Test and Verification Solutions

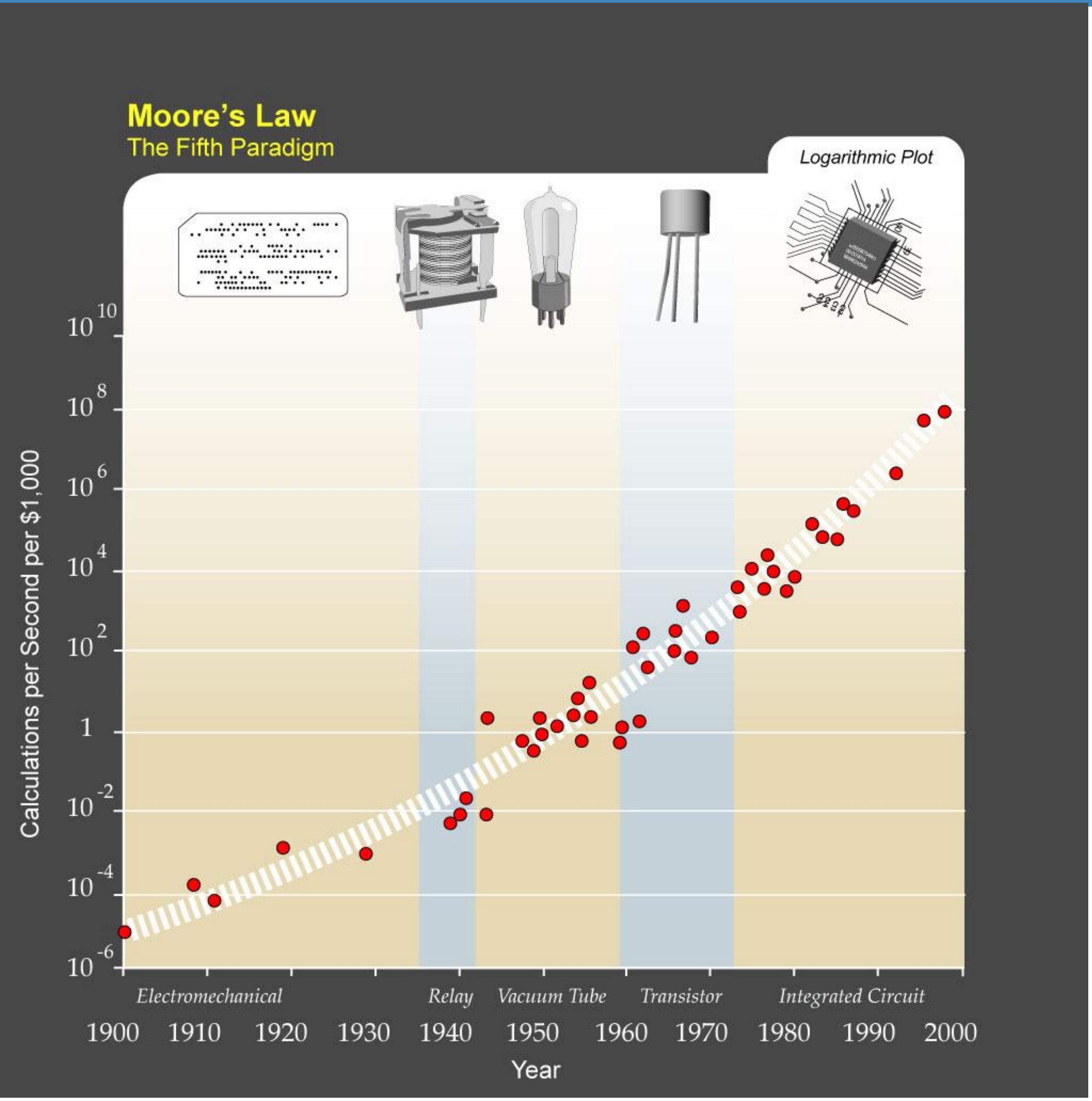
Innovations for Testing Parallel Software

Mike Bartley, TVS

Agenda

- **Have you noticed that the hardware in changing?**
  - The various types of distributed computing
  - The rise of multicore computing
- **Where is this stuff used?**
- **Is it important?**
- **The problems**
- **The challenges for**
  - Testing
  - Debug

