CSC 258/458 Guest Lecture

Safe Parallel Programming

Chen Ding

Professor

February 19, 2015

Innovative Technology for Computing Professionals
Computer

May 2006

COVER FEATURE

The Problem with Threads

Edward A. Lee
University of California, Berkeley

For concurrent programming to become mainstream, we must discard threads as a programming model. Nondeterminism should be judiciously and carefully introduced where needed, and it should be explicit in programs.

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

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

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

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

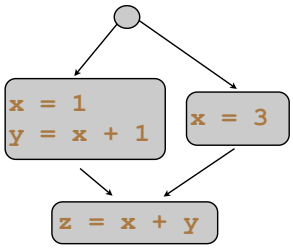
Processes Instead of Threads

- Two-step strategy
  - copy-on-write when running parallel
  - sequential merge afterwards

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

- Uses
  - speculative parallelization
    - BOP, PLDI'07, OOPSLA'11
    - CorD*, MICRO'08, PLDI'10; SMTX, ASPLOS'09.
  - race-free and deterministic execution of threaded code
    - Grace, OOPSLA'09; Isolation/revision type, OOPSLA'10; CoreDet, ASPLOS'10; Determinator, OSDI'10; DoublePlay/uniparallelism, ASPLOS'11/13, SOSP'11; Spice C, PPOPP'11, PLDI'12.


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



BOP, PLDI 2007

```
p1 = fork {
  xt1 = 1
  yt1 = xt1 + 1
}
xt2 = 3
join(p1)
xt3 = ω(xt1, xt2)=3
z = xt3 + yt1
```

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]

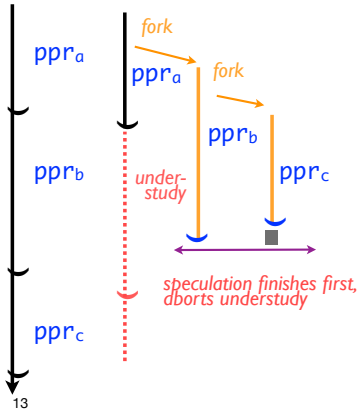
Correctness Checking and Error Recovery

Correctness

- Conflict detection
  - task i is checked after task i-1
  - no incorrect value prediction and no true dependences (otherwise recover)
  - 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
- Copy-on-write at speculative tasks
- Correctness checking at the end
- Error recovery by understudy

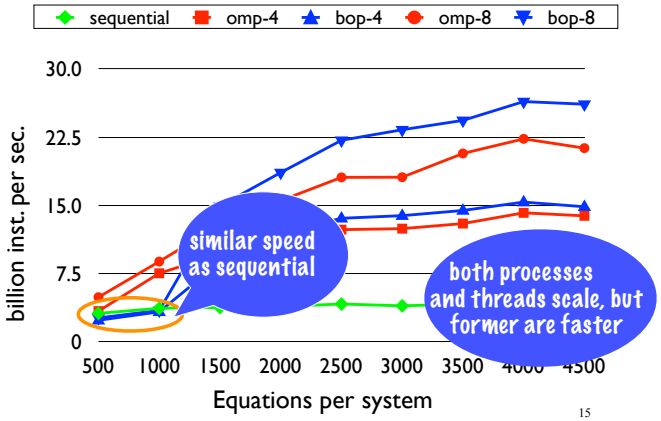


Gzip compressing an 84MB file

version	sequential	speculation depth 1	3	7
times (sec)	8.46, 8.56, 8.50, 8.51 8.53, 8.48			
avg time	8.51			
avg speedup	1.00			

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

Intel MKL (Solving 8 Linear Systems)



Copy-and-Merge Parallelization

Reducers

- Naïve Cilk++:

```
cilk_for (i = 0; i < n; ++i)
  s += x[i];
```

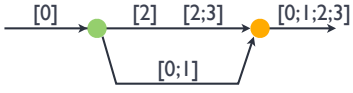
- Correct Cilk: Use "reducer" object

```
cilk::reducer_opadd<float> r;
cilk_for (i = 0; i < n; ++i)
  r += x[i];
s += r.get_value ();
```


Reduction over list concatenation

```
x.append (0);
cilk_spawn x.append (1);
x.append (2);
x.append (3);
cilk_sync;
```

Parallel execution: Reduction over monoids is deterministic.



