < 10; ++jj)
    a[2 * ii + 3][jj + 4] = ... a[ii – 1][2 * jj - 5] ...
```

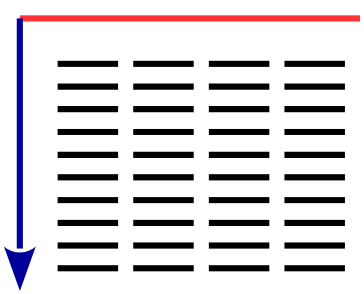
Is there an anti-dependence between different iterations?

Three ways to parallelize:

Parallelism

Time

Outer-only



Index constraints:	$0 <= ii_r <= 3$, $0 <= ii_w <= 3$,
	$0 <= jj_r <= 9$, $0 <= jj_w <= 9$
Memory location constraint:	$2 * ii_w + 3 == ii_r – 1$,
	$jj_w + 4 == 2 * jj_r - 5$
Anti-dependence constraint:	$ii_r < ii_w$ *[no constraint on jj]*

a[2 * ii + 3][jj + 4] and a[ii – 1][2 * jj – 5] would <u>not</u> conflict for the same value of ii.

Nested Loops

```
for (ii = 0; ii < 4; ++ii)
    for (jj = 0; jj < 10; ++jj)
        a[2 * ii + 3][jj + 4] = ... a[ii – 1][2 * jj - 5] ...
```

Is there an anti-dependence between different iterations?

Three ways to parallelize:

Parallelism

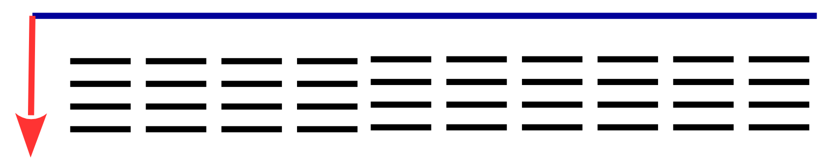
Time

Index constraints: $0 <= ii <= 3$, $0 <= jj_r <= 9$, $0 <= jj_w <= 9$

Memory location constraint: $2 * ii + 3 == ii – 1$, $jj_w + 4 == 2 * jj_r - 5$

Anti-dependence constraint: $jj_r < jj_w$ *[no constraint on ii]*

Inner-only



a[2 * ii + 3][jj + 4] and a[ii – 1][2 * jj – 5] would <u>not</u> conflict for different values of ii.

Nested Loops

```
for (ii = 0; ii < 4; ++ii)
   for (jj = 0; jj < 10; ++jj)
      a[2 * ii + 3][jj + 4] = … a[ii – 1][2 * jj - 5] …
```

Is there an anti-dependence between different iterations?

Three ways to parallelize:

Parallelism

Time

Index constraints: $0 <= ii_r <= 3$, $0 <= ii_w <= 3$,
$0 <= jj_r <= 9$, $0 <= jj_w <= 9$

Memory location constraint: $2 * ii_w + 3 == ii_r – 1$,
$jj_w + 4 == 2 * jj_r - 5$

Anti-dependence constraint: $ii_r < ii_w$ OR
$ii_r == ii_w$, $jj_r < jj_w$

Nested

GCD Test

- Solving linear set of equations may be time-consuming.
- A quicker way is GCD Test.
- **Theorem:** A linear equation: $\Sigma a_i x_i = c$ has an integral solution iff $GCD(a_1, a_2, ..., a_n)$ divides c.

```
for (ii = 0; ii < N; ++ii) {
    arr[4 * ii] = brr[ii];
    crr[ii] = arr[8 * ii + 3];
}
```

```
for (ii = 0; ii < N; ++ii) {
    arr[4 * ii] = brr[ii];
    crr[ii] = arr[8 * ii + 4];
}
```

Here, $4ii_1 = 8ii_2 + 3$

That is, $4ii_1 - 8ii_2 = 3$

Here, GCD(4, -8) == 4
which does not divide 3.
Hence, no true dependence.

GCD(4, -8) == 4 which divides 4.
Hence, true dependence MAY exist.

GCD Test: Classwork

- Solving linear set of equations may be time-consuming.
- A quicker way is GCD Test.
- **Theorem:** A linear equation: $\Sigma a_i x_i = c$ has an integral solution iff $GCD(a_1, a_2, ..., a_n)$ divides c.

