Lecture 11—Dependencies

January 28, 2015

Roadmap

Last Time: C++ atomics; C Compilers
This Time: Dependencies

Atomics when not using C++11

Not really.

gcc supports atomics via extensions:

`https://gcc.gnu.org/onlinedocs/gcc/_005f_005fatomic-Builtins.html`

OS X has atomics via OS calls:

`https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/Multithreading/ThreadSafety/ThreadSafety.html`

etc. . .

Reference:

`http://stackoverflow.com/questions/1130018/unix-portable-atomic-operations`

Part I

Dependencies

Next topic: Dependencies

Dependencies are the main limitation to parallelization.

Example: computation must be evaulated as `XY` and not `YX`.

Not synchronization

Assume (for now) no synchronization problems.

Only trying to identify code that is safe to run in parallel.

Dependencies: Analogies

Must extract bicycle from garage before closing garage door.

Must close washing machine door before starting the cycle.

Must be called on before answering questions? (sort of)

Students must submit assignment before course staff can mark the assignment.

Memory-carried Dependencies

Dependencies limit the amount of parallelization.

Can we execute these 2 lines in parallel?

```
x = 42
x = x + 1
```

Memory-carried Dependencies

Dependencies limit the amount of parallelization.

Can we execute these 2 lines in parallel?

```
x = 42
x = x + 1
```

No.

- Assume x initially 1. What are possible outcomes?

Memory-carried Dependencies

Dependencies limit the amount of parallelization.

Can we execute these 2 lines in parallel?

```
x = 42
x = x + 1
```

No.

- Assume x initially 1. What are possible outcomes?
  $x = 43$ or $x = 42$

Next, we'll classify dependencies.

Read After Read (RAR)

Can we execute these 2 lines in parallel? (initially x is 2)

```
y = x + 1
z = x + 5
```

Read After Read (RAR)

Can we execute these 2 lines in parallel? (initially x is 2)

```
y = x + 1
z = x + 5
```

Yes.

- Variables y and z are independent.
- Variable x is only read.

RAR dependency allows parallelization.

Read After Write (RAW)

What about these 2 lines? (again, initially x is 2):

```
x = 37
z = x + 5
```

Read After Write (RAW)

What about these 2 lines? (again, initially x is 2):

```
x = 37
z = x + 5
```

No, $z = 42$ or $z = 7$.

RAW inhibits parallelization: can't change ordering.
Also known as a true dependency.

Write After Read (WAR)

What if we change the order now? (again, initially x is 2)

```
z = x + 5
x = 37
```

Write After Read (WAR)

What if we change the order now? (again, initially x is 2)

```
z = x + 5
x = 37
```

No. Again, $z = 42$ or $z = 7$.

- WAR is also known as a anti-dependency.
- But, we can modify this code to enable parallelization.

Removing Write After Read (WAR) Dependencies

Make a copy of the variable:

```
x_copy = x
z = x_copy + 5
x = 37
```

Removing Write After Read (WAR) Dependencies

Make a copy of the variable:

```
x_copy = x
z = x_copy + 5
x = 37
```

We can now run the last 2 lines in parallel.

- Induced a true dependency (RAW) between first 2 lines.
- Isn't that bad?

Removing Write After Read (WAR) Dependencies

Make a copy of the variable:

```
x_copy = x
z = x_copy + 5
x = 37
```

We can now run the last 2 lines in parallel.

- Induced a true dependency (RAW) between first 2 lines.
- Isn't that bad?

Not always:

```
z = very_long_function(x) + 5
x = very_long_calculation()
```

Write After Write (WAW)

Can we run these lines in parallel? (initially x is 2)

```
z = x + 5
z = x + 40
```

Write After Write (WAW)

Can we run these lines in parallel? (initially x is 2)

```
z = x + 5
z = x + 40
```

Nope, $z = 42$ or $z = 7$.

- WAW is also known as an output dependency.
- We can remove this dependency (like WAR):

Write After Write (WAW)

Can we run these lines in parallel? (initially x is 2)

```
z = x + 5
z = x + 40
```

Nope, z = 42 or z = 7.

