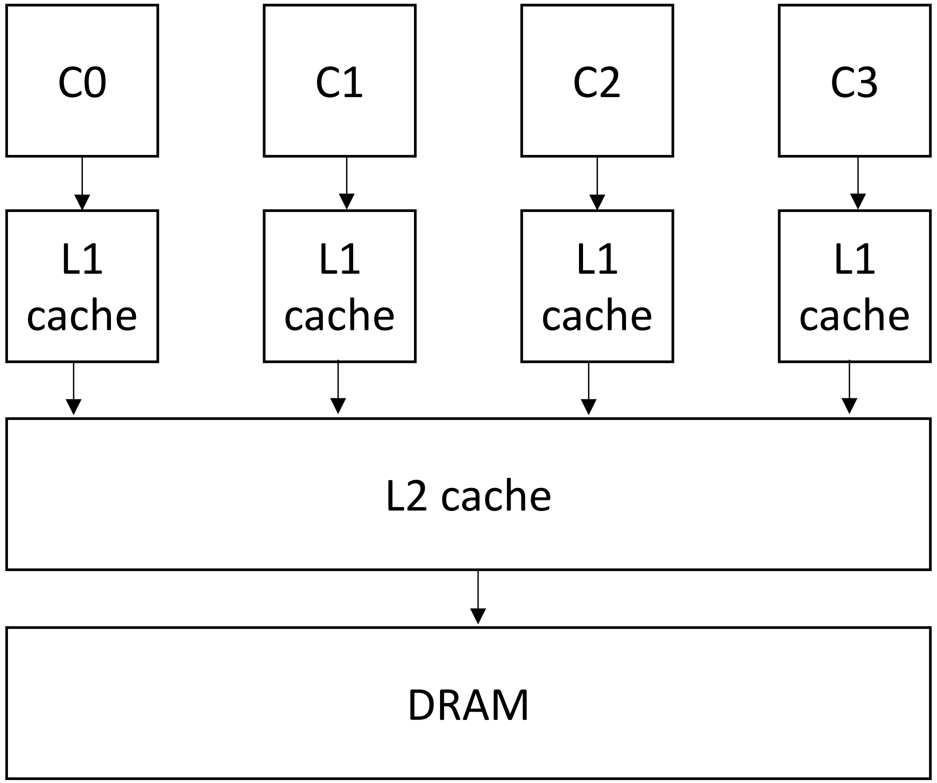
CSE211: Compiler Design

Nov. 3, 2021

- **Topic**: SMP parallelism continued
  - Safety checking
  - Restructuring loops

- **Discussion questions**:
  - Have you used tools to check for data-races?



Announcements

- Midterm is due today!
  - likely won't be answering questions tonight
- Homework 1 grades are out
  - Let me know ASAP if there are any issues
- Homework 3 should be released today (I might need 1 more day...)
  - You have 2 weeks to finish

