# Lecture 23 - Midterm Solution/Review

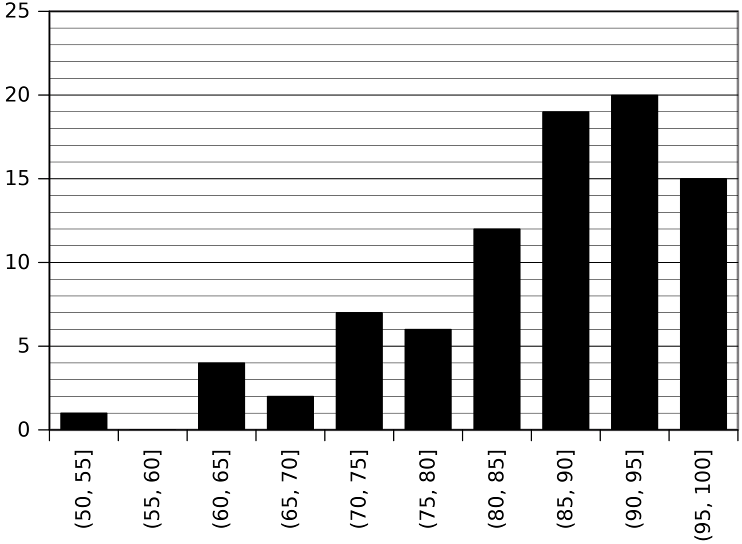
ECE 459: Programming for Performance

Jon Eyolfson

University of Waterloo

March 7, 2012

## Midterm Distribution



## Midterm Results

**Average:** 86

- **Question 1 Average:** 92
- **Question 2 Average:** 80
- **Question 3 Average:** 90
- **Question 4 Average:** 82

If you have any problem with the grading, you have 1 week to bring it up

## Question 1

Definitions

- All definitions are in lectures 3-7, I'll just include one

**Task parallelism** is the simultaneous execution on multiple cores of many different functions across the same or different datasets.

**Data parallelism (aka SIMD)** is the simultaneous execution on multiple cores of the same function across the elements of a dataset.

## Question 2

Consider a program which has a serial and parallelizable component. The serial component executes in 2 seconds and the parallel component executes in 8 seconds, for a total runtime of 10 seconds **when run serially**.

**Part 1**
Assume the problem size is **fixed**. Calculate how many processors would you need to reach a desired speedup of 3.75.

## Question 2 Part 1 Solution

Using fractions:

$S = 0.2$

$P = 0.8$

$speedup = \frac{1}{S+\frac{P}{N}}$

$0.2 + \frac{0.8}{N} = \frac{1}{3.75}$

$\frac{0.8}{N} = \frac{4}{15} - \frac{3}{15}$

$N = \frac{\frac{12}{15}}{\frac{1}{15}}$

$N = 12$

## Question 2 Part 2

For the same problem, now assume the problem size is not fixed. We have **4 processors** and the serial runtime is **constant** for any problem size (larger problems only require that the parallel component runs for longer), which is 2 seconds.

Calculate how long the parallel component would have to run in a **parallel execution** (in seconds) to reach the speedup of 3.75.

Next, calculate how long the parallel component would run in a **serial execution** under the same conditions.

Finally, assuming the problem size scales linearly with the execution time, how many times larger did we have to make the problem until we saw our desired scaling?

## Question 2 Part 2 Solution (1)

Again, using fractions:

$S(n) = \frac{2}{2+x}$

$P(n) = \frac{x}{2+x}$

where $x$ is the runtime in the parallel execution (in seconds)

Plug in the numbers into the equation:

$speedup = S(n) + N \cdot P(n)$

$3.75 = \frac{2}{2+x} + 4 \cdot \frac{x}{2+x}$

$4x + 2 = 7.5 + 3.75x$

$x = \frac{5.5}{0.25} = 22$

## Question 2 Part 2 Solution (2)

$x = \frac{5.5}{0.25} = 22$

$x$ is the runtime in the parallel execution (in seconds)

**Parallel component, time of execution in parallel (s)** $= 22$

**Parallel component, time of execution in serial (s)**
$= 22 \cdot 4 = 88$

**How many times larger is the problem**
**= Above answer / 8 (s)** $= 11$

## Question 3

Consider the following code:

```
void swap(int *x, int *y) {
    int t = *x;
    *x = *y;
    *y = t;
}
```

For this question, assume the following global variables: `int a = 1, b = 2, c = 3`.