```
for (ii = 1; ii < 6; ++ii) {
    b[ii] = a[3 * ii – 5] + 2.0;
    a[2*ii + 1] = 1.0 / ii;
}
```

Here, $3ii_2 - 5 = 2ii_1 + 1$
That is, $3ii_2 – 2ii_1 = 6$
Here, GCD(3, -2) == 1
which does divide 6.
Hence, true dependence may exist.
Due to additional constraint ii < 6,
true dependence does not exist.

Managing Races

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

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
  - Single producer, multiple consumers

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

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

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
- Use of instrumentation *(cf. dynamic analysis)*

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.

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

Inserting Atomics

- **Classwork:** convert the following example from locks to atomics (assume *atomicAdd*)

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

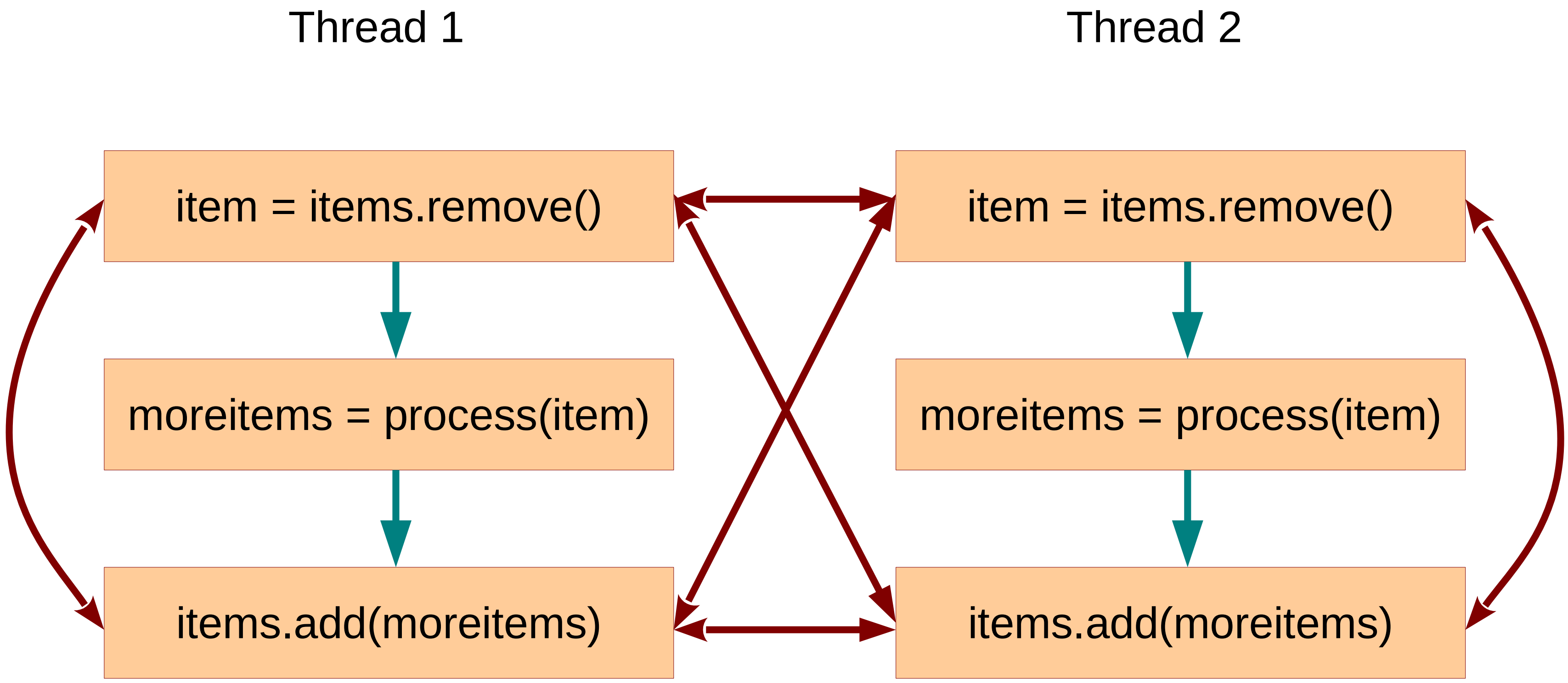
- **Classwork:** write parallel slist insertion and deletion routines using atomics
- **Homework:** write parallel dlist insertion routine using atomics

Inserting Locks

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

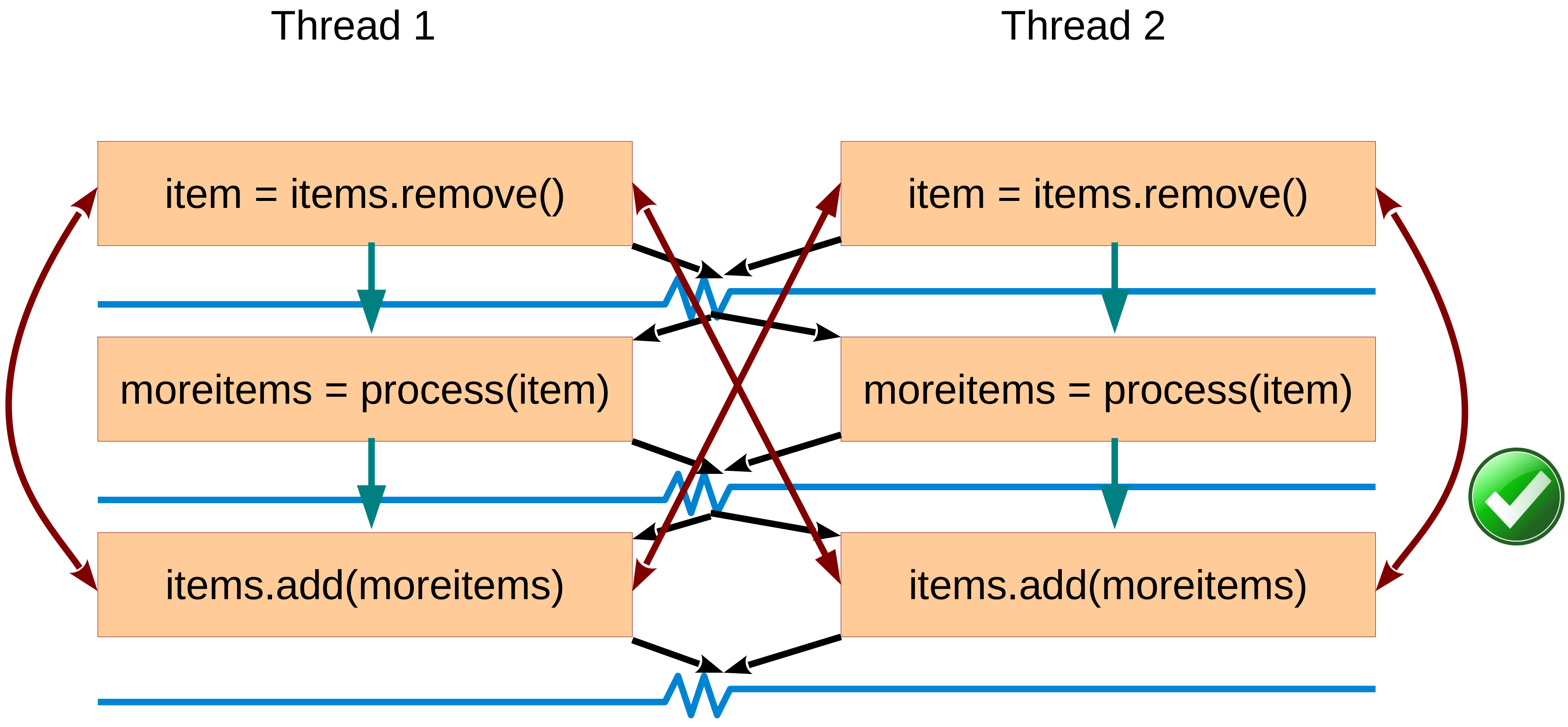