Paper/Project proposals

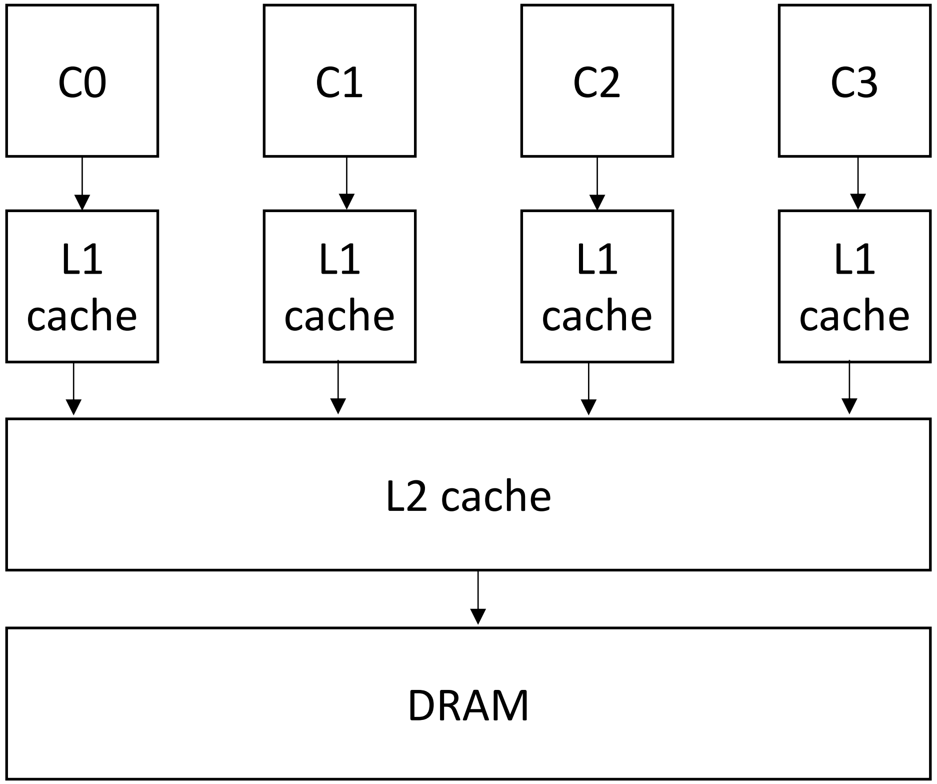
- Please start thinking about these.
  - Message me for recommendations
  - Tell me what you’re interested in so we can find a good fit!
- Proposals due on Nov. 14 (less than 2 weeks)
  - Please be pro-active about this. If you don’t have one in mind, please send me an email with some of your interests ASAP
- Midterm is a good indicator for how the final will be.

CSE211: Compiler Design

Nov. 3, 2021

- **Topic**: SMP parallelism continued
  - Safety checking
  - Restructuring loops

- **Discussion questions**:
  - Have you used tools to check for data-races?



Aside from homework 1

- Parsing with derivatives

Adding ? to parsing with derivatives

re =
- |{}
- | “”
- | *c* (single character)
- | re_{lhs} | re_{rhs}
- | re_{lhs} . re_{rhs}
- | $re_{starred}$ *
- | $re_{optional}$?

re_optional = ”” | re_optional

What is a method for computing NULL?

Consider the recursive cases:

- NULL(*re*) = match re with:
  - re_{lhs} | re_{rhs}

    return NULL(re_{lhs}) | NULL(re_{rhs})
  - $re_{starred}$*

    return “”
  - re_{lhs} . re_{rhs}

    return NULL(re_{lhs}) . NULL(re_{rhs})

- re =
  - | {}
  - | ε
  - | a (single character)
  - | re_{lhs} | re_{rhs}
  - | re_{lhs} . re_{rhs}
  - | $re_{starred}$ *
  - | $re_{optional}$?

Derivative Recursive Cases

Consider the recursive cases:

- $\delta_c\ (re)$ = match re with:
  - $re_{lhs}\ |\ re_{rhs}$

    return $\delta_c(re_{lhs})\ |\ \delta_c\ (re_{rhs})$
  - $re_{starred}{}^*$

    return $\delta_c(re_{starred})\ .\ re_{starred}{}^*$
  - $re_{lhs}\ .\ re_{rhs}$

    return $\delta_c(re_{lhs})\ .\ re_{rhs}\ |$

    $NULL(re_{lhs})\ .\ \delta_c(re_{rhs})$
  - $re_{optional}$? return $\delta_c(re_{optional})$

- re =

  |{}

  | ε

  | a (single character)

  | $re_{lhs}\ |\ re_{rhs}$

  | $re_{starred}\ ^*$

  | $re_{optional}$?

Back to parallelism

Review

- What sorts of components do modern architectures have that allow us to exploit parallelism?

Review

- What sorts of components do modern architectures have that allow us to exploit parallelism?
  - ILP (instruction level parallelism)
  - SMP (symmetric multiprocessing)
- Pros and cons to each?

