Introduction to Multiprocessors

- Why Multiprocessors?

Why Multiprocessors?

- Back to the early 2000s....

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

Why Multiprocessors?

- Back to the early 2000s....

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

- Instead:

Multiprocessors

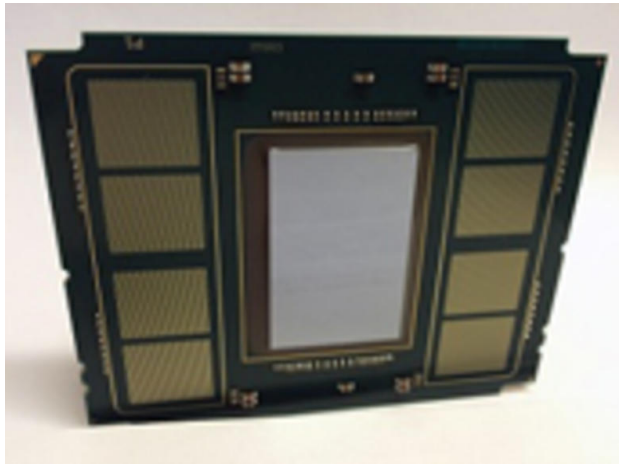
Why multiprocessors?

- 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

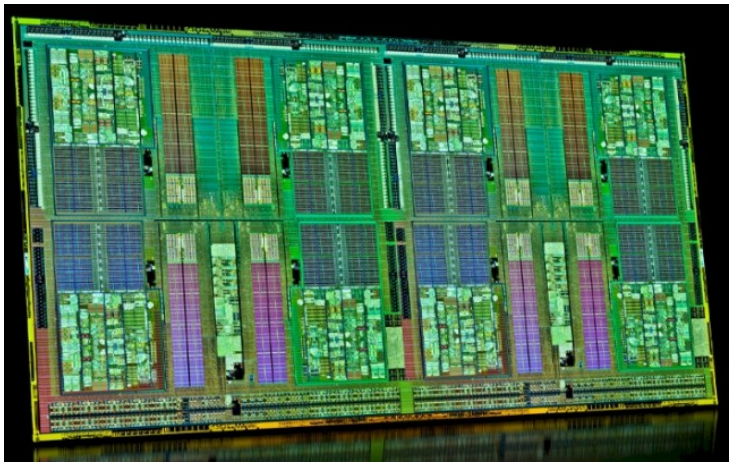
Chip multiprocessors (CMPs): the dividends of Moore's Law

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

Today's Chip Multiprocessors



Intel Xeon Phi upto 72 cores (knights landing)



AMD Opteron 6200: 16 cores

Exynos 7 (Samsung S6): 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.

Amdahl's Law and Efficiency

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

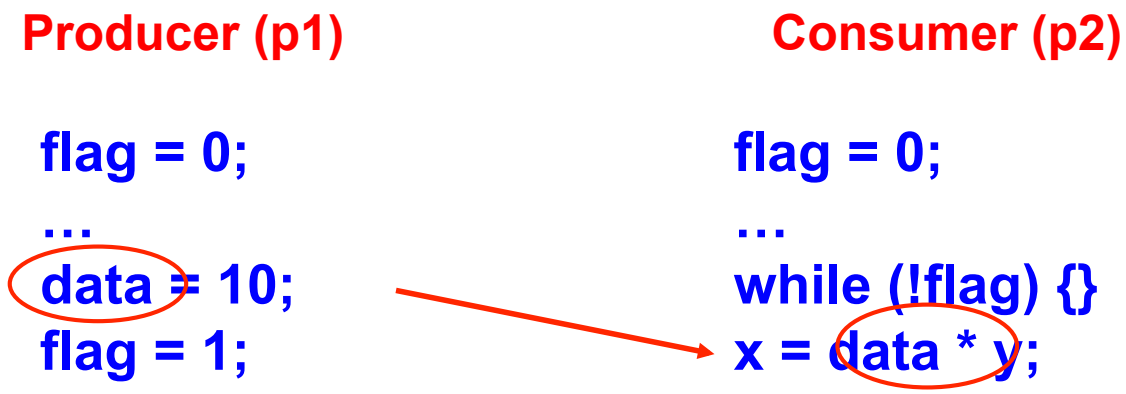
$$S_{overall} = \frac{1}{(1 - F) + \frac{F}{S_{par}}} \qquad \text{Efficiency} = \frac{S_{overall}}{P}$$