**Part 1**
Let's say we have two threads, one calls `swap(&b, &a)` and the other `swap(&b, &c)`. Write down all possible expected outputs of the values of the variables after the calls complete, assuming the code is **thread-safe**.

## Question 3 Part 1 Solution

If a function is thread-safe, it's effect is going to be the same as if the threads executed the function one after another in an undefined order

`swap(&b, &a)` first - a = 2, b = 3, c = 1
`swap(&b, &c)` first - a = 3, b = 1, c = 2
are the two expected outputs

## Question 3 Part 2

Assume each line of code is atomic. Write down an interleaving of the code in two separate threads that shows this function is not thread-safe along with the final output. That is, your output should not match either expected result you found in Part 1.

## Question 3 Part 2 Solution

| `swap(&b, &a)` | `swap(&b, &c)` | **a** | **b** | **c** |
|---|---|---|---|---|
| t = b (2) | | 1 | 2 | 3 |
| | t = b (2) | 1 | 2 | 3 |
| | b = c | 1 | 3 | 3 |
| | c = t (2) | 1 | 3 | 2 |
| b = a | | 1 | 1 | 2 |
| a = t (2) | | 2 | 1 | 2 |

This is not an expected outputs (since there is a data-race to `b`)

**Reminder:** you can spot a data-race by two accesses to the same memory location, one of which is a write

## Question 3 Part 3

Explain briefly (one sentence) what the `restrict` keyword does. Assume we declare both pointers with a `restrict` keyword. Would this make the function thread-safe? Explain why or why not. If it does make the function thread-safe, ignore the locking portion when completing Part 4.

## Question 3 Part 3 Solution

The `restrict` keyword tells the compiler that a pointer will never alias (point to the same location) for the lifetime of the pointer

It would not make this function thread-safe. For these function calls, declaring the pointers as `restrict` would change nothing. For each call in the question, each pointer does point to a different location already. Letting the compiler know wouldn't change anything to do with concurrency.

`restrict` is present to allow the compiler to make optimizations to serial code. You are responsible for ensuring there are no data-races. (Insert Smokey Bear reference here).

## Question 3 Part 4

Assume in your code you create a mutex, `m`, and properly destroy it when the program ends(!). Write below where you would lock and unlock the mutex to make the code thread-safe (pseudocode like `lock(m)` is okay). Use the finest-grain locking possible with a single lock.

Show what happens with the same interleaving (that you used in **Part 2**) on your modified code, now with locks. Verify you get one of the expected outputs. Explain why the code is now thread-safe in all cases.

My intention of “all cases” was for any arguments, but I wasn’t clear enough

## Question 3 Part 4 Solution

```
1 void swap(int *x, int *y) {
2     lock(m);
3     int t = *x;
4     *x = *y;
5     *y = t;
6     unlock(m);
7 }
```

This code is thread-safe, because with a lock around the entire function, it follows the definition

It is thread-safe in all cases because there is no data-races, no two threads can access the same location (with one being a write)

A lock only around lines 3 and 4 above, will not protect any data races involving the location of `y`. For the given example, `y` is a different location and is not involved in a race. **Example: switch the arguments in one, or both calls, you should see a lock around lines 3 and 4 is not thread-safe**

## Question 3 Part 4 Solution

| `swap(&b, &a)` | `swap(&b, &c)` | **a** | **b** | **c** |
|---|---|---|---|---|
| lock(m) | | 1 | 2 | 3 |
| t = b (2) | | 1 | 2 | 3 |
| | blocked | 1 | 2 | 3 |
| b = a | | 1 | 1 | 3 |
| a = t (2) | | 2 | 1 | 3 |
| unlock(m) | | 2 | 1 | 3 |
| | lock(m) | 2 | 1 | 3 |
| | t = b (1) | 2 | 1 | 3 |
| | b = c | 2 | 3 | 3 |
| | c = t (1) | 2 | 3 | 1 |
| | unlock(m) | 2 | 3 | 1 |