Review

- How can compilers help with parallelism?
  - ILP
  - SMP

Review

- We are thinking about a special kind of “for” loop, DOALL Loops
    - What are some of the conditions

Review

- We are thinking about a special kind of “for” loop, DOALL Loops
  - What are some of the conditions

  - disjoint arrays
  - bounds from 0 to N
  - only side effects are array writes

Review

- We are thinking about a special kind of “for” loop, DOALL Loops
  - What are some of the conditions

  - disjoint arrays
  - bounds from 0 to N
  - only side effects are array writes

- It is safe to do these loops in parallel if:

Review

- We are thinking about a special kind of “for” loop, DOALL Loops
  - What are some of the conditions

  - disjoint arrays
  - bounds from 0 to N
  - only side effects are array writes

- It is safe to do these loops in parallel if:
  - Loop iterations are independent
  - threads can be assigned different loop iterations

Review

- We are thinking about a special kind of “for” loop, DOALL Loops
  - What are some of the conditions

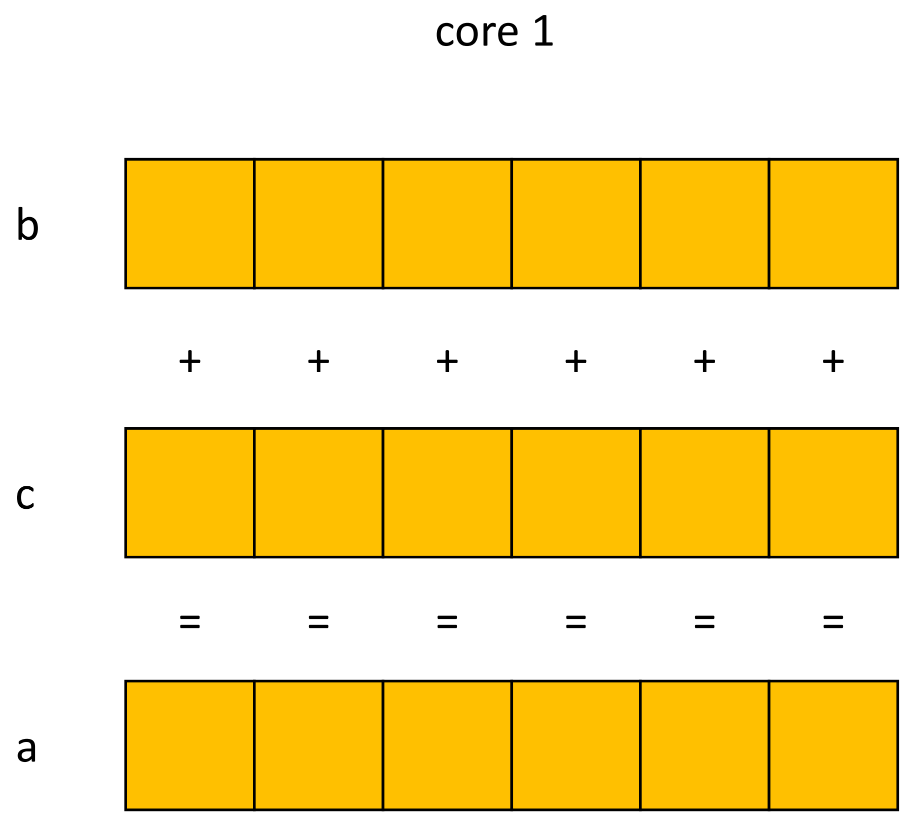
  - disjoint arrays
  - bounds from 0 to N
  - only side effects are array writes

- It is safe to do these loops in parallel if:
  - Loop iterations are independent
  - threads can be assigned different loop iterations

What about performance?

