The Problem with Threads

- Sequential execution is semantically function composition
  - deterministic components compose into deterministic results
- Checking serializability is much harder
  - must be checked for all possible interleavings
  - become exponentially worse with more threads
  - threads are effective if they do not share data directly
    - parallel make, web servers
  - "Threads are seriously flawed as a computation model"
    - "wildly nondeterministic"
  - Lee manifesto
    - Pruning nondeterminism is the wrong way to go
      - what remains is still intrinsically intractable
    - "Deterministic ends should be accomplished with deterministic means"
      - use nondeterminism only when needed

Problems with Locks [Scherer, Rochester 2005]

- Fault intolerance
  - a thread may die while holding a lock
- Preemption intolerance
  - a thread may be preempted while holding a lock
    - e.g. a page fault
- Deadlock
- Priority inversion
  - a high-priority thread H waits for a low-priority thread L
  - scheduling H instead L may lead to deadlock
- Convoying
  - threads tend to follow each other in lock steps after blocking together
- Non-composability
  - operations must be carefully composed to prevent deadlock

Current model considered harmful

- Threads and shared memory
  - Dominant model for parallel programming
- Difficult to program:
  - Hard to reason about all possible orderings
  - Subtle interactions of threads through shared memory
  - Easy to forget synchronization, introduce subtle bugs
  - Unintuitive model, implicit thread interactions and orderings

Step 1: Removing Races
Preliminaries

- Definitions
  - a data race
  - a racy program
- Road map today
  - (parallel) sharing -> no sharing -> sharing
- Its effect (the topics on the class schedule)
  - dependence
  - coherence
  - memory consistency
  - parallel programming models
  - software DSM
  - transactional memory
  - locking / nonblocking

Processes Instead of Threads

- Two-step strategy
  - copy-on-write when running parallel
  - sequential merge afterwards
  - fork
    
    ```
    x = 1
    y = x + 1
    ```
  - join(t1)
    
    ```
    x = 3
    ```
- Uses
  - speculative parallelization
    - BOP, PLDI'07, OOPSLA'11
    - Core*, MICRO'08, PLDI'10: SMTX, ASPLOS'09.
  - race-free and deterministic execution of threaded code

BOP, PLDI 2007

```plaintext
fork {
  x = 1
  y = x + 1
}
x = 3
join(t1)
z = x + y
```

PPR Hint [Ding+ PLDI'07]

- Likely rather than definite parallelism
  - bop ppr (code)
    - PPR: possibly parallel region/routine
    - an optimistic fork/spawn (w/o join)
- Sequential equivalence
  - same result as sequential execution
  - incorrect hints may hurt parallelism but not correctness
    - no non-determinism
    - no deadlock
    - no live lock
    - no lock (lock and wait free)
    - no parallel debugging
  - parallelized gzip, Lisp interpreter, Intel MKL, parser [PLDI'07]

Correctness

- Conflict detection
  - task i is checked after task i-1
  - no incorrect value prediction and no true dependences
  - correctness proof in PLDI'07 paper [similar to Allen & Kennedy, 2001]
- Error recovery through understudy
- User feedback
  - feedback on the cause of conflicts
  - non-trivial program changes may be needed
  - changing sequential code only
  - no parallel programming or debugging
BOP: Speculative Parallelization [Ding PLDI’07]

- Start from sequential code
- Divide it into a series of possibly parallel tasks
- Correctness checking at the end
- Error recovery by understudy

Correctness checking at understudy

speculation finishes first, aborts understudy

Copy-on-write at speculative tasks

Dell PowerEdge 6850 with 4 dual-core Intel 3.4GHz, Xeon 7140M processors, GCC 4.0.1 with “-O3”

Gzip compressing an 84MB file

<table>
<thead>
<tr>
<th>version</th>
<th>speculation depth</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>sequential</td>
</tr>
<tr>
<td>times (sec)</td>
<td>8.46, 8.56,</td>
</tr>
<tr>
<td></td>
<td>8.50, 8.51,</td>
</tr>
<tr>
<td></td>
<td>8.53, 8.48,</td>
</tr>
<tr>
<td>avg time</td>
<td>8.51</td>
</tr>
<tr>
<td>avg speedup</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Intel MKL (Solving 8 Linear Systems)

both processes and threads scale, but former are faster

Copy-and-Merge Parallelization

Reduction over list concatenation

Parallel execution: Reduction over monoids is deterministic.