The underlying law of hardware: Moore's Law



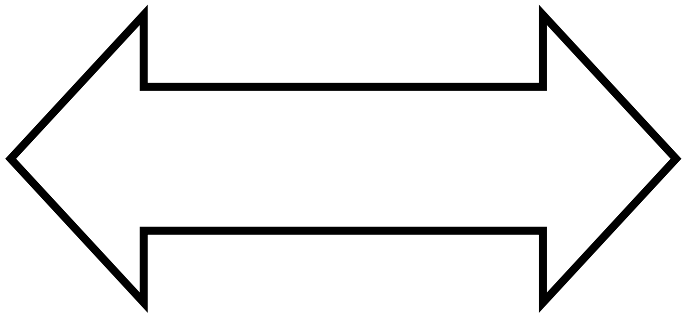
- **Hardware doubles the number of transistors & speed**
  - every 18 months
- **Power issues**
  - Frequency scaling through process minimisation increases power
- **How to keep increased performance with low power?**

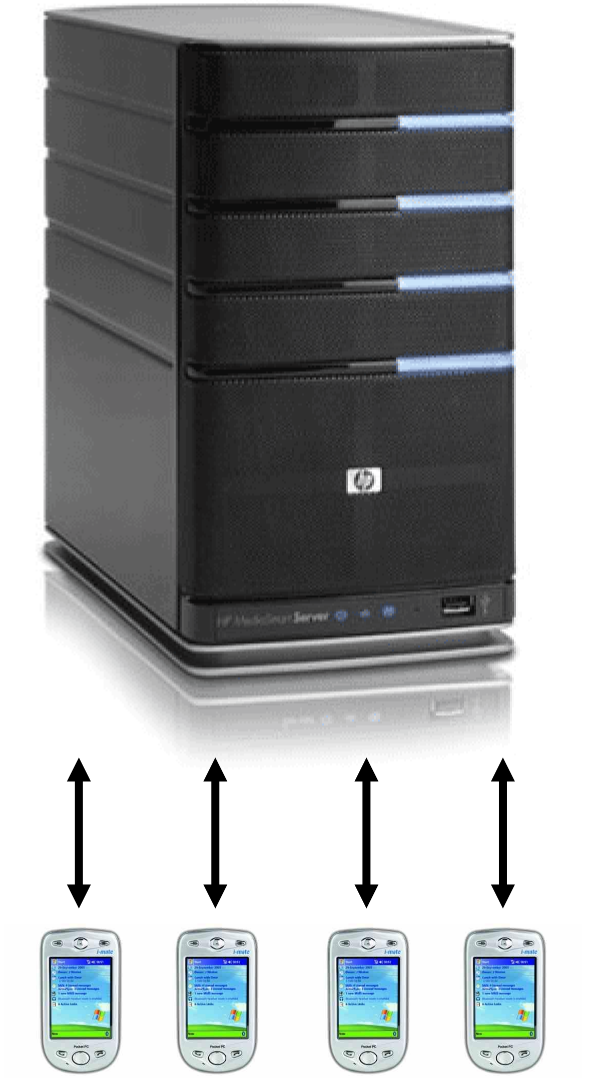
The hardware response is multiple cores!

The Intel® Core™ i7 processor delivers best-in-class performance for the most demanding applications. This quad-core processor features 8-way multitasking capability and additional L3 cache.

One of the goals of .NET Framework 4 was to make it easier for developers to write parallel programs that target multi-core machines. To achieve that goal, .NET 4 introduces various parallel-programming primitives that abstract away some of the messy details that developers have to deal with when implementing parallel programs from scratch.

