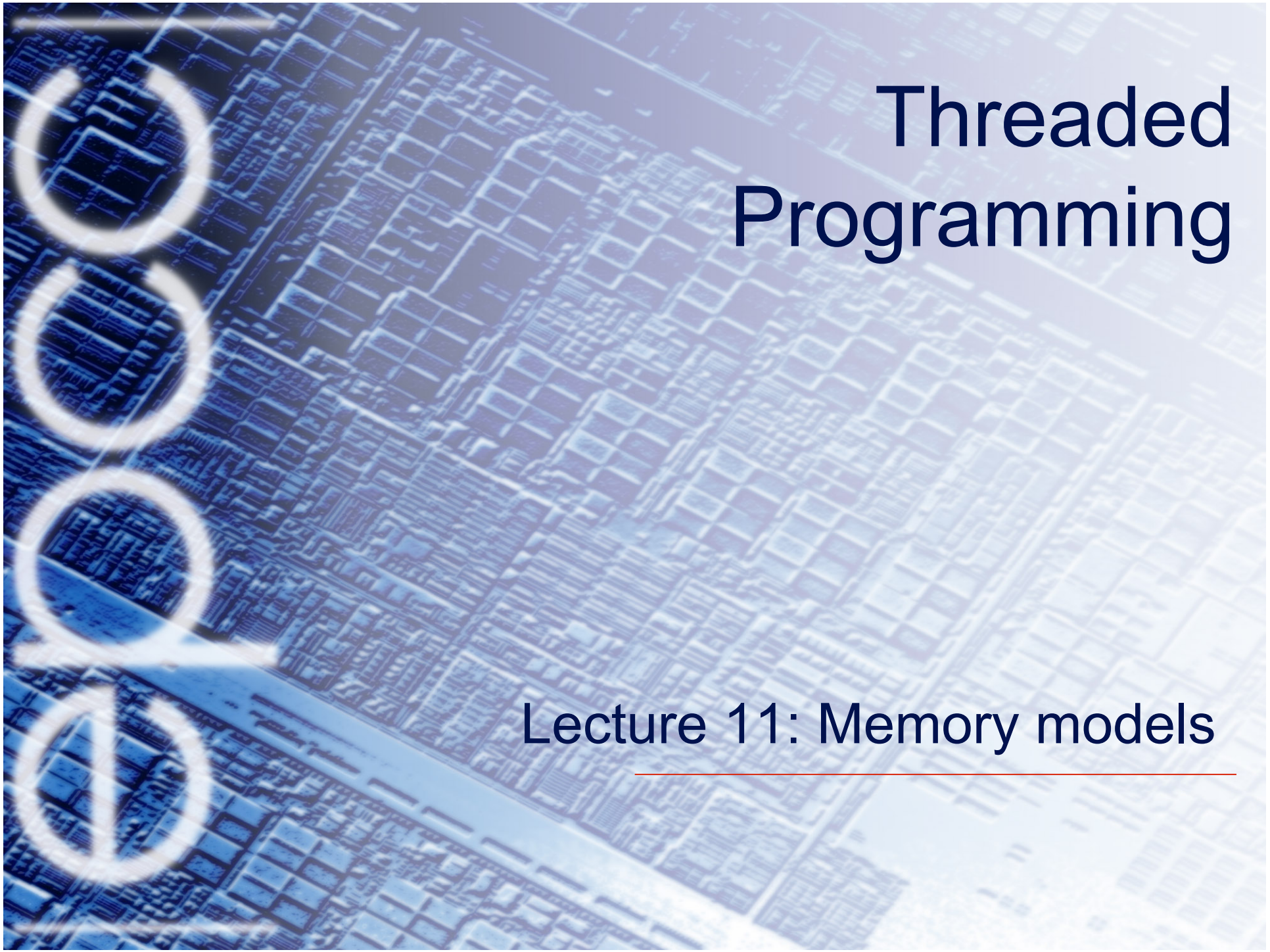
Threaded
Programming
Lecture 11: Memory models

Why do we need a memory model?

