Parallel Data Structures

Story so far

- **Wirth’s motto**
  - Algorithm + Data structure = Program
- **So far, we have studied**
  - parallelism in regular and irregular algorithms
  - scheduling techniques for exploiting parallelism on multicores
- **Now let us study parallel data structures**

Parallel data structure

- Class for implementing abstract data type (ADT) used in Joe programs
  - (eg) Graph, map, collection, accumulator
- **Need to support concurrent API calls from Joe program**
  - (eg) several concurrently executing iterations of Galois iterator may make calls to methods of ADT
- **Three concerns**
  - must work smoothly with semantics of Galois iterators
  - overhead for supporting desired level of concurrency should be small
  - code should be easy to write and maintain

Working smoothly with Galois iterators

- **Consider two concurrent iterations of an unordered Galois iterators**
  - orange iteration
  - green iteration
- **Iterations may make overlapping invocations to methods of an object**
- **Semantics of Galois iterators:** output of program must be same as if the iterations were performed in some sequential order (serializability)
  - first orange, then green
  - or vice versa
Pictorially

Object: queue
Methods: enq(), deq()

Iteration 1
Thread 1 starts executing Stephanie's code for m1
Thread 1 completes Stephanie's code and resumes Joe code

Iteration 2
Thread 1 completes Joe's code

Two concerns

Object: o
Methods: m1, m2, m3, m4, m5

Iteration 1
Thread 1 starts executing Stephanie's code for m1
Thread 1 completes Stephanie's code and resumes Joe code

Iteration 2
Thread 1 completes Joe's code

Some simple mechanisms

- Conservative locking
- Deadlock-free
- Hard to know what to lock upfront: (e.g.) DMR
- How does this address serializability?

- Two-phase locking
- Incremental
- Requires rollback
  - Old idea: Eswaran and Gray (1976)
  - We will call it catch-and-keep

Problem: potential for deadlock

- Problem: what if thread t1 needs to acquire a lock on an object o, but that lock is already held by another thread t2?
- If t1 just waits for t2 to release the lock, we may get deadlock

- Solution:
  - If a thread tries to acquire a lock that is held by another thread, it reports the problem to the runtime system.
  - Runtime system will rollback one of the threads, permitting the other to continue.
  - To permit rollback, runtime system must keep a log of all modifications to objects made by a thread, so that the thread can be rolled back if necessary
  - Log is a list of <object-name, field, old value> tuples
Discussion

• Stephanie’s job
  – write sequential data structure class
  – add a lock to each class
  – instrument methods to log values before overwriting
  – instrument methods to proceed only after relevant lock is acquired
  – object-based approach
• Holding locks until the end of the iteration prevents cascading roll-backs
  – compare with Timewarp implementation

Performance problem

• Iterations can execute in parallel only if they access disjoint sets of objects
  – locking policy is catch-and-keep
• In our applications, some objects are accessed by all iterations
  – (eg) workset, graph
• With this implementation,
  – lock on workset will be held by some iteration for entire execution of iteration
  – other threads will not be able to get work
  – lock on graph object will be held by some iteration
  – even if other threads got work, they cannot access graph

Catch-and-release objects

1. Use lock to ensure consistency but release lock on object after method invocation is complete
2. Check commutativity explicitly in gatekeeper object
   – Maintains serializability
3. Need inverse method to undo the effect of method invocation in case of rollback

Gatekeeping
Gatekeeping

- Gatekeeper ensures that outstanding operations commute with each other
  - Operation (transaction) = sequence of method invocations by single thread
- Catch-and-keep is a simple gatekeeper
  - And so is transactional memory, also similar ideas in databases
- But for max concurrency, we want to allow as many "semantically" commuting invocations as possible

KD-Trees

- Spatial data-structure
  - nearest(point) : point
  - add(point) : boolean
  - remove(point) : boolean
- Typically implemented as a tree
  - But finding nearest is a little more complicated
- Would like nearest and add to be concurrent if possible
  - When? Why?

Commutativity Conditions

<table>
<thead>
<tr>
<th>Definitions:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d(a, b)$ is a given fixed distance metric such that $d(x, y)$ returns the shortest path according to $d$.</td>
</tr>
<tr>
<td>(1) $\text{nearest}(x)_r$ - commutes with $\text{nearest}(y)_r$ if $d(x, y) + d(x, r) + d(y, r) &lt; d(x, r) + d(y, r)$.</td>
</tr>
<tr>
<td>(2) $\text{nearest}(x)_r$ - commutes with $\text{add}(y, r)$ if $d(x, y) + d(x, r) + d(y, r) &lt; d(x, r) + d(y, r)$.</td>
</tr>
<tr>
<td>(3) $\text{nearest}(x)_r$ - commutes with $\text{remove}(y)$ if $d(x, y) + d(x, r) + d(y, r) &lt; d(x, r) + d(y, r)$.</td>
</tr>
<tr>
<td>(4) $\text{add}(x, r)$ - commutes with $\text{nearest}(y)_r$ if $d(x, y) + d(x, r) + d(y, r) &lt; d(x, r) + d(y, r)$.</td>
</tr>
<tr>
<td>(5) $\text{add}(x, r)$ - commutes with $\text{add}(y, r)$ if $d(x, y) + d(x, r) + d(y, r) &lt; d(x, r) + d(y, r)$.</td>
</tr>
<tr>
<td>(6) $\text{add}(x, r)$ - commutes with $\text{remove}(y)$ if $d(x, y) + d(x, r) + d(y, r) &lt; d(x, r) + d(y, r)$.</td>
</tr>
</tbody>
</table>

Figure 9: Commutativity conditions for kd-tree data structure.

Gatekeeping

- Solution: keep log of nearest invocations and make sure that no add invalidates them
  - More general solution is to log all invocations and evaluate commutativity conditions wrt outstanding invocations
    - Tricky when conditions depend on state of data structure
      - Forward and general gatekeeping approaches
  - Other issues:
    - Gatekeeper itself should be concurrent
    - Inverse method should have same commutativity as forward method
Gatekeeping in Galois

- Most commutativity conditions are simple
  - Equalities and disequalities of parameters and return values
- Simple implementation
  - Instrument data structure
  - Acquire locks on objects in different modes
  - Abstract locking approach
  - How is the different than the object-based approach?

Partition-based locking

- If topology is graph or grid, we can partition the data structure and associate locks with partitions
- How do we ensure consistency?
- How do we ensure commutativity?

Linearizability: Intuitively for the single lock queue

- Sequential Consistency
  - Concurrent operations happen in some sequential order
  - Partial order on non-overlapping operations
- Linearizability
  - Enq(x) → Enq(y) → Deq(x)
Partitioning arrays

Standard array partitions

- Standard partitions supported in Scalapack (dense numerical linear algebra library), High-performance FORTRAN (HPF), etc.
  - block
  - cyclic
  - block-cyclic
- Block-cyclic subsumes the other two

Block

Cyclic partitioning
Common use of block-cyclic

- BLOCK distribution: small number of processors ends up with all the work after a while
- CYCLIC distribution: better load balance
- BLOCK-CYCLIC: lower communication costs than CYCLIC

Distributing both