Consider a mobile phone streaming video





Power considerations

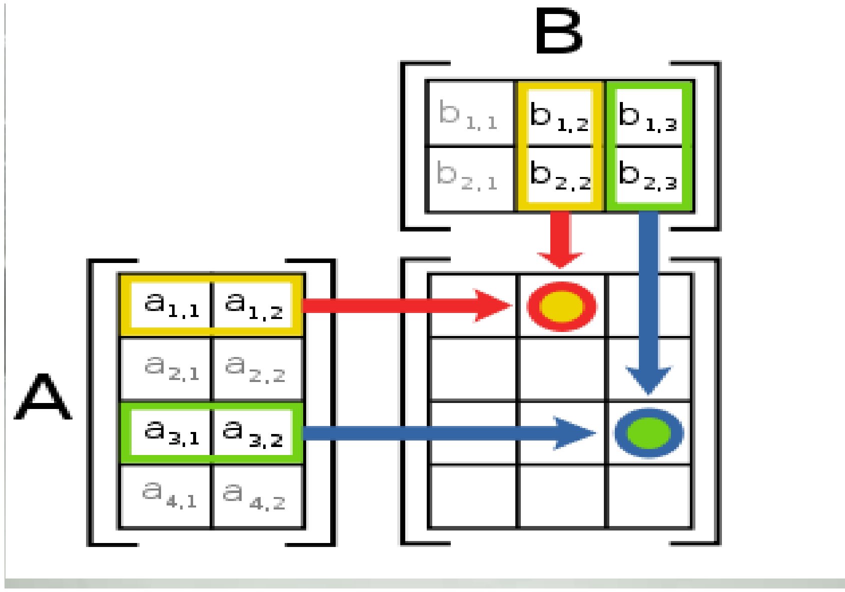
- **Mobile = battery**
- **Server = cost + heat**

Parallelism helps with

- **Mobile = Latency**
- **Server = Throughput**

Supercomputers

- **National Supercomputing Center in Tianjin**
  - OS = Linux
  - Main Memory = 229376 GB
  - Processor = Intel EM64T Xeon X56xx
    - 186368 of them delivering 4701000 GFlops
- **Supercomputers are used regularly in a number of applications**
  - Banking
  - Weather forecasting
  - Drug analysis
  - Modelling



The types of distributed computing

- **Distributed CPU and memory**
  - Client-server
  - Internet applications
- **Multi-processing**
  - ***Multiprocessing*** is the use of two or more central processing units (CPUs) within a single ***computer*** system
- **Multi-tasking**
  - The execution of multiple concurrent software processes in a system as opposed to a single process at any one instant
- **Multi-threading**
  - Threads have to share the resources of a single CPU
- **Multicore**
  - An integrated circuit which has two or more individual processors (called *cores* in this sense).

From sequential to parallel

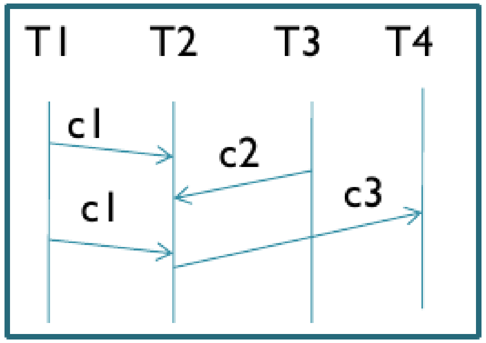
- Execution of a sequential program only depends on the starting state and its inputs.
- Execution of parallel programs also depends on the interactions between them.

Shared Memory:

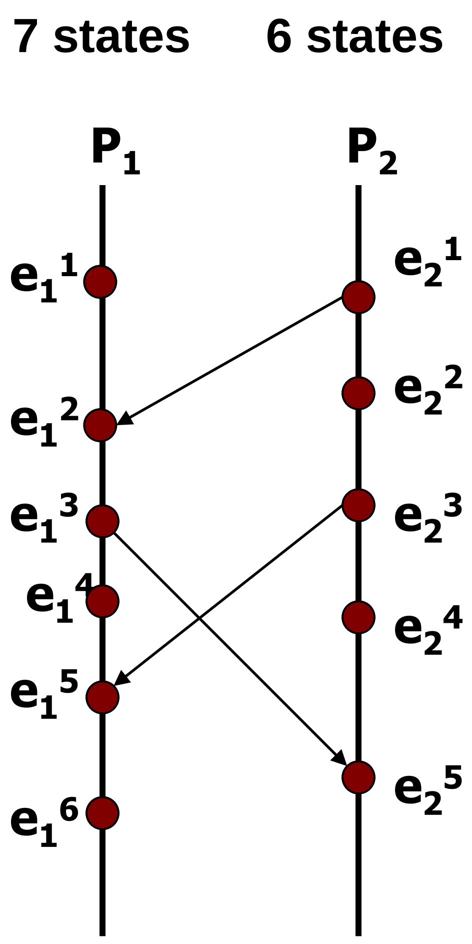
- Communication is based on altering the contents of shared memory locations (Java).
- Usually requires the application of some form of locking (e.g., mutexes, semaphores, or monitors) to coordinate between threads.