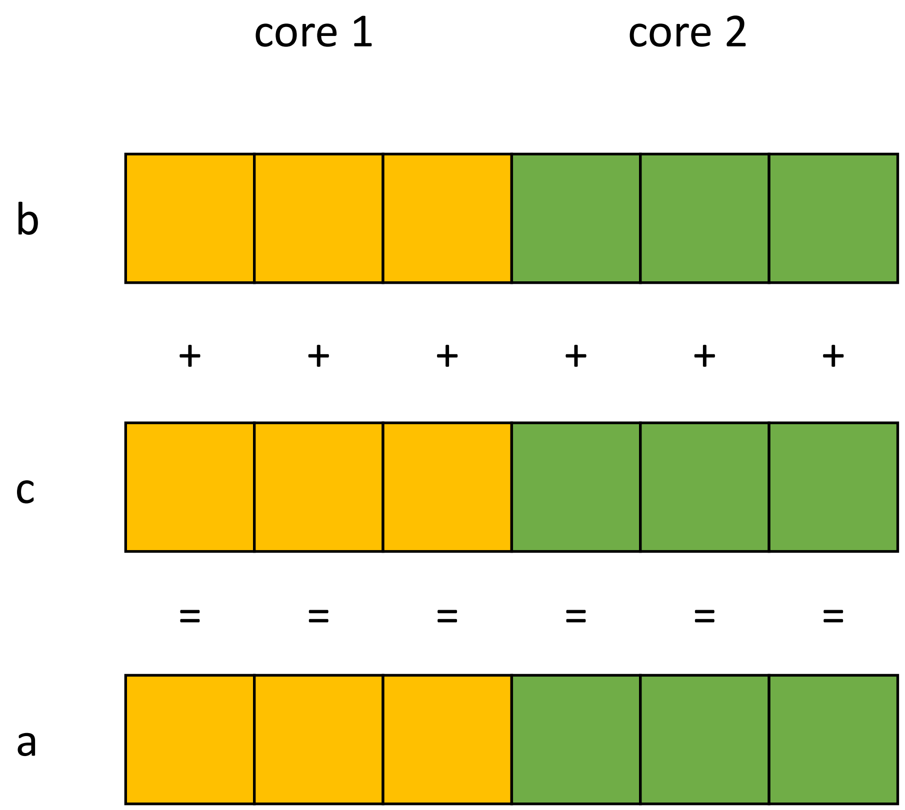
For loops are great candidates for SMP parallelism

```
for (int i = 0; i < 6; i++) {
   a[i] = b[i] + c[i]
}
```



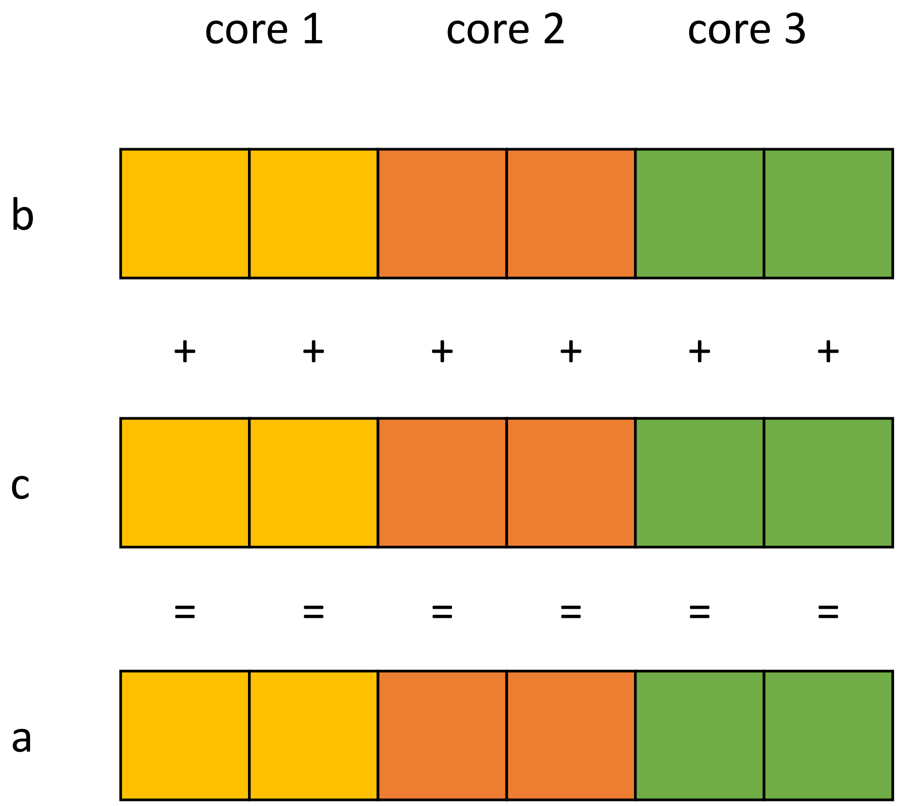
For loops are great candidates for SMP parallelism

```
for (int i = 0; i < 6; i++) {
   a[i] = b[i] + c[i]
}
```