- If there are many threads involved in the if(...) condition and the operation is multi-step, overapproximate the dependences.

Races

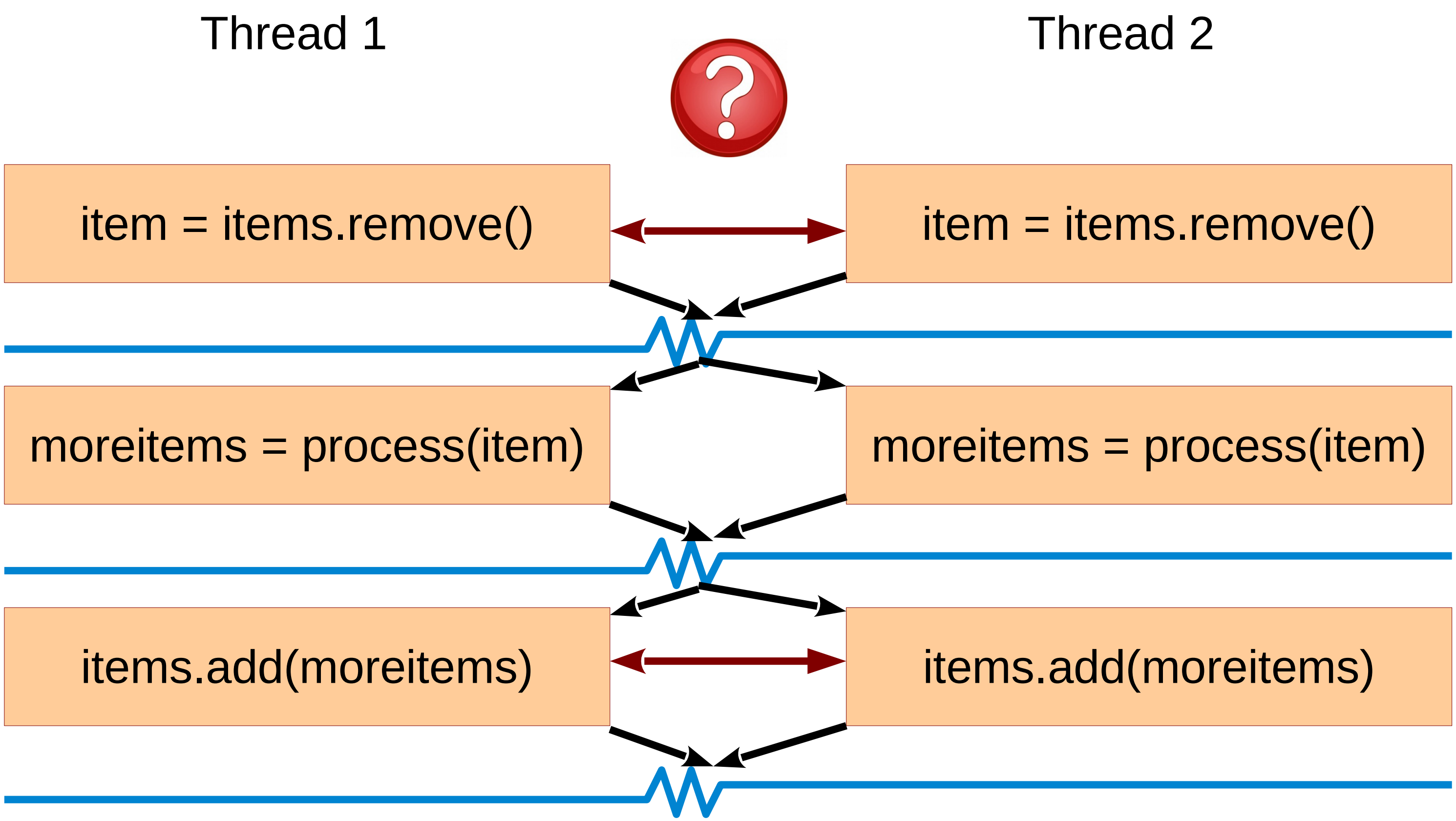


Thread 1
Thread 2
item = items.remove()
moreitems = process(item)
items.add(moreitems)
item = items.remove()
moreitems = process(item)
items.add(moreitems)

Barriers


Thread 1
Thread 2
item = items.remove()
item = items.remove()
moreitems = process(item)
moreitems = process(item)
items.add(moreitems)
items.add(moreitems)

Barriers


Thread 1
Thread 2
item = items.remove()
item = items.remove()
moreitems = process(item)
moreitems = process(item)
items.add(moreitems)
items.add(moreitems)

Inserting Barriers