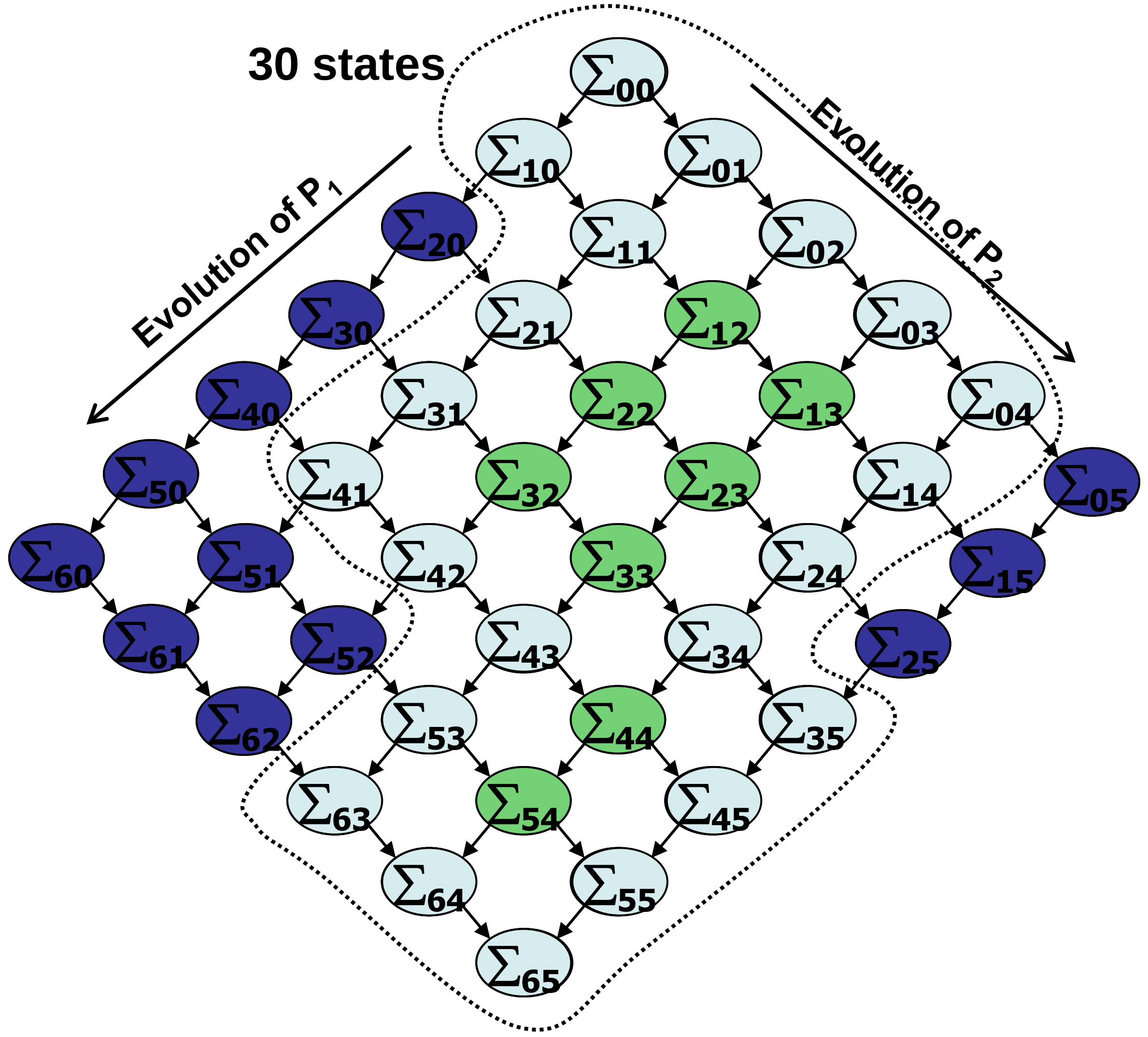
Message Passing:

- Communication is based on exchanging messages (occam, XC).