Source: M. Frigo, CScADS 2009 talk

Additional reducers

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

GRACE

It is now safe to turn on your multicores.

OOPSLA 2009

MSR talk video: http://research.microsoft.com/apps/video/default.aspx?id=115873

- Grace: Safe Multi-threaded Programming for C/C++
- Fork-join parallelism
  - Cilk, TBB, OpenMP, Map-reduce do not prevent errors
- Solution:
  - sequential semantics---execute threads in program order

Efficient System-Enforced Deterministic Parallelism

Amittai 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:

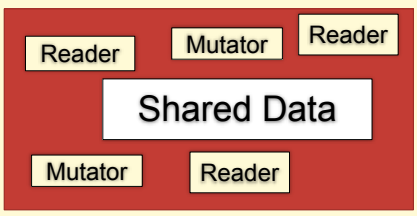
A deterministic concurrency model.

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

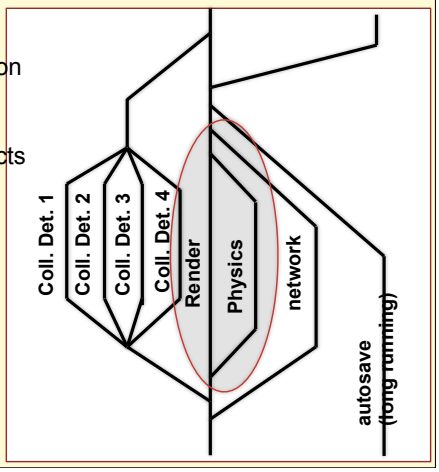
- Deterministic Conflict Resolution, never roll-back
- No restrictions on tasks (can be long-running, do I/O)
- Full concurrent reading and writing of shared data
- Clean semantics (see technical report)
- Fast and space-efficient runtime implementation

"Problem Example 1" is solved

Render task reads position of all game objects

Physics task updates position of all game objects

No interference!




```
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();
 }
}
```

- http://www.rise4fun.com/revisions

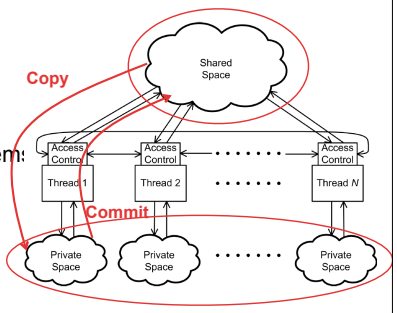
PPOPP 2011

SpiceC: *S*calable *p*arallelism via *i*mplicit *c*opying and explicit *C*ommit

Min Feng, Rajiv Gupta, and Yi Hu
University of California, Riverside

SpiceC: Computation Model

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



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 Dependences?

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

```
while (has_more()) {
  n = next_item()

  if (search(n) != nil)
    return n
}
```


True Dependences, Truly Dependent

- Are there true dependences among PPR tasks?
- How to synchronize?
- How to communicate?