```
if (i == 1 || i == 2 || i == 4 || ...) {
    lock(f(i));
    item = items.remove();
    unlock(f(i));
    -- barrier --

    moreitems = process(item);
    -- barrier --

    lock(f(i));
    items.add(moreitems);
    unlock(f(i));
    -- barrier --
}
```

Can be converted to atomics.

Can lead to good parallelism.

Can be converted to atomics.

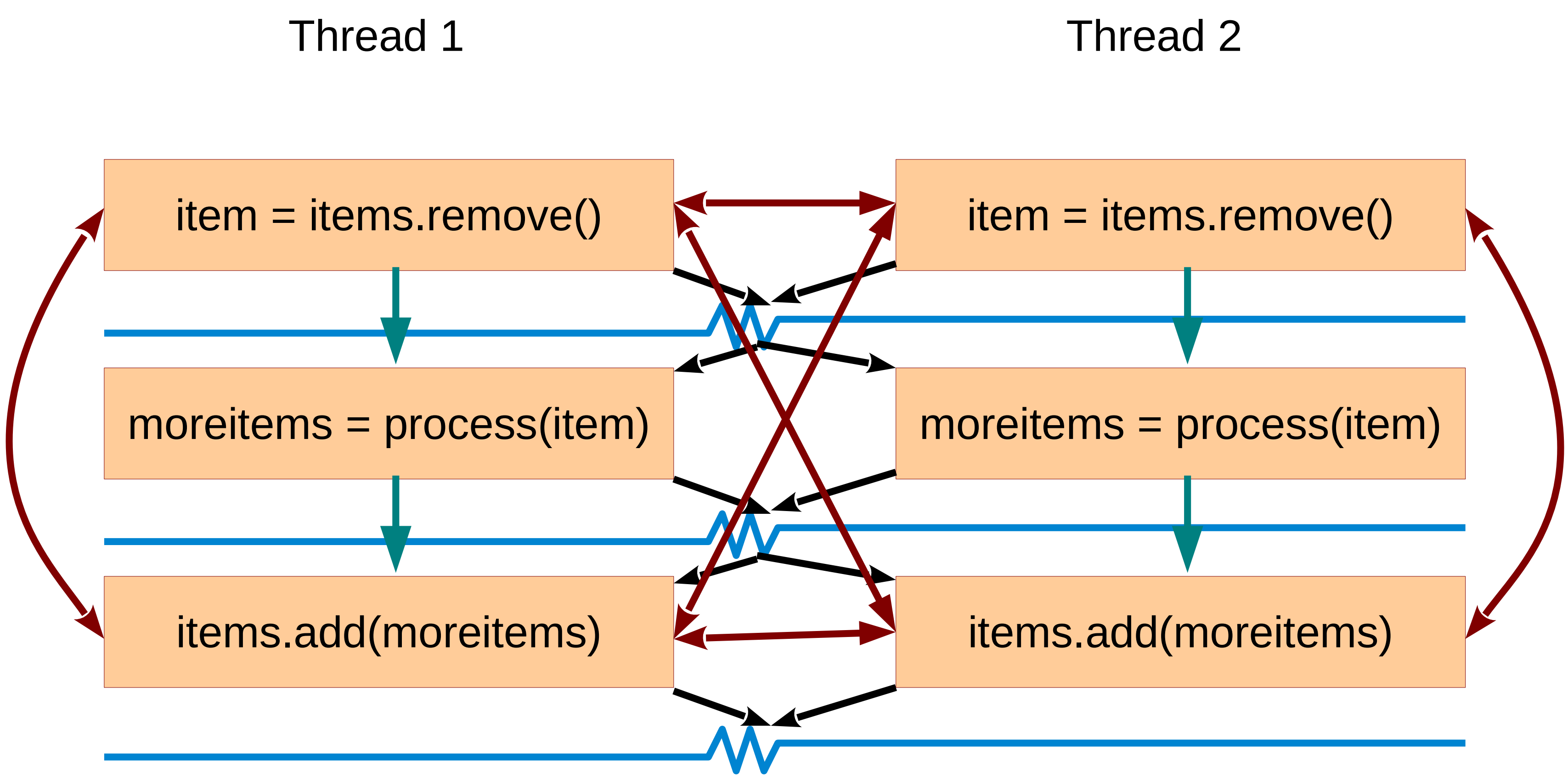
Inserting Barriers