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

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

Inter-processor Communication Models

- Shared memory



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 vs Message Passing

- Shared memory pros
  - Easier to program
    - correctness first, performance later
  - For OS only (relatively) minor extensions required

- 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
    - Write propagation
    - Write serialization
- Memory Consistency
  - How are memory operations ordered?
  - What value should a read return?
  - When should writes be made visible to others?
- Primitive synchronization
  - Memory fences: memory ordering on demand
  - Atomic operations (e.g., Read-Modify-writes): support for locks (to protect critical sections)

Cache Coherence

Producer (p1)

```
flag = 0;
…
data = 10;

flag = 1;
```

Consumer (p2)

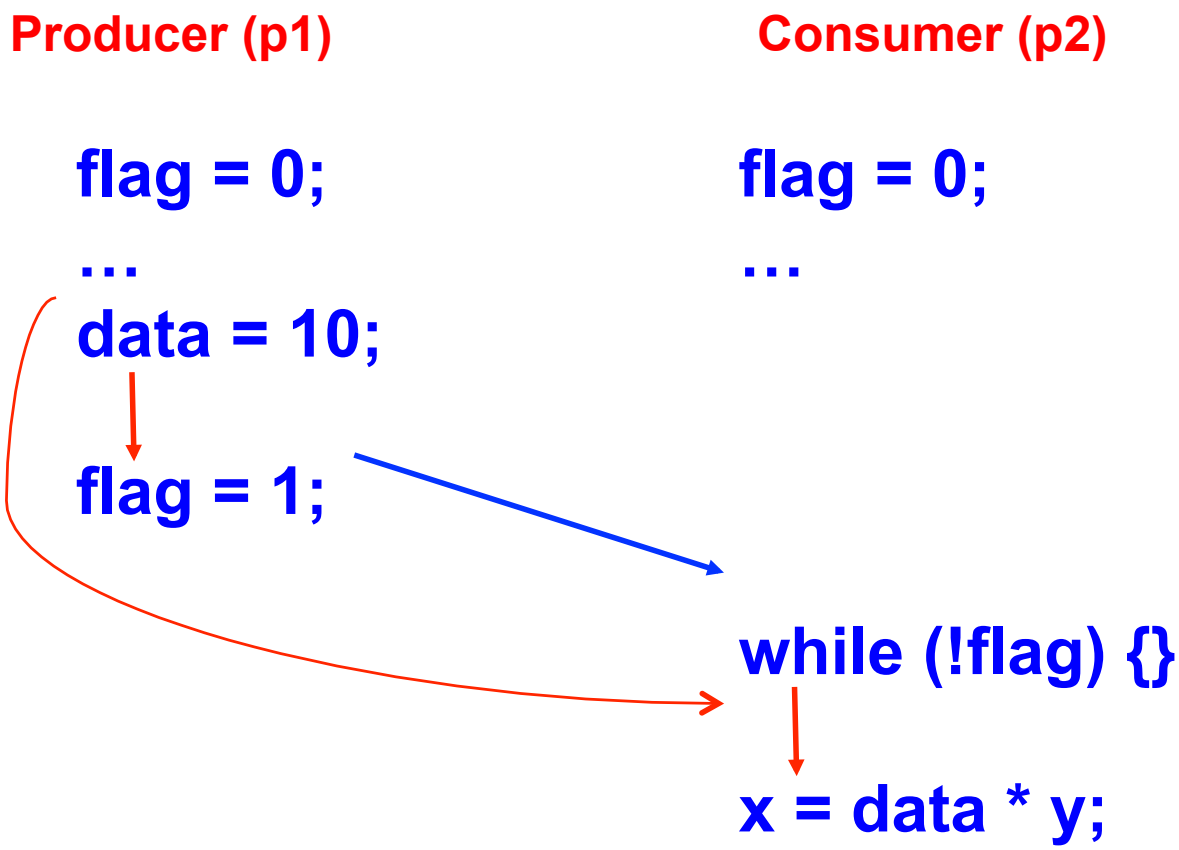
```
flag = 0;
…

while (!flag) {}

x = data * y;
```

