Introduction to Multiprocessors

- Why Multiprocessors?

A bit of history...

- Back to the early 2000s....

- Intel supposed to come up their with their new processors (Tejas and Jayhawk) clocked at 5-10 GHz.

A bit of history...

- Back to the early 2000s....

- Intel supposed to come up with their new processors (Tejas and Jayhawk) clocked at 5-10 GHz.

- Instead:

Enter multiprocessors

Why multiprocessors?

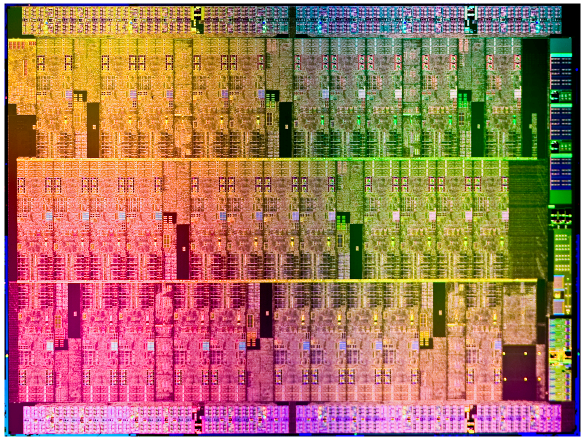
- ILP Wall
  - Limitation of ILP in programs
  - Complexity of superscalar design
- Power Wall
  - ~100W/chip with conventional cooling
- Cost-effectiveness:
  - Easier to connect several ready processors than designing a new, more powerful, processors

Chip multiprocessors (CMPs):

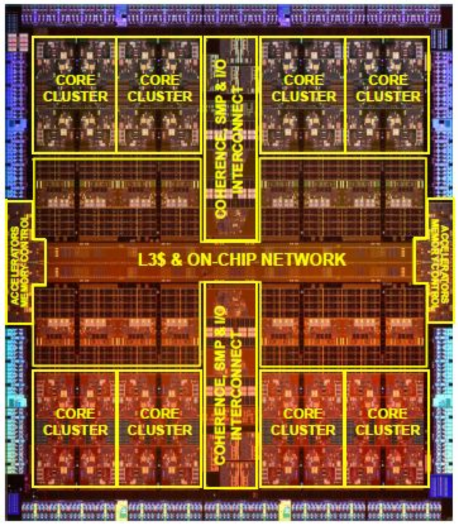
the dividends of Moore's Law

- Billions of transistors per chip affords many (10-100s) of cores

Today's Chip Multiprocessors



Intel Xeon Phi: 72 cores
(aka Knight's Landing)


CORE CLUSTER
CORE CLUSTER
COHERENCE, SMP & I/O INTERCONNECT
CORE CLUSTER
CORE CLUSTER
ACCELERATORS MEMORY CONTROL
L3$ & ON-CHIP NETWORK
ACCELERATORS MEMORY CONTROL
COHERENCE, SMP & I/O INTERCONNECT
CORE CLUSTER
CORE CLUSTER
CORE CLUSTER
CORE CLUSTER

Oracle M7:
32 cores

Exynos 7 (Samsung): 8 cores

But…

Software must expose the parallelism

- Programmers need to write parallel programs
- Legacy code need to be parallelized

…as hard as any (problem) that computer science has faced.

John Hennessy: recipient of the 2018 Turing Award

Amdahl's Law and Efficiency

- Let: F $\rightarrow$ fraction of problem that can be parallelized
  S_{par} $\rightarrow$ speedup obtained on parallelized fraction
  P $\rightarrow$ number of processors

$$S_{overall} = \frac{1}{(1 - F) + \frac{F}{S_{par}}} \qquad \text{Efficiency} = \frac{S_{overall}}{P}$$

- e.g.: 16 processors (S_{par} = 16), F = 0.9 (90%),

$$S_{overall} = \frac{1}{(1 - 0.9) + \frac{0.9}{16}} = 6.4 \qquad \text{Efficiency} = \frac{6.4}{16} = 0.4\ (40\%)$$

Parallelism not always easy or free

- "Embarrassing" parallelism: little effort is required to generate a correct, completely parallel algorithm
  - E.g. find a unique key in an unsorted dataset.
    Each thread processes a fixed number of sequential elements until a key is found or dataset is exhausted.

- But what if threads need to communicate?
  - E.g., producer-consumer communication
    Consider a database query in which one thread extracts students taking a particular class, and passes the results to another thread that computes their GPA.