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
    n = new_qnode(t)
    # make n the new tail
    append(outputs, n)
  }
end
```

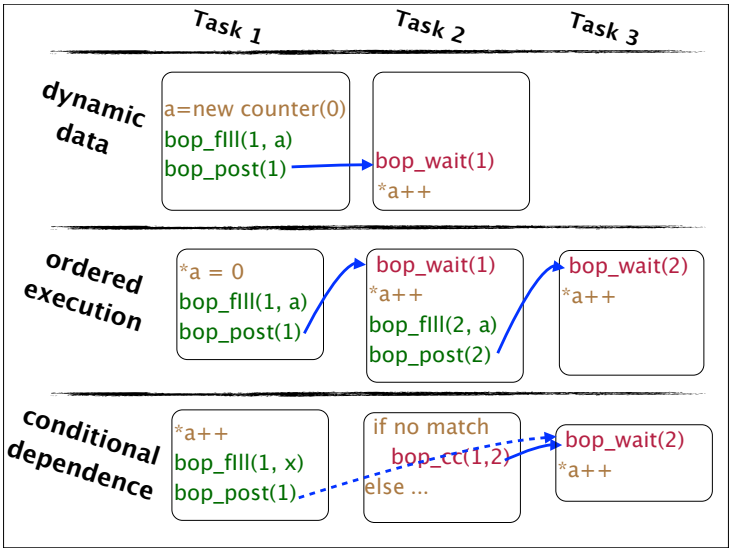
Channel-based Dependence Hints

- 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

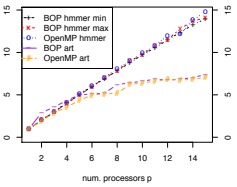
- Sender-side addressing
  - what to communicate
- Sequential access to shared data
- Safety
  - unmatched post/wait
  - wrong channel

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
```



- Hmmer 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




```
for (i = 0; i < parallelism; i++) {
  bop_ppr {  // begin possibly parallel region (PPR)
    mx = CreatePlan7Matrix(1, hmm->M, 25, 0);
    for (idx=i*temp; idx<(i+1)*temp && idx<nsample; idx++) {
      dsq = DigitizeSequence(seq[idx], sqlen[idx]);
      if (P7ViterbiSize(sqlen[idx], hmm->M) <= RAMLIMIT)
        score = P7Viterbi(dsq, sqlen[idx], hmm, mx, NULL);
      else
        score = P7SmallViterbi(dsq, sqlen[idx], hmm, mx, NULL);
      hhu[idx%temp] = score;
      free(dsq); free(seq[idx]);
    }
    FreePlan7Matrix(mx);
  bop_ordered {  // implemented by post-wait
    for (idx=i*temp; idx<(i+1)*temp && idx<nsample; idx++) {
      length_zc = AddToHistogram_zc(&(post_zc.a), hhu[idx%temp]);
      if (hhu[idx%temp] > post_zc.b) post_zc.b = hhu[idx%temp];
    }
  } // end bop_ordered
} // end bop_ppr
}
```

- 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

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

rollback PPRs after convergence

make the check non-blocking

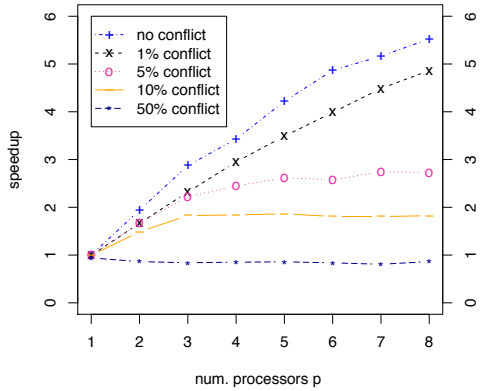
wait for inner loop to finish

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

Listing 8: string scanning and pattern conversion

```
str[0...n], src=''aba'', target=''bab''

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

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

     end
    end
  }
end
# update the whole string

return str[0 ... n]
```

Parallel Performance of String Substitution




```
for i in 1 ... n
  begin_pipelined_loop
  // serial stage 1
  pipeline( p )
  // parallel stage 2
  pipeline
  // serial stage 3
  end_pipelined_loop
end for
```

(a) 3-stage TCA pipeline

```
for i in 1 ... n
  bop_ppr {
    bop_wait( <my_ppr-1, s1> )
    // serial stage 1
    bop_post( <my_ppr, s1> )
    bop_wait( <my_ppr-1, s1> )
    // parallel stage 2
    bop_post( <my_ppr, s2> )
    bop_wait( <my_ppr-1, s3> )
    // serial stage 3
    bop_post( <my_ppr, s3> )
  }
end for
```

(b) implementation by dependence hints

Details [OOPSLA 2011]

- 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

BOP Demo

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?

Expression and Implementation of Parallelism