Have you noticed?
Even the Hardware is changing

Mike Bartley, TVS
• The rise of multicore computing
  – The types of distributed computing
• Where is this stuff used?
• Why is it important?
• The problems
• The challenges for
  – Testing
  – Debug
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
Example mobile phone chip
The underlying law of hardware: Moore’s Law

- Hardware doubles the number of transistors & speed
  - every 18 months
- But there are power issues
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.
Why is this important to the UK?

• Multicore devices are becoming all-pervasive, deployed in
  – embedded systems in washing machines and cars, mobile telephones, communications networks including the Internet
  – high-performance systems running applications in sectors including pharmaceutical, energy, aerospace, finance and engineering

• System inefficiencies that could arise from failing to exploit parallel computing effectively could have a substantial economic impact

• UK has world-class expertise in this area stemming in part from the development of the Transputer in the 1980s

• Much of this expertise is now dispersed
Isn’t this just distributed processing?

- **Distributed CPU and memory**
  - Client-server
  - Internet applications
- **Multitasking**
  - The execution of multiple concurrent software processes in a system as opposed to a single process at any one instant
- **Multi-processing**
  - *Multiprocessing* is the use of two or more central processing units (CPUs) within a single *computer* system
- **Multicore**
  - A *multi-core processor* is composed of two or more independent cores.
- **Multi-threading**
  - threads have to share the resources of a single CPU
The problems with distributed processing

Non-determinism

- Execution of a sequential program only depends on the starting state and its inputs
- Execution of distributed programs also depends on the interactions between processes

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

Diagram:

- Code running on core1
- Code running on core2
- Shared Memory
Race Condition Examples

So why don’t we just use “num++”? 
static void transfer(Transfer t) {
    balances[t.fundFrom] -= t.amount;
    balances[t.fundTo] += t.amount;
}

• **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

• **So what are the solutions?**
class mutex{
    mutex() { //get the appropriate mutex
        ~mutex() { //release it }
        private: sometype mHandle;
    }
    void foo() {
        mutex get; //get themutex
        ...
        if(a) return; //released here
        ...
        if(b) throw "oops"; //or here
        ...
        return; //or here
### Deadlock example

<table>
<thead>
<tr>
<th>Code for Process P</th>
<th>Code for Process Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock(M)</td>
<td>Lock(N)</td>
</tr>
<tr>
<td>Lock(N)</td>
<td>Lock(M)</td>
</tr>
<tr>
<td><strong>Critical Section</strong></td>
<td><strong>Critical Section</strong></td>
</tr>
<tr>
<td>Unlock(N)</td>
<td>Unlock(M)</td>
</tr>
<tr>
<td>Unlock(M)</td>
<td>Unlock(N)</td>
</tr>
</tbody>
</table>
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).
Other issues in testing parallel programs: debugging

- Research shows
  - 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)
    - We need to have some ways of disturbing the execution
  - So a disproportionate number are found late or by the customer
  - So they are the most expensive!
- Why is this the case?
  - Heisenbug!!!
  - Knowing 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 behaviour and the bug disappears
  - Even adding a print statement changes behaviour!

- For example
  - In a sequential program an uninitialised variable
    - 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?
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**
Other models of parallel execution?

- Processes pass messages rather than share data
  - But there is still non-determinism
  - And they can still deadlock
- Can consider coverage of process communications
Conclusions

- Parallel computing for multicore is driven by
  - The increasing complexity of the applications
  - A need to continue Moore’s law without the power cost
- The hardware suppliers and languages are responding
- This will make testing harder
  - Non-determinism
  - Deadlocks
  - Heisenbugs
  - How to provoke those 10% of bugs down to parallelism
  - How to know when you are done