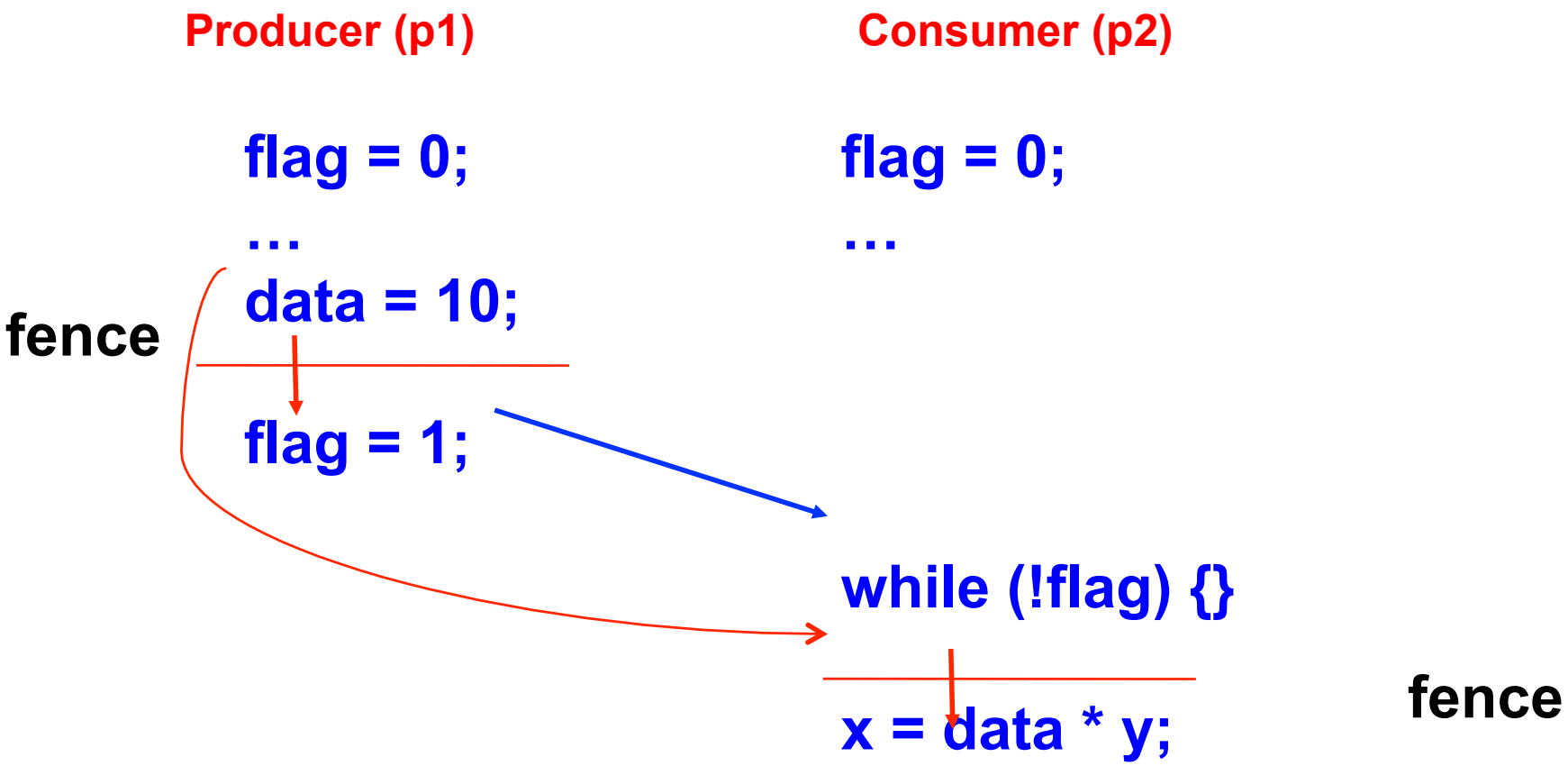
p2 should be able to see the latest value of `flag` & `data`

Memory Consistency



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

Primitive Synchronization



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!