Reducers

- Naïve Cilk++:
  
  ```
  cilk_for (i = 0; i < n; ++i)
  s += x[i];
  ```

- Correct Cilk: Use "reducer" object
  
  ```
  cilk::reducer_opadd<float> r;
  cilk_for (i = 0; i < n; ++i)
  r += x[i];
  s += r.get_value();
  ```

Source: M. Frigo, CSoADS 2009 talk
Additional reducers

- Provided
- Lists (append, prepend)
- min, max, [min, max]_index
- opadd
- ostream
- basic_string
- May build your own

Efficient System-Enforced Deterministic Parallelism

Amitai Aviram, Shu-Chun Weng, Sen Hu, Bryan Ford

Decentralized/Distributed Systems Group, Yale University
http://dedis.cs.yale.edu/

9th OSDI, Vancouver – October 5, 2010

Concurrent Revisions:
A deterministic concurrency model.

Daan Leijen, Alexandro Baldassin, and Sebastian Burckhardt
Microsoft Research
(OOPSLA 2010)

Application = Shared Data and Tasks

Example: Office application
- Save the document
- React to keyboard input by the user
- Perform a spellcheck in the background
- Exchange updates with remote users

Our Proposed Programming Model:
Revisions and Isolation Types

Revision
A logical unit of work that is forked and joined

Isolation Type
A type which implements automatic copying/merging of versions on write-write conflict

- Deterministic Conflict Resolution, never roll-back
- No restrictions on tasks (can be long-running, do I/O)
- Full concurrent reading and writing of shared data
- Clean semantics (see technical report)
- Fast and space-efficient runtime implementation
“Problem Example 1” is solved

Render task reads position of all game objects

Physics task updates position of all game objects

No interference!

---

```csharp
public void Run()
{
    // Fork a revision: conceptually all versioned state is copied and each revision is fully isolated
    var r = CurrentRevision.Fork(() => {
        s = "world"; // write to s in the forked revision
    });
    s = "Hello"; // and write to s in the main revision
    Console.WriteLine(s);  // revision are isolated so it always shows 'hello'
    CurrentRevision.Join(r);  // writes are merged back on the join
    Console.WriteLine(s);  // by default writes in the child
}

class Program
{
    static void Main(string[] args)
    {
        new Sample().Run();
    }
}
```

---

SpiceC: Computation Model

- Memory Spaces
  - Shared & Private
- Copy & Commit
  - Support speculative parallelism
  - Uniform program for systems with and without cache coherence

### Features

- Programmability – easy-to-use compiler directives
- Versatility – support multiple forms of parallelism and speculative parallelism
- Performance portability – does not rely on cache coherence

Implementation on both multicore and manycore systems

2x – 18x speedup on a 24-core machine

---

Conclusion

- SpiceC programming model
  - Copy/commit computation model
  - Programming interfaces
- Features
  - Programmability – easy-to-use compiler directives
  - Versatility – support multiple forms of parallelism and speculative parallelism
  - Performance portability – does not rely on cache coherence
- Implementation on both multicore and manycore systems
  - 2x – 18x speedup on a 24-core machine

---

What About True Dependences?

[Ke et al. OOSPLA 2011]
How to Tackle True Dependencies?

- Infrequent true dependences
  - speculation
  - both data and control
- Circumventable true dependences
  - value prediction
- Otherwise
  - serialize
  - speculative synchronization

```plaintext
while (has_more()) {
  n = next_item()
  if (search(n) != nil)
    return n
}
```