Inter-processor Communication Models

- Shared memory

Producer (p1)

```
flag = 0;
…
data = 10;
flag = 1;
```

Consumer (p2)

```
flag = 0;
…
while (!flag) {}
x = data * y;
```

Inter-processor Communication Models

- Shared memory

Producer (p1)

```
flag = 0;
…
data = 10;
flag = 1;
```

Consumer (p2)

```
flag = 0;
…
while (!flag) {}
x = data * y;
```

- Message passing

Producer (p1)

```
…
data = 10;
send(p2, data, label);
```

Consumer (p2)

```
…
receive(p1, b, label);
x = b * y;
```

Shared Memory: Pros & Cons

- Shared memory pros
  - Easier to program
    - correctness first, performance later
- Shared memory cons
  - Synchronization complex
  - Communication implicit → harder to optimize
  - **Must guarantee coherence**

HW Support for Shared Memory

- Cache Coherence
  - Caches + multiprocessers → stale values
  - System must behave correctly in the presence of caches → RAW, WAR and WAW dependencies must be observed across <u>all</u> threads
    - Operations are on memory addresses: renaming not an option
- Memory Consistency
  - How are memory operations to <u>different</u> memory addresses orders?
- Primitive synchronization
  - Memory fences: memory ordering on demand
  - Atomic operations (e.g., Read-Modify-Write): support for locks (to protect critical sections)

Cache Coherence

Producer (p1)

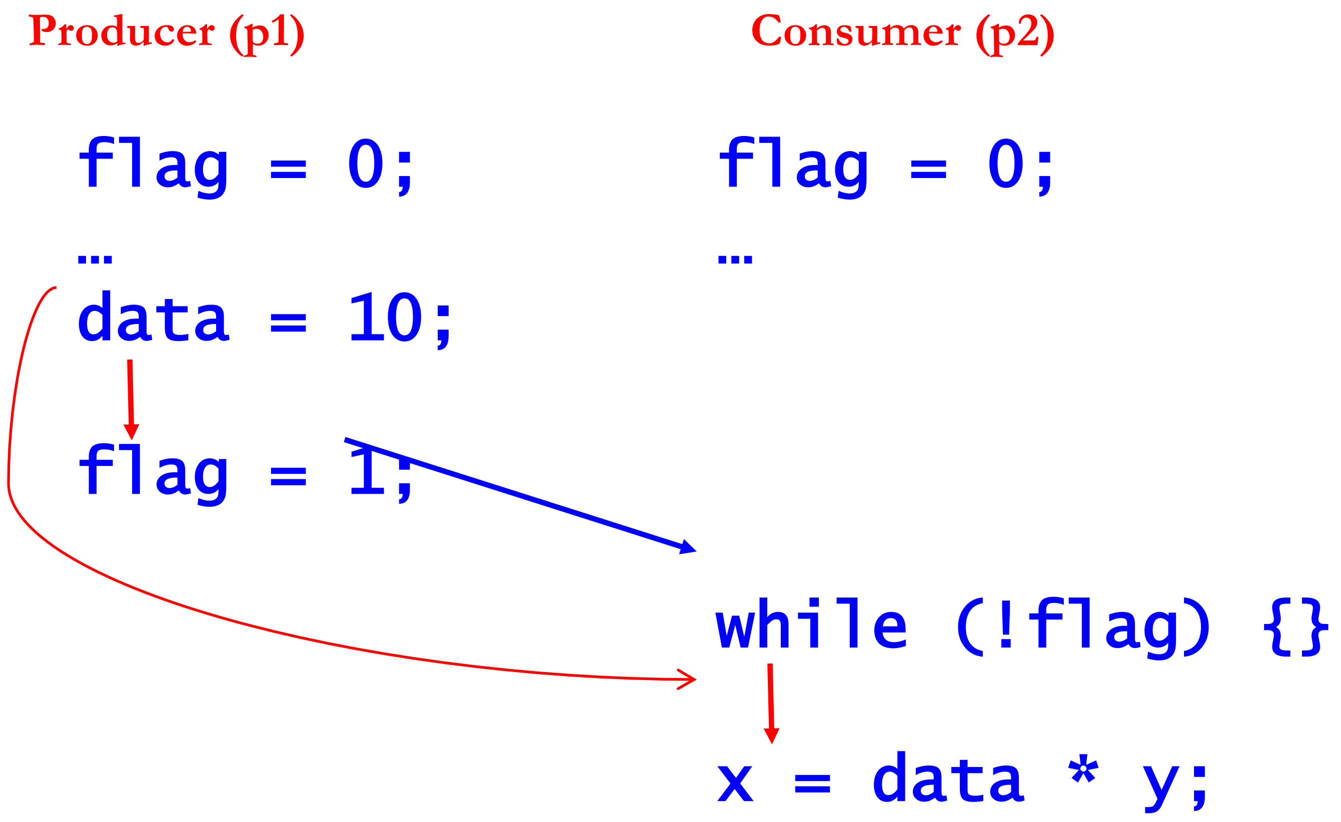
```
flag = 0;
…
data = 10;

flag = 1;
```

Consumer (p2)

```
flag = 0;
…



while (!flag) {}

x = data * y;
```

p2 should be able to see the latest value of `flag` & `data`

Memory Consistency



Producer (p1)