```
if (i == 1 || i == 2 || i == 4 || ...) {
        atomicDec(items.size);
        -- barrier --

        moreitems = process(item);
        -- barrier --

        atomicAdd(items.size, size(moreitems));
        items.addParallel(moreitems);
        -- barrier --
}
```

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.

Barriers and Dependences



Did we add any extra dependences?

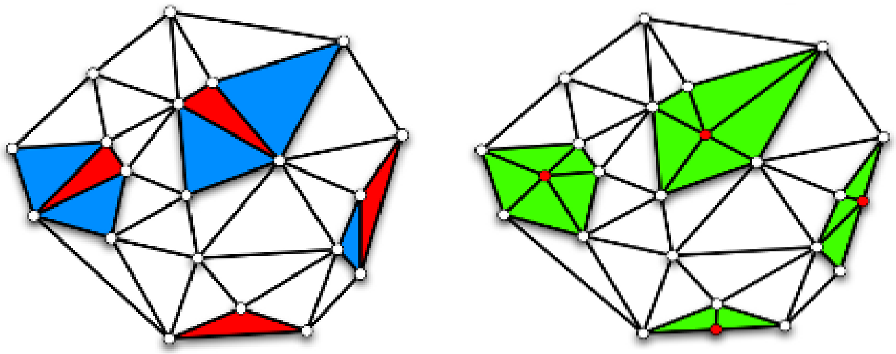
What does the do-while loop achieve when the function is executed concurrently by multiple threads? How would you implement **cleanup()**? Write one application of such a functionality.

```
#define K 29
int gg = 0;
...
void PAmistery {
   do {
      if (gg < K) {
          old = atomicInc(&gg);
          if (old >= K) atomicDec(&gg);
      }
   } while (old >= K);