True Dependences, Truly Dependent

Channel-based Dependence Hints

- Channel fill/post/wait
  - bop_fill(addr, channel id): add an address range to a channel
  - bop_post( channel id ): copy/send channel data
    - modified data only, single post per channel
  - bop_wait( channel id ): stall until a post, copy incoming data
  - bop_cc( channel 1, channel 2 ): equate two channels
- Properties
  - sender-side addressing
  - no sender/receiver disagreement
  - can pass dynamic data
  - message aggregation
  - channel chaining
  - to express conditional dependence
  - partial dependence marking

```
Listing 3: A possibly partially parallel loop

while (has_more(inputs)) begin
  w = get_next(inputs)
  # try computing w in parallel
  bop_ppr {
    t = compute(w)
    # allocate a new node
    n = new_qnode(t)
    # make n the new tail
    append(outputs, n)
  }
end
```

```
Listing 4: Safe parallelization using basic primitives

cid = 0  # channel id
while (has_more(inputs)) begin
  w = get_next(inputs)
  bop_ppr {
    t = compute(w)
    n = new_qnode(t)
    # wait for the last tail
    bop_wait( cid - 1 ) if cid > 0
    append(outputs, n)
    # send the new tail
    bop_fill( cid, n, sizeof(qnode) )
    bop_post( cid )
  }
  cid ++
end
```

```
Task 1  Task 2  Task 3

<table>
<thead>
<tr>
<th>dynamic data</th>
<th>ordered execution</th>
<th>conditional dependence</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = new counter(0)</td>
<td>bop_wait(1)</td>
<td>bop_wait(1)</td>
</tr>
<tr>
<td>bop_fill(1, a)</td>
<td>bop_post(1)</td>
<td>a++</td>
</tr>
<tr>
<td>bop_post(1)</td>
<td>bop_wait(1)</td>
<td>a++</td>
</tr>
<tr>
<td>bop_fill(1, a)</td>
<td>bop_wait(1)</td>
<td>a++</td>
</tr>
<tr>
<td>bop_post(1)</td>
<td>bop_wait(1)</td>
<td>a++</td>
</tr>
<tr>
<td>bop_fill(1, a)</td>
<td>bop_wait(1)</td>
<td>a++</td>
</tr>
<tr>
<td>bop_post(1)</td>
<td>bop_wait(1)</td>
<td>a++</td>
</tr>
</tbody>
</table>
```
• Hammer from SPEC 2006
  - genetic search
  - 36K SLOC
  - most time spent in calibration and search
  - most time in calibration spent in function P7Viterbi called in the loop in the right

```c
for (i = 0; i < parallelism; i++) {
    bop_ppr(); // begin possibly parallel region (PPR)

    if (PP7ViterbiSize(sq, id)) {
        score = P7Viterbi(sq, id, sq, sq); // parallelism
        bop_wait(); // sync
    }
    // do more

    bop_wait(); // sync
    bop_ordered(); // implemented by post-wait
}
```

• String substitution
  - e.g. "aba" -> "bab"
  - maybe parallel
  - e.g. "a ... a" -> "aa ... a"
  - maybe sequential
  - "abas ... a" -> "bb ... bab"

```c
for ii in 2 ... n with step b do
    # try next block in parallel
    bop_ppr {
        # update block boundary

        for i = ii ... min(i+b-1, n) do
            if matches(str[i-2 ... i], src) {
                str[i-2 ... i] = target
                # 3 letters are changed
            }
        end
    end

    # update the whole string
    return str[0 ... n]
```

• Iterative solvers
  - fix-point solutions
  - data clustering
  - dataflow analysis
  - scientific simulation
  - Convergence check
    - forcing time steps to serialize
  - Time skewing
    - overlap time steps
    - okay until the last step

```c
for i in 1...n
    # try inner!
    bop_ppr {
        r = compute
        bop_ordered {
            s = s.add_result(r)
            # try next time step in parallel
        } if good_enough?( s )
        bop_ordered {
            converged = true
        }
    end while not converged
```

• Parallel Performance of String Substitution

```c
Listing 8: string scanning and pattern conversion
str[0...n], src = "aba", target = "bab"
```

• Parallelism hint [Ding, PLDI 2007]
  - process-based design, understudy for recovery
  - Dependence hint implementation
    - correctness and progress guarantee
    - filtered posting, sender/receiver conflict checking, last-writer checking, silent drop, reordered receive, hint overrides
  - High-level constructs
    - OpenMP ordered, Thies et al.'s pipeline
  - System design
    - process reuse and continuous speculation [ICT branch]
    - byte-granularity checking
    - Examples and evaluation

(a) 3-stage TCA pipeline
(b) implementation by dependence hints
**Review**

- **BOP system design**
  - What it mean by safe parallelization?
  - What programming primitives does it provide?
  - Why can’t it include concurrency constructs such as atomic?
  - How to create parallel tasks?
  - How to shared data?
  - How to synchronized shared data access?
  - How to recover from error?

- **Parallel programming**
  - Can you parallel while-loops, e.g. string find/substitution?
  - Can you implement time skewing?