- On modern computers code is rarely executed in the same order as it was specified in the source code.
- Compilers, processors and memory systems reorder code to achieve maximum performance.
- Individual threads, when considered in isolation, exhibit *as-if-serial* semantics.
- Programmer's assumptions based on the memory model hold even in the face of code reordering performed by the compiler, the processors and the memory.

Example

- Reasoning about multithreaded execution is not that simple.

```
T1
x=1;
y=1;
```

```
T2
int r1=y;
int r2=x;
```

- If there is no reordering and *T2* sees value of *y* on read to be 1 then the following read of *x* should also return the value 1.
- If code in *T1* is reordered we can no longer make this assumption.

OpenMP Memory Model

- OpenMP supports a relaxed-consistency shared memory model.
  - Threads can maintain a temporary view of shared memory which is not consistent with that of other threads.
  - These temporary views are made consistent only at certain points in the program.
  - The operation which enforces consistency is called the flush operation

Flush operation

- Defines a sequence point at which a thread is guaranteed to see a consistent view of memory
  - All previous read/writes by this thread have completed and are visible to other threads
  - No subsequent read/writes by this thread have occurred
  - A flush operation is analogous to a fence in other shared memory API's

Flush and synchronization

- A flush operation is implied by OpenMP synchronizations, e.g.
  - at entry/exit of parallel regions
  - at implicit and explicit barriers
  - at entry/exit of critical regions
  - whenever a lock is set or unset

  ….

  (but not at entry to worksharing regions or entry/exit of master regions)

Example: producer-consumer pattern

Thread 0

```
a = foo();
flag = 1;
```

Thread 1

```
while (!flag);
b = a;
```

- This is incorrect code
- The compiler and/or hardware may re-order the reads/writes to a and flag, or flag may be held in a register.
- OpenMP has a flush directive which specifies an explicit flush operation
  - can be used to make the above example work
  - … but it's use is difficult and prone to subtle bugs

Java memory model

- Java also has a memory model
- It is similar to (but not the same as) that of OpenMP.
- It was the first such model for a popular programming language (1995).
  - Originally it was specified in a very different way
  - Some subtle flaws were discovered and it was revised in Java 5 (2004).

Synchronisation operations

- Java defines synchronisation operations on *monitors*
  - monitors are essentially locks
  - can be *acquired* and *released*
  - a **`synchronized`** block/method has an acquire at the start and a release at the end
  - a monitor release ensures that all previous reads and writes have completed
  - a monitor acquire ensures than no subsequent reads and writes have begun

```
public synchronized void add(int value){
  this.count += value;
}
```

monitor acquire

monitor release

Synchronisation and visibility

- One of the most important guarantees of the Java Memory Model is that an unlock on a monitor *happens-before* every subsequent lock on that monitor.

- If one action *happens-before* another, then the first is visible to and ordered before the second.

- Whatever memory operations are visible to *Thread 1* after it exits a synchronized block are also visible to *Thread 2* when it enters synchronized block protected by the same monitor.

The use of volatile

- It is guaranteed that a write to a *volatile* field *happens-before* every subsequent read of that field.

- Writing to a *volatile* field has the same memory effect as a monitor release, and reading from a volatile field has the same memory effect as a monitor acquire.

- Note: the volatile keyword in C/C++ and Fortran does not have multithreaded semantics and cannot be used in this way!

volatile example

```
class VolatileExample {
  int x = 0;
   volatile boolean v = false;
  public void writer() {
     x = 42;
     v = true;
   }
   public void reader() {
     if (v == true) {
       //uses x - guaranteed to see 42.
     }
   }
}
```

OpenMP vs Java

- In both APIs, the best way to ensure correct programs is to always use the built-in mechanisms to synchronise threads
  - the implied flushes or acquires/releases are guaranteed to avoid any problems
- In Java, it is *relatively* straightforward to use *volatile* to synchronise threads using user variables.
- In OpenMP, it is much more difficult
  - need to use flush and atomics
  - best avoided