    // critical section

   cleanup();
}
```

Limitations of Static Parallelization

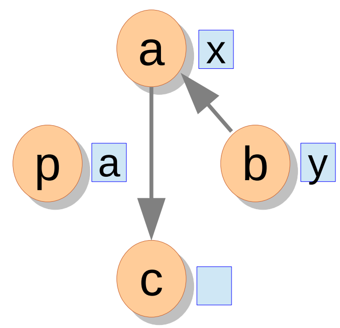
- Some programs cannot be effectively parallelized using static techniques.
  - e.g. graph algorithms, pointer-savvy programs
- Existing static optimization techniques (analyses) are also very conservative for such programs.
- Ineffectiveness of static techniques forces us to use dynamic approaches.

Examples of Graph Algorithms

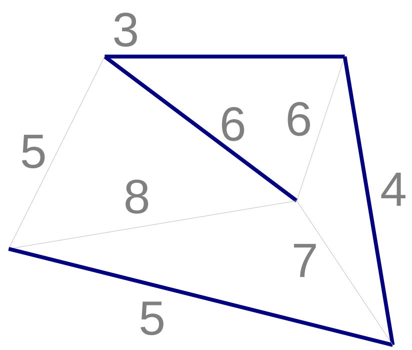


Delaunay Mesh Refinement

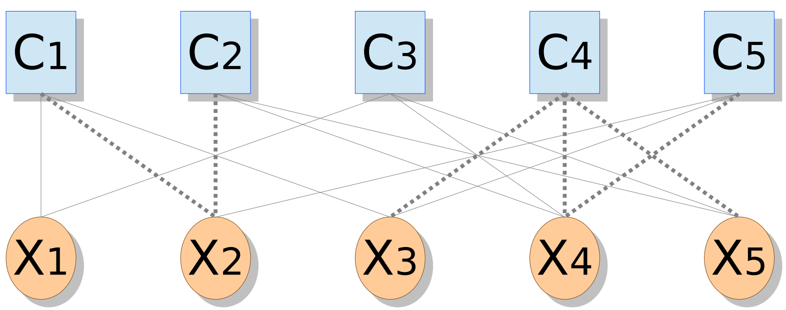
```
a = &x
b = &y
p = &a
*p = b
c = a
```



Points-to Analysis



Minimum Spanning
Tree Computation



Survey Propagation

What is IrRegularitTy?

- Data-access or control patterns are unpredictable at compile time.

Irregular data-access

```
int a[N], b[N], c[N];
readinput(a);

c[5] = b[a[4]];
```

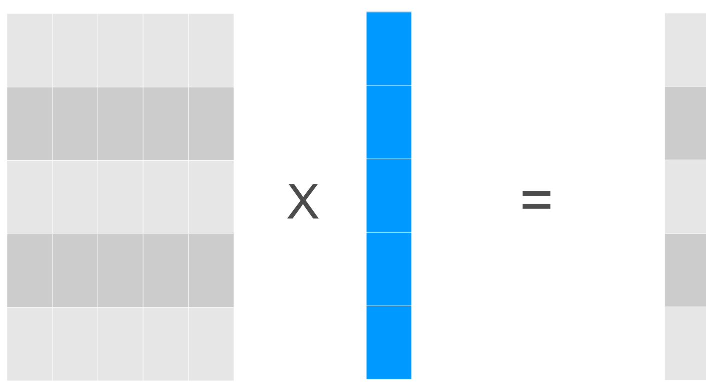
Irregular control-flow

```
int a[N];
readinput(a);