For loops are great candidates for SMP parallelism

```
for (int i = 0; i < 6; i++) {
    a[i] = b[i] + c[i]
}
```



Write-write conflicts

- Criteria: every iteration of the outer-most loop must be *independent*
- the loop must produce the same result for any order of the iterations

```
for (i = 0; i < size; i++) {
    a[index(i)] = loop(i);
}
```

for two distinct iterations:
i_x != i_y
Check:
index(i_x) != index(i_y)

Why?
Because if
index(i_x) == index(i_y)
then:
a[index(i_x)] will equal
either loop(i_x) or loop(i_y)
depending on the order

Write-write conflicts

- Criteria: every iteration of the outer-most loop must be *independent*
- the loop must produce the same result for any order of the iterations

```
for (i = 0; i < size; i++) {
   a[index(i)] = loop(i);
}
```

Examples:

```
for (i = 0; i < 128; i++) {
   a[i]= i*2;
}
```

Write-write conflicts

- Criteria: every iteration of the outer-most loop must be *independent*
- the loop must produce the same result for any order of the iterations

```
for (i = 0; i < size; i++) {
    a[index(i)] = loop(i);
}
```

Examples:

```
for (i = 0; i < 128; i++) {
    a[i]= i*2;
}
```

```
for (i = 0; i < 128; i++) {
    a[i%64]= i*2;
}
```

Read-write conflicts

- Criteria: every iteration of the outer-most loop must be *independent*

```
for (i = 0; i < size; i++) {
   a[write_index(i)] = a[read_index(i)] + loop(i);
}
```

Read-write conflicts:

for two distinct iteration variables:

i_x != i_y

Check:

write_index(i_x) != read_index(i_y)

Read-write conflicts

- Criteria: every iteration of the outer-most loop must be *independent*

```
for (i = 0; i < size; i++) {
    a[write_index(i)] = a[read_index(i)] + loop(i);
}
```

Read-write conflicts:

for two distinct iteration variables:

i_x != i_y

Check:

write_index(i_x) != read_index(i_y)

Why?

if i_x iteration happens first, then iteration i_y reads an updated value.

if i_y happens first, then it reads the original value

Safety Criteria

- Criteria: every iteration of the outer-most loop must be *independent*

```
for (i = 0; i < size; i++) {
   a[write_index(i)] = a[read_index(i)] + loop(i);
}
```

Read-write conflicts:

for two distinct iteration variables:
`i`$_x$ `!=` `i`$_y$
Check:
`write_index(`i_x`) != read_index(`i_y`)`

Example

```
for (i = 0; i < 2; i++) {
   a[i]= a[0]*2;
}
```

Examples:

```
for (i = 0; i < 128; i++) {
   a[i]= a[i]*2;
}
```

Examples:

```
for (i = 0; i < 128; i++) {
   a[i]= a[i]*2;
}
```

```
for (i = 0; i < 128; i++) {
   a[i]= a[0]*2;
}
```

Examples:

```
for (i = 0; i < 128; i++) {
   a[i]= a[i]*2;
}
```

```
for (i = 0; i < 128; i++) {
   a[i]= a[0]*2;
}
```

```
for (i = 1; i < 128; i++) {
   a[i]= a[0]*2;
}
```

Examples:

```
for (i = 0; i < 128; i++) {
   a[i]= a[i]*2;
}
```

```
for (i = 0; i < 128; i++) {
   a[i]= a[0]*2;
}
```

```
for (i = 1; i < 128; i++) {
   a[i]= a[0]*2;
}
```

```
for (i = 0; i < 128; i++) {
   a[i%64]= a[i]*2;
}
```

Examples:

```
for (i = 0; i < 128; i++) {
   a[i]= a[i]*2;
}
```

```
for (i = 0; i < 128; i++) {
   a[i]= a[0]*2;
}
```

```
for (i = 1; i < 128; i++) {
   a[i]= a[0]*2;
}
```

```
for (i = 0; i < 128; i++) {
   a[i%64]= a[i]*2;
}
```

```
for (i = 0; i < 128; i++) {
   a[i%64]= a[i+64]*2;
}
```

Automation?