- WAW is also known as an output dependency.
- We can remove this dependency (like WAR):

```
z_copy = x + 5
z = x + 40
```

Summary of Memory-carried Dependencies

		Second Access	
		Read	**Write**
First Access	**Read**	No Dependency Read After Read (RAR)	Anti-dependency Write After Read (WAR)
	Write	True Dependency Read After Write (RAW)	Output Dependency Write After Write (WAW)

Part III

Loop-carried Dependencies

Loop-carried Dependencies (1)

Can we run these lines in parallel?
(initially a[0] and a[1] are 1)

```
a[4] = a[0] + 1
a[5] = a[1] + 2
```

Loop-carried Dependencies (1)

Can we run these lines in parallel?
(initially a[0] and a[1] are 1)

```
a[4] = a[0] + 1
a[5] = a[1] + 2
```

Yes.

- There are no dependencies between these lines.
- However, this is not how we normally use arrays. . .

Loop-carried Dependencies (2)

What about this? (all elements initially 1)

```
for (int i = 1; i < 12; ++i)
    a[i] = a[i-1] + 1
```

Loop-carried Dependencies (2)

What about this? (all elements initially 1)

```
for (int i = 1; i < 12; ++i)
    a[i] = a[i-1] + 1
```

No, a[2] = 3 or a[2] = 2.

- Statements depend on previous loop iterations.
- An example of a loop-carried dependency.

Loop-carried Dependencies (3)

Can we parallelize this? (again, all elements initially 1)

```
for (int i = 4; i < 12; ++i)
    a[i] = a[i-4] + 1
```

Loop-carried Dependencies (3)

Can we parallelize this? (again, all elements initially 1)

```
for (int i = 4; i < 12; ++i)
    a[i] = a[i-4] + 1
```

Yes, to a degree.

- We can execute 4 statements in parallel:
  - $a[4] = a[0] + 1$, $a[8] = a[4] + 1$
  - $a[5] = a[1] + 1$, $a[9] = a[5] + 1$
  - $a[6] = a[2] + 1$, $a[10] = a[6] + 1$
  - $a[7] = a[3] + 1$, $a[11] = a[7] + 1$

Loop-carried Dependencies (3)

Can we parallelize this? (again, all elements initially 1)

```
for (int i = 4; i < 12; ++i)
    a[i] = a[i-4] + 1
```

Yes, to a degree.

- We can execute 4 statements in parallel:
    - a[4] = a[0] + 1, a[8] = a[4] + 1
    - a[5] = a[1] + 1, a[9] = a[5] + 1
    - a[6] = a[2] + 1, a[10] = a[6] + 1
    - a[7] = a[3] + 1, a[11] = a[7] + 1

Always consider dependencies between iterations.

Larger example: Loop-carried Dependencies

```
// Repeatedly square input, return number of iterations before
// absolute value exceeds 4, or 1000, whichever is smaller.
int inMandelbrot(double x0, double y0) {
  int iterations = 0;
  double x = x0, y = y0, x2 = x*x, y2 = y*y;
  while ((x2+y2 < 4) && (iterations < 1000)) {
    y = 2*x*y + y0;
    x = x2 - y2 + x0;
    x2 = x*x; y2 = y*y;
    iterations++;
  }
  return iterations;
}
```

How can we parallelize this?

Larger example: Loop-carried Dependencies

```
// Repeatedly square input, return number of iterations before
// absolute value exceeds 4, or 1000, whichever is smaller.
int inMandelbrot(double x0, double y0) {
  int iterations = 0;
  double x = x0, y = y0, x2 = x*x, y2 = y*y;
  while ((x2+y2 < 4) && (iterations < 1000)) {
    y = 2*x*y + y0;
    x = x2 - y2 + x0;
    x2 = x*x; y2 = y*y;
    iterations++;
  }
  return iterations;
}
```

How can we parallelize this?

- Run `inMandelbrot` sequentially for each point, but parallelize different point computations.