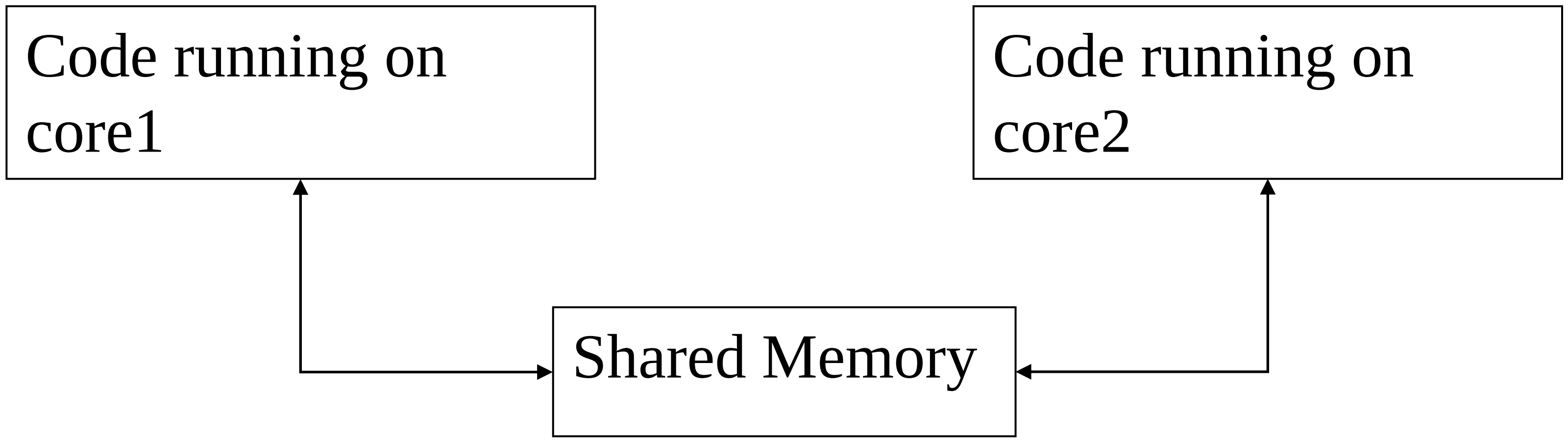
Many Potential Execution Orders



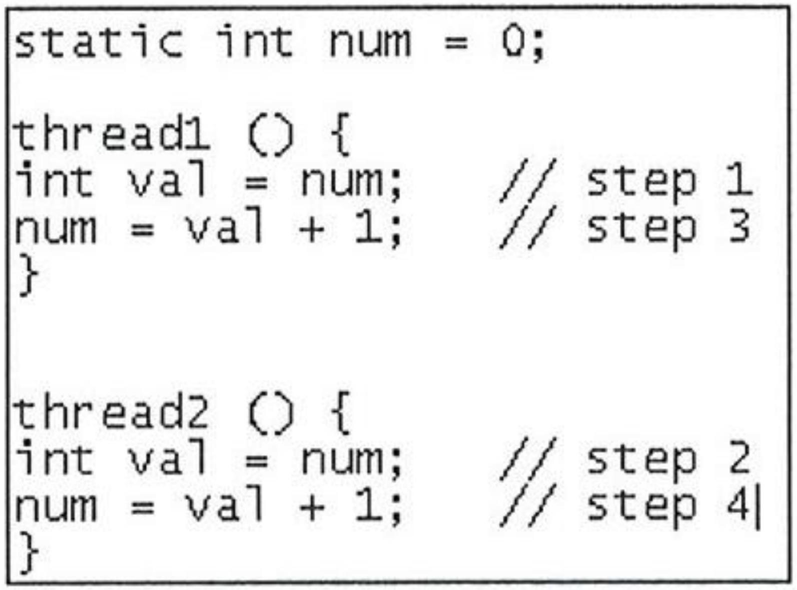


The problems with distributed processing
Non-determinism

- **Non-determinism**
  - We cannot guarantee the order of execution
  - And this can lead to race conditions