- We have decent intuition about this, but if its going to be in a compiler, then it needs to be automatable

```
for (i = 0; i < 128; i++) {
   a[i]= a[i]*2;
}
```

two integers: i_x != i_y

i_x >= 0

i_x < 128

i_y >= 0

i_y < 128

write-write conflict write_index(i_x) == write_index(i_y)

read-write conflict write_index(i_x) == read_index(i_y)

Ask if these constraints are satisfiable (if so, it is not safe to parallelize)

Automation?

- We have decent intuition about this, but if its going to be in a compiler, then it needs to be automatable

```
for (i = 0; i < 128; i++) {
   a[i]= a[i]*2;
}
```

two integers: i_x != i_y

$i_x >= 0$

$i_x < 128$

$i_y >= 0$

$i_y < 128$

$i_x == i_y$

$i_x == i_y$

Automation?

- We have decent intuition about this, but if its going to be in a compiler, then it needs to be automatable

```
for (i = 0; i < 128; i++) {
   a[i]= a[i]*2;
}
```

two integers: i_x != i_y

i_x >= 0

i_x < 128

i_y >= 0

i_y < 128

i_x == i_y

i_x == i_y

We can feed these constraints to an SMT Solver!

SMT Solver

- Satisfiability Modulo Theories (SMT)
  - Generalized SAT solver
- Solves many types of constraints over many domains
  - Integers
  - Reals
  - Bitvectors
  - Sets
- Complexity bounds are high (and often undecidable). In practice, they work pretty well

Microsoft Z3

- State-of-the-art
- Python bindings
- Tutorials:
  - Python: https://ericpony.github.io/z3py-tutorial/guide-examples.htm
  - SMT LibV2: https://rise4fun.com/z3/tutorial

Automation?

- We have decent intuition about this, but if its going to be in a compiler, then it needs to be automatable

```
for (i = 0; i < 128; i++) {
   a[i]= a[i]*2;
}
```

two integers: i_x != i_y

i_x >= 0

i_x < 128

i_y >= 0

i_y < 128

i_x == i_y

i_x == i_y

We can feed these constraints to an SMT Solver!

Another example:

```
for (i = 0; i < 128; i++) {
   a[i%64]= a[i+64]*2;
}
```

Another example:

```
for (i = 0; i < 128; i++) {
   a[i%64]= a[i+64]*2;
}
```

two integers: i_x != i_y

i_x >= 0

i_x < 128

i_y >= 0

i_y < 128

i_x % 64 == i_y % 64

Another example:

```
for (i = 0; i < 128; i++) {
   a[i%64]= a[i+64]*2;
}
```

two integers: i_x != i_y
i_x >= 0
i_x < 128
i_y >= 0
i_y < 128
i_x % 64 == i_y % 64

what about write-read?

Another example:

```
for (i = 0; i < 128; i++) {
   a[i%64]= a[i+64]*2;
}
```

two integers: $i_x \neq i_y$

$i_x \geq 0$

$i_x < 128$

$i_y \geq 0$

$i_y < 128$

$i_x \,\%\, 64 == i_y + 64$

what about write-read?

General formula:

```
for (int i0 = init0; i0 < bound0(); i0++) {
    for (int i1 = init1(i0); i1 < bound1(i0); i1++) {
       ...
       for (int iN = initN(i0, i1, ...); iN < boundN(i0, i1 ...); iN++) {
            write(a, write_index(i0, i1 .. iN))
            read(a, read_index(i0, i1 .. iN));
       }
   }
}
```

General formula:

```
for (int i0 = init0; i0 < bound0(); i0++) {
    for (int i1 = init1(i0); i1 < bound1(i0); i1++) {
       ...
       for (int iN = initN(i0, i1, ...); iN < boundN(i0, i1 ...); iN++) {
            write(a, write_index(i0, i1 .. iN))
            read(a, read_index(i0, i1 .. iN));
       }
    }
}
```

1. Create two variables for each loop variable: $i0_x$, $i0_y$, $i1_x$, $i1_y$ ...
Set outer loop: $i0_x$!= $i0_y$

2. Constrain them to be inside their bounds:
for w in from (0,N): $iw_{x,y}$ >= initw(...), $iw_{x,y}$ < boundN(...)

3. Enumerate all pairs of potential write-write conflicts:
check: write_index($i0_x$,$i1_x$ ...) == write_index ($i0_y$,$i1_y$ ...)