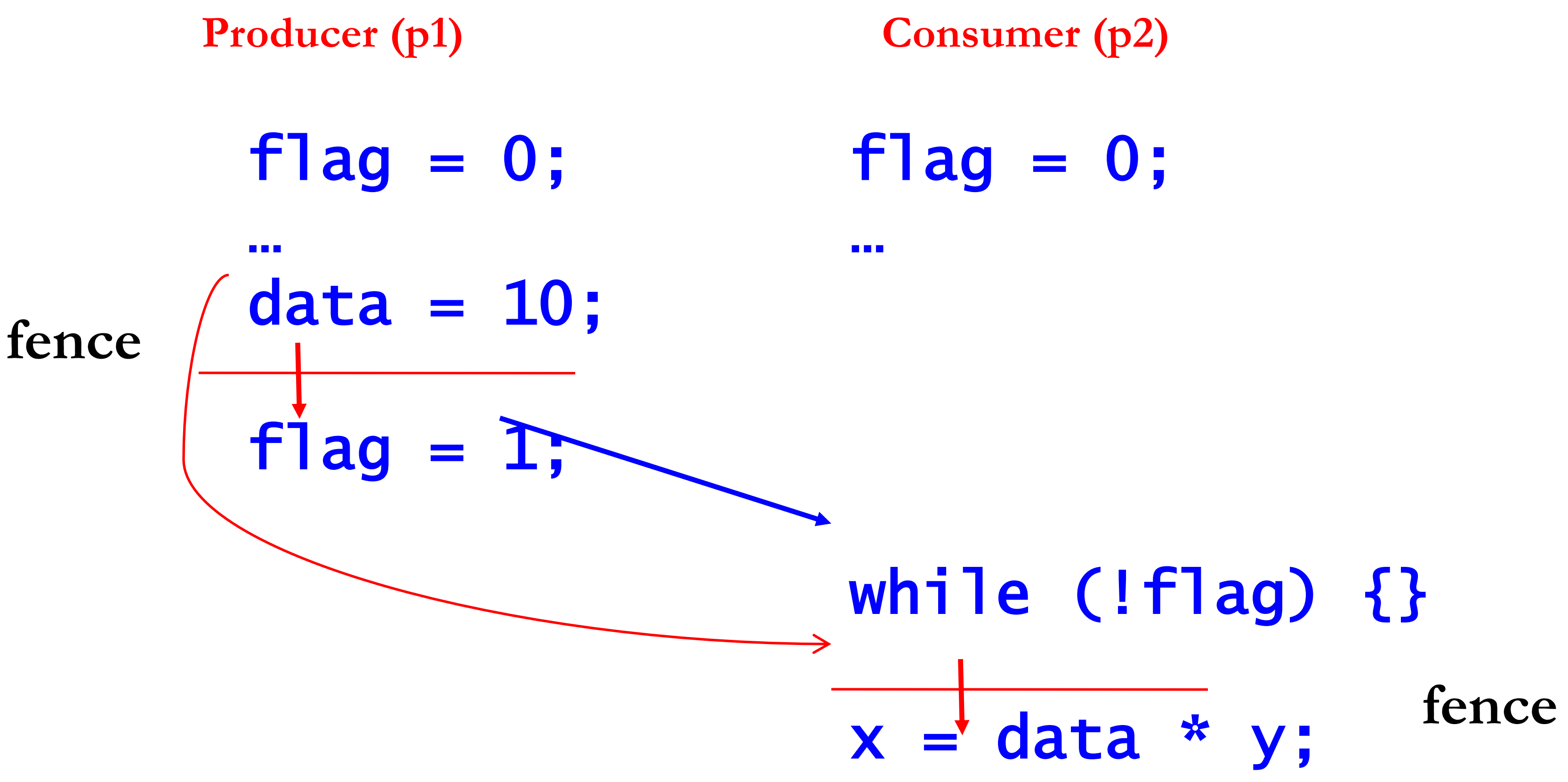
```
flag = 0;
…
data = 10;

flag = 1;
```

Consumer (p2)

```
flag = 0;
…


while (!flag) {}

x = data * y;
```

If p2 sees the update to `flag`, will p2 see the update to `data`?

Primitive Synchronization



Producer (p1)

```
flag = 0;
…
data = 10;
```

```
flag = 1;
```

Consumer (p2)

```
flag = 0;
…

while (!flag) {}
```

```
x = data * y;
```

The memory fence ensures that loads and stores are correctly ordered across threads

Parallel Architectures

- Types of parallelism
- Uniprocessor parallelism (advance concepts)
- Shared memory multiprocessors
  - Cache coherence and Consistency
  - Synchronization and transactional memory
- Hardware Multithreading
- Vector processors and GPUs
- Supercomputer and Datacentre architectures (if time permits)

The End!

- Student feedback questionnaires https://edin.ac/CEQ
  - We listen! Please provide feedback.
- Exam: May 1, 09:30 to 11:30
  - Similar in format and spirit to previous years