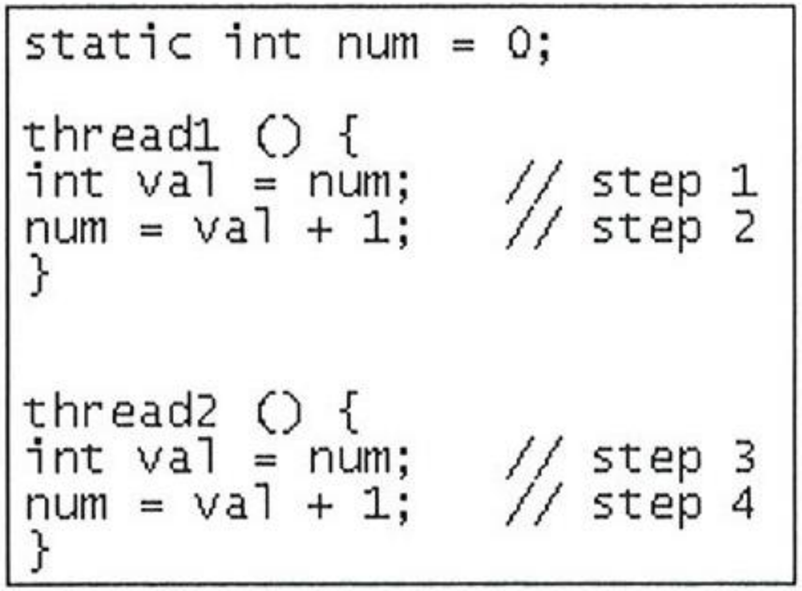
Race Condition Examples



```
static int num = 0;

thread1 () {
int val = num;        // step 1
num = val + 1;        // step 3
}


thread2 () {
int val = num;        // step 2
num = val + 1;        // step 4|
}
```



```
static int num = 0;

thread1 () {
int val = num;        // step 1
num = val + 1;        // step 2
}


thread2 () {
int val = num;        // step 3
num = val + 1;        // step 4
}
```

So why don’t we just use “num++”?

Another example

```
static void transfer(Transfer t) {
    balances[t.fundFrom] -= t.amount;
    balances[t.fundTo] += t.amount;
}
```

$\}$ Remember these are NOT atomic actions

- **Expected Behavior:**
  - **Money should pass from one account to another**
- **Observed Behavior:**
  - **Sometimes the amount taken is not equal to the amount received**
- **Possible bug:**
  - **Thread switch in the middle of money transfers**
  - **Second thread updates “Transfer” or “balances” in parallel**

- **So what are the solutions?**
  - Locks and mutual exclusion

The use of locks

```
class mutex{
    mutex() {//gettheappropriatemutex}
    ~mutex() {//release it }
    private: sometype mHandle;
}
voidfoo() {
    mutex get; //getthemutex
    …
    if(a) return; //released here
    …
    if(b) throw“oops”; //or here
    …
    return; //or here
```

Must remember to release the “mutex”!

But there are so many possible “exits” from the code

Locks can cause bugs – especially DEADLOCK!

Deadlock example

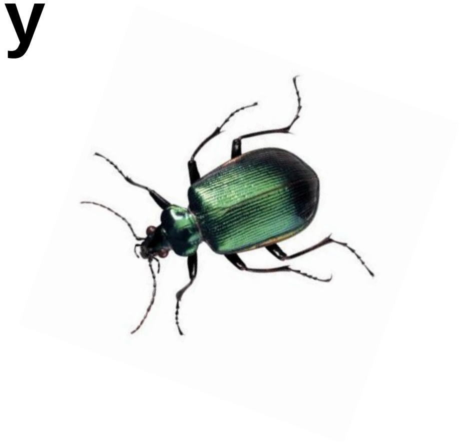
Code for Process P	**Code for Process Q**
Lock(M)	Lock(N)
Lock(N)	Lock(M)
Critical Section	*Critical Section*
Unlock(N)	Unlock(M)
Unlock(M)	Unlock(N)

Cause of deadlock

1. Tasks claim exclusive control of the resources they require ("mutual exclusion“ condition).
2. Tasks hold resources already allocated to them while waiting for additional resources ("wait for" condition).
3. Resources cannot be forcibly removed from the tasks holding them until the resources are used to completion ("no preemption" condition).
4. A circular chain of tasks exists, such that each task holds one or more resources that are being requested by the next task in the chain ("circular wait“ condition).