4. Do the same for write-read conflicts

General formula:

```
for (int i0 = init0; i0 < bound0(); i0++) {
    for (int i1 = init1(i0); i1 < bound1(i0); i1++) {
        ...
        for (int iN = initN(i0, i1, ...); iN < boundN(i0, i1 ...); iN++) {
            write(a, write_index(i0, i1 .. iN))
            read(a, read_index(i0, i1 .. iN));
        }
    }
}
```

1. Create two variables for each loop variable: $i0_x$, $i0_y$, $i1_x$, $i1_y$ ...

Set outer loop: $i0_x \neq i0_y$

2. Constrain them to be inside their bounds:

for w in from (0,N): $iw_{x,y}$ >= initw(...), $iw_{x,y}$ < boundN(...)

3. Enumerate all pairs of potential write-write conflicts:

check: write_index($i0_x$, $i1_x$ ...) == write_index ($i0_y$, $i1_y$ ...)

4. Do the same for write-read conflicts

What if we want to parallelize an inner loop?

Are data races ever okay?

- Thoughts?

Are data races ever okay?

- Consider this program:

```
int x = 0;
for (int i = 0; i < 1024; i++) {
    int tmp = *(&x);
    tmp += 1;
    *(&x) = tmp;
}
```

What can go wrong if we run the loop in parallel?

December 28, 2011
Volume 9, issue 12

PDF

You Don’t Know Jack about Shared Variables or Memory Models

Data races are evil.

Hans-J. Boehm, HP Laboratories, Sarita V. Adve, University of Illinois at Urbana-Champaign

The final count can also be too high. Consider a case in which the count is bigger than a machine word. To avoid dealing with binary numbers, assume we have a decimal machine in which each word holds three digits, and the counter x can hold six digits. The compiler translates x++ to something like

```
tmp_hi = x_hi;
tmp_lo = x_lo;
(tmp_hi, tmp_lo)++;
x_hi = tmp_hi;
x_lo = tmp_lo;
```

Now assume that x is 999 (i.e., `x_hi = 0`, and `x_lo = 999`), and two threads, a blue and a red one, each increment x as follows (remember that each thread has its own copy of the machine registers `tmp_hi` and `tmp_lo`):

```
tmp_hi = x_hi;
tmp_lo = x_lo;
(tmp_hi, tmp_lo)++;   //tmp_hi = 1, tmp_lo = 0
x_hi = tmp_hi;        //x_hi = 1, x_lo = 999, x = 1999
          x++;        //red runs all steps
                      //x_hi = 2, x_lo = 0, x = 2000
x_lo = tmp_lo;        //x_hi = 2, x_lo = 0
```

Horrible data races in the real world

Therac 25: a radiation therapy machine

- Between 1987 and 1989 a software bug caused 6 cases where radiation was massively overdosed
- Patients were seriously injured and even died.
- Bug was root caused to be a data race.
- https://en.wikipedia.org/wiki/Therac-25

Horrible data races in the real world

2003 NE power blackout

- second largest power outage in history: 55 million people were effected
- NYC was without power for 2 days, estimated 100 deaths
- Root cause was a data race
- https://en.wikipedia.org/wiki/Northeast_blackout_of_2003

But checking for data conflicts is hard...

- Tools are here to help (Professor Flanagan is famous in this area)

- My previous group:
  - “Dynamic Race Detection for C++11” Lidbury and Donaldson
  - Scalable (complete) race detection
    - Firefox has ~40 data races
    - Chromium has ~6 data races

Next class

- Topics:
  - Restructuring loops

- Remember:
  - Midterm is due today by midnight, please don’t be late!
  - Homework 3 assigned today (or tomorrow)