if (a[4] > 30) {
      ...
}
```

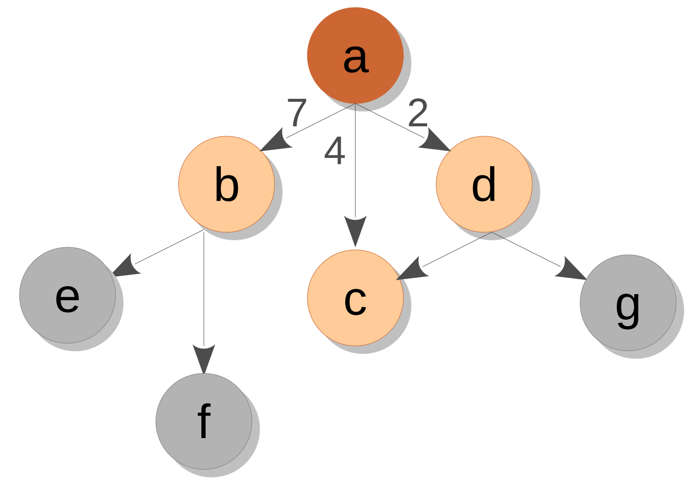
Needs dynamic techniques

Pointer-based data structures often contribute to irregularity.

Regular vs. Irregular Algorithms



Matrix Multiplication

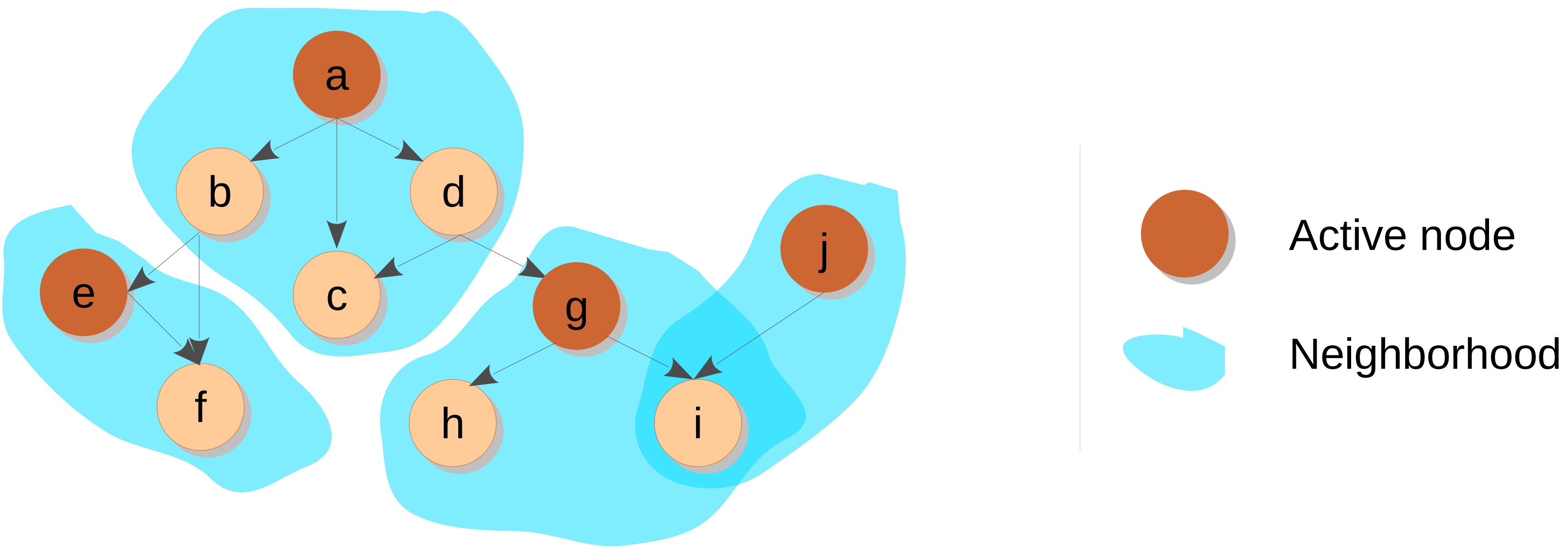


Shortest Paths Computation

By knowing matrix size and its starting address, and ***without knowing its values*** we can determine the dynamic behavior.

Dynamic behavior is ***dependent on the input graph***. This results in pessimistic synchronization.

Dynamic Techniques



Non-overlapping neighborhoods can be processed in parallel.

Overlapping neighborhoods require synchronization.

Leads to optimistic and cautious parallelizations.

Sequential to Parallel

- Sequential programs often overspecify dependencies.

```
for (int ii = 0; ii < N; ++ii) {
	process(a[ii]);
}
```

Processing of a[ii + 1] is specified after that of a[ii].

```
x = y;
f(a, b);
while (m < n) {
	process(m);
	m = next(m);
}
```

Processing of assignment, function call and while are sequentially specified.

We need a way to specify that various operations need not be executed in a specific order.

Unordered Execution

```
for (int ii = 0; ii < N; ++ii) {
	process(a[ii]);
}
```

```
forall (e in a) {
	process(e);
}
```

```
x = y;
f(a, b);
while (m < n) {
	process(m);
	m = next(m);
}
```

```
unordered(
	x = y;
	f(a, b);
	while (m < n) {
		process(m);
		m = next(m);
	}
);
```

Sequential to Parallel

- We added unorderedness.
- We added non-determinism.
- We added higher-level information.

Learning Outcomes

- Recall control and data dependences.
- Recall RAW, WAR and WAW dependences.
- Given a loop with array accesses, formulate an ILP to identify dependences.
- Write simple codes with atomics and barriers.
- Understand the limitations of static parallelization.