Commonly found bugs

- **An operation is assumed to be atomic but it is actually not.**
- **Wrong or no lock**
- **Message protocol errors**
  - Mismatch between channel ends
  - Missing send or receive
- **Orphaned threads due to abnormally terminating master thread**



Catching these bugs is challenging

Finding bugs in parallel SW

- **Research shows that bugs due to parallel code execution represent only ~10% of the bugs**
- **But they are the hardest to find**
  - If we run the same test twice it is not guaranteed to produce the same result (non-determinism)
    - Heisenbug
- **A disproportionate number are found late or by the customer**
  - Require large configurations to test
  - Typically appear only in specific configurations
  - These bugs are the most expensive!

- **We need to have some ways of disturbing the execution**
- **We need to know when we are done**

Heisenbugs

- ***A bug that disappears or alters its behavior when one attempts to probe or isolate it***
- **Why?**
  - Using a debugger alters the execution order and the bug disappears
  - Even adding a print statement changes behaviour!
- **For example**
  - Uninitialised variable in a sequential program
    - In C, 9 out of 10 Heisenbugs are from uninitialized auto variables
  - In parallel programs
    - Race conditions and deadlocks are both examples

- **How much time do you currently spend in debug?**

Disturbing the execution

Philosophy

- **Modify the program so it is more likely to exhibit bugs (without introducing new bugs – no false alarms)**
- **Minimize impact on the testing process (under-the-hood technology)**
- **Reuse existing tests**

Techniques to instrument concurrent events

- **Concurrent events are the events whose order determines the result of the program, such as accesses to shared variables, calls to synchronization primitives**
- **At every concurrent event, a random-based decision is made whether to cause a context switch – noise injection**
  - e.g., using a sleep statement