## Question 4 Part 1 (1)

Consider the following code:

```
z = long_calc_a(x)
x = long_calc_b(y)
i = long_calc_a(x) + z
j = long_calc_c(x) + y
```

List all RAW (read after write), WAR (write and read) and WAW (write after write) dependencies of the variables, along with their type in the provided table below.

## Question 4 Part 1 Solution (1)

| **Variable** | **First Line** | **Second Line** | **Type of Dependency** |
|---|---|---|---|
| x | 1 | 2 | WAR (write after read) |
| z | 1 | 3 | RAW (read after write) |
| x | 2 | 3 | RAW (read after write) |
| x | 2 | 4 | RAW (read after write) |

## Question 4 Part 1 (2)

Now, assume the following runtimes for each function (which do not executive speculatively):

`long_calc_a` = 2 seconds
`long_calc_b` = 1 second
`long_calc_c` = 3 seconds

Show how you minimize the runtime of the code using any number of processors. Assume the cost of reading/writing/copying variables is 0. Clearly identify what lines can run in parallel.

What is the minimum amount of time required to execute the code in parallel?

## Question 4 Part 1 Solution (2)

We can remove the dependency between lines 1 and 2 by making a copy of `x`

Line 3 depends on lines 1 and 2

Line 4 depends only on line 2

Therefore, we can execute lines 1 and 2 in parallel

We can execute line 4 right after line 2 completes

We can execute line 3 right after lines 1 and 2 complete

Line 3 completes in 2 seconds after line 1 (slowest, 2 seconds) for a total of 4 seconds

Line 4 completes in 3 seconds after line 2 (1 second) for a total of 4 seconds

Therefore the minimum runtime is the maximum between lines 3 and 4, which is 4 seconds

## Question 4 Part 2 (1)

Consider the following code:

```
for (int i = 0; i < 5; ++i) {
    a[i + 1] = a[i % 3];
}
```

Show how you would minimize the runtime of the loop using any number of processors. Assume that each iteration of the loop takes 1 unit of time. Clearly identify which iterations are run sequentially/in parallel. For parallel execution, show which processor executes each iteration.

What is the minimum amount of time required to execute the code in parallel?

## Question 4 Part 2 Solution (1)

Let's unroll the loop to get an idea what's going on

```
0  a[1] = a[0];
1  a[2] = a[1];
2  a[3] = a[2];
3  a[4] = a[0];
4  a[5] = a[1];
```

`i = 1` and `i = 4` depends on `i = 0`

`i = 2` depends on `i = 1`

Therefore you're going to need a least 3 time units, as you're bounded by the dependencies

Naive answer is to just throw all iterations but `i = 3` into a thread to execute sequentially

## Question 4 Part 2 Solution cont. (1)

You may also notice that there are only 3 read memory locations 0, 1, 2

There are two options for showing how you can execute all iterations in 3 time units:

**Option 1:**

Do `i = 0`, then `i = 1` sequentially. Now there are no dependencies, each iteration can go in it's own thread

**Option 2:**

Do all operations that read element 0 in different threads, have an implicit barrier, then do all the operations that read element 1 in different threads, do another implicit barrier, then finally do all the operations that read element 2

## Question 4 Part 2 (2)

What is the minimum amount of time required to execute the code in parallel if there are 100 iterations of the loop (assuming again, there's an unlimited number of processors)?

My intention was to give you an example that would make giving this answer with the naive solution too messy so you would rethink how you did the question.

## Question 4 Part 2 Solution (2)

Following our options from the first part, if we do more iterations that will not change our runtime (it will require more processors however). Our runtime will remain as 3 time units.

**Option 1:**
Requires 98 processors

**Option 2:**
Requires 34 processors