Commuting Operations and Deterministic Execution in Parallel Programs

Martin Rinard
MIT CSAIL
Massachusetts Institute of Technology
Cambridge, MA 02139
Purpose of Talk

• Motivate need to
  – Recognize and exploit commuting operations
  – In deterministic parallel computations

• Present issues that arise when
  – Formalize concept of commuting operations
    (especially for linked data structures)
  – Formalize connection between commuting operations and deterministic execution
Context

• Static program verification
• Analyze program before it runs
• Two (dual) options
  – Either analyze parallel program,
    Verify that program will execute deterministically
    • If give program same input
    • All parallel executions will produce same output
      (generalize to equivalent outputs)
  – Or, analyze sequential program,
    Automatically generate deterministic parallel program
A Model of Parallel Computing

Parallel Threads  Heap  Parallel Threads

Goal:

Deterministic Execution

Parallel Updates: Read, Write
A Model of Parallel Computing

Parallel Threads

Heap

Parallel Threads

If Threads Update Disjoint Objects
Then Get Deterministic Execution
A Model of Parallel Computing

Parallel Threads  |  Heap  |  Parallel Threads

But If Threads Update Same Object
Then May Get Nondeterministic Execution
Issues With Conflicting Updates

• Atomicity Violations
  – Mutual exclusion locks, transactional memory, ...
  – Not topic of this talk

• Divergent Object States

![Diagram showing state transitions and updates](image-url)
One Common Solution

- Totally order threads (serial program order)
- Execute updates in thread order
Enforcing Update Order

Need to Delay Thread B Update!

- Save Thread B state (time, space overhead)
- Context switch (time overhead)
- Wait for Thread A to finish update₀ (critical path gets longer)

Machine Gets Bogged Down With Heavyweight Mechanism
Enforcing Update Order

Thread A

if (c) update_0

Thread B

update_1

Need to Delay Thread B Update!

• Save Thread B state (time, space overhead)
• Context switch (time overhead)
• Wait for Thread A to finish update_0 (critical path gets longer)
• update_0 may not even happen!

Machine Gets Bogged Down With Heavyweight Mechanism
A Better Solution (when available)

Exploit Commuting Updates

State_0

Update_0  Update_1

Update_1  Update_0

State_1  State_2
A Better Solution (when available)

Exploit Commuting Updates

If All Conflicting Updates Commute
You Get Deterministic Execution
What Commuting Updates Give You

Only Need Atomicity Mechanism
Not Heavyweight Ordering Mechanism
Kinds of Commuting Updates

• Reads and Writes (Bernstein 1966)
  
  \((x=6 \; | \; x=5)\), rest of computation does not access \(x\)

• Identical States (Rinard and Diniz 1996)
  
  \((x+=6 \; | \; x+=5)\), some graph algorithms

• Semantically Equivalent States
  
  List \(l\); \((l.insert(5) \; | \; l.insert(6))\)

• Observationally Equivalent States
  
  List \(l\); \((l.insert(5) \; | \; l.insert(6))\); print \(l.sum()\); output = 11
  
  List \(l\); \((l.insert(5) \; | \; l.insert(6))\); print \(l\);
  
  output = 5 6 vs. output = 6 5
More Detailed Example
(Map Implemented as Hash Table)

- How to formalize semantic commutativity
- Illustrate connection between
  - Semantic commutativity
  - Deterministic execution
Semantic Commutativity

Concrete Implementation

class map {
    void insert(int key, int val);
    int lookup(int key);
}

parallel for (i = 0; i < n; i++)
    m.insert(i, f(i));

Abstract Specification

abstract state S ⊆ int x int,
S = (old S – {<key, _>}) U {<key, val>}

S ∈ S  ⇒  ret = val
S ⊄ S  ⇒  ret = nil

S = {<i, f(i)> . 0 ≤ i < n}

{<0,0>, <1,2>, <2,4>, <3,6>}

One Outcome

Another Outcome

Always Get Same Abstract State
Connection Between Semantic Commutativity and Deterministic Execution
Key Question for Deterministic Execution

Does Rest of Program Observe Difference Between (Semantically Equivalent) Concrete States?

Depends On
Observations It Can Make
Observations It Does Make
What Observations Does Map Provide?

class map {
    void insert(int key, int val);
    int lookup(int key);
}

Calls to insert observe nothing

S = (old S – {<key, _>}) ∪ {<key, val>}

<key, val> \in S \implies ret = val

<key, _> \notin S \implies ret = nil

Calls to lookup observe

value that key maps to

or nil (if key maps to no value)

If maps have same abstract states

then program can’t observe difference

even if maps have different concrete states

Nondeterminism is contained within class

Program executes deterministically!
Putting Pieces Together

• If reorder insert operations
  – May get different concrete states
  – But abstract states are same
• Same abstract states imply same observables
• Same observables implies deterministic execution
• If reorder insert operations, still get deterministic execution
• Determinism conversion happens at class boundary
What About More Operations?

class map {
    void insert(int key, int val);
    int lookup(int key);
    Set<int> values(); // Returns set of values in map
}

parallel for (i = 0; i < n; i++)
    m.insert(i, f(i)); // Creates map

lastValue = -1;
for (v : m.values()) {
    if (lastValue > v) crash;
    lastValue = v;
}

Nondeterminism escapes via the values() operation
Different Scenario, Different Outcome

class map {
    void insert(int key, int val);
    int lookup(int key);
    Set<int> values();
}

parallel for (i = 0; i < n; i++)
    m.insert(i, f(i));

sum = 0;
for (v : m.values()) {
    sum += v;
}

Returns set of values in map
Creates map
Different concrete states
Same abstract states
Iterates over set
Same result regardless of order!

Sum kills nondeterminism (outside class)
Research Agenda

• Verify semantic equivalence of different execution orders of operations on objects
  – Find operations that generate same abstract state
  – Or equivalent states (equivalence property)

• Assuming semantic equivalence, verify deterministic execution
  – Trace flow of information out of abstract state
  – Find out where nondeterminism is killed
    • Ideally killed inside class
    • In more challenging cases, killed outside class
  – Key concept – observational power
Commutativity Consumers

• *A Type and Effect System for Deterministic Parallel Java.*

• *Revisiting the Sequential Programming Model for Multi-Core.*
  M. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. August, Micro 2007

• *Optimistic Parallelism Requires Abstractions.*

• *The Design, Implementation, and Evaluation of Jade.*
  M. Rinard and M. Lam, TOPLAS 1997
Commutativity Producers

• Semantic Equivalence
  – *An Integrated Proof Language for Imperative Programs.* Karen Zee, Viktor Kuncak, and Martin Rinard, PLDI 2009
  – *Full Functional Verification of Linked Data Structures.* Karen Zee, Viktor Kuncak, and Martin Rinard, PLDI 2008

• Observational Equivalence
  – *Commutativity Analysis for Software Parallelization: Letting Program Transformations See the Big Picture.* Farhana Aleen and Nathan Clark, ASPLOS 2009
Bigger Picture

• Exploiting (semantic) commutativity will be a critical part of future parallel computing efforts.
• Sophisticated tools for reasoning about software are (finally) now available.
• Should be able to leverage this capability to
  – Formalize and verify semantic commutativity
  – Verify connection between semantic commutativity, observational equivalence, and deterministic execution
• Interesting research problem with significant practical impact
Am I Still Doing Research In This Area?

• Yes (static analysis/program verification)
  – Semantic equivalence of commuting operations
  – Observational determinism in presence of commuting operations

• And No (automatic parallelization)
  – Nondeterministic parallelizations of serial programs
  – Change result program produces
  – But with statistical accuracy guarantees
Application Characteristics Changing

• Accuracy/Quality of service key issue
• Correctness not even meaningful concept
  – Video encoding, image processing
  – Search/information retrieval
  – Machine learning
• Lots of flexibility in output
• How/should we exploit this fact?