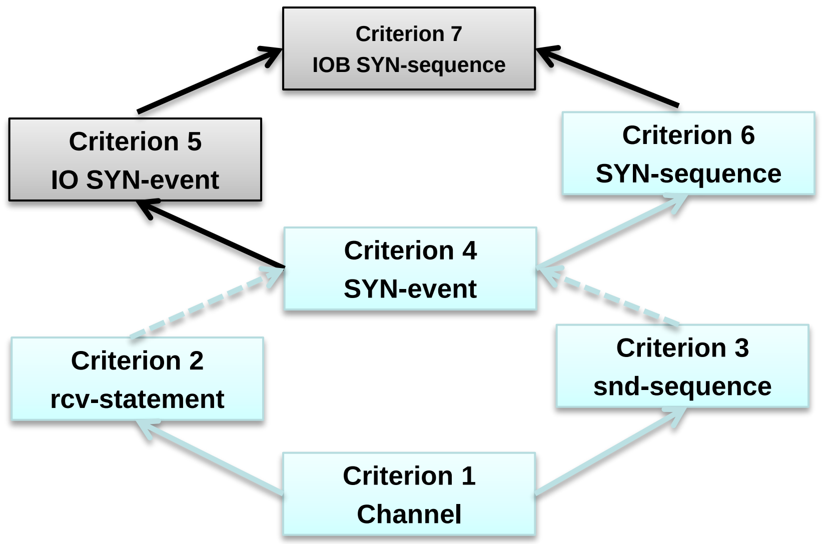
Knowing when we are done

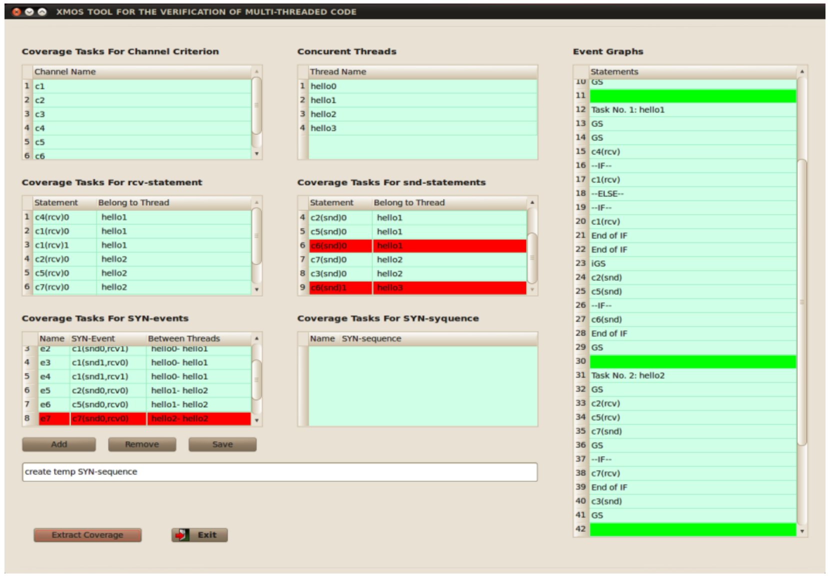
- **What are our current models of test completion?**
  - Black box?
    - Requirements coverage
    - Test matrix
    - Use-case coverage
    - All tests written and passing?
  - White box?
    - Code coverage at 90%? or some other random number
- **These will not be enough!**
  - Can we ensure no races?
  - Can we ensure no deadlocks?
- **Static analysis will become more important**

New coverage models

- **For shared memory:**
  - Synchronization coverage
    - Make sure that every synchronization primitive was fully exercised
  - Shared variable coverage
    - Make sure shared variables were accessed by more than one thread
  - ConTest from IBM implements these coverage models
- **For message passing:**
  - Communication coverage for multi-threaded message passing programs

New Coverage Model [Kyriakos Georgiou, K. Eder, TVS, XMOS]





Coverage Tasks For Channel Criterion

	Channel Name
1	c1
2	c2
3	c3
4	c4
5	c5
6	c6

Concurent Threads

	Thread Name
1	hello0
2	hello1
3	hello2
4	hello3

Coverage Tasks For rcv-statement

	Statement	Belong to Thread
1	c4(rcv)0	hello1
2	c1(rcv)0	hello1
3	c1(rcv)1	hello1
4	c2(rcv)0	hello2
5	c5(rcv)0	hello2
6	c7(rcv)0	hello2

Coverage Tasks For snd-statements

	Statement	Belong to Thread
4	c2(snd)0	hello1
5	c5(snd)0	hello1
6	🟥 c6(snd)0	🟥 hello1
7	c7(snd)0	hello2
8	c3(snd)0	hello2
9	🟥 c6(snd)1	🟥 hello3

Coverage Tasks For SYN-events

	Name	SYN-Event	Between Threads
3	e2	c1(snd0,rcv1)	hello0- hello1
4	e3	c1(snd1,rcv0)	hello0- hello1
5	e4	c1(snd1,rcv1)	hello0- hello1
6	e5	c2(snd0,rcv0)	hello1- hello2
7	e6	c5(snd0,rcv0)	hello1- hello2
8	🟥 e7	🟥 c7(snd0,rcv0)	🟥 hello2- hello2

Coverage Tasks For SYN-syquence

	Name	SYN-sequence

create temp SYN-sequence

Event Graphs

	Statements
10	GS
11	🟩
12	Task No. 1: hello1
13	GS
14	GS
15	c4(rcv)
16	--IF--
17	c1(rcv)
18	--ELSE--
19	--IF--
20	c1(rcv)
21	End of IF
22	End of IF
23	iGS
24	c2(snd)
25	c5(snd)
26	--IF--
27	c6(snd)
28	End of IF
29	GS
30	🟩
31	Task No. 2: hello2
32	GS
33	c2(rcv)
34	c5(rcv)
35	c7(snd)
36	GS
37	--IF--
38	c7(rcv)
39	End of IF
40	c3(snd)
41	GS
42	🟩

- **Captures communication infrastructure in a message-passing program**
  - Fully automatic extraction of coverage tasks up to SYN-events
  - User input required for SYN-sequences (tool support, semantics)
  - Complements existing code and functional coverage models
- Added value in practice
  - **Bugs can be captured before even running test cases on the code**
  - **SYN-sequences permit testing protocol compliance**

Conclusions

Testing of parallel SW has many challenges:

- Non-determinism
- Race conditions, deadlocks, livelocks
- Heisenbugs

– How to provoke the 10% of bugs down to parallelism?

- Making rare events happen more often

– How to know when you are done?

- New coverage models

Questions

Mike Bartley

Test and Verification Solutions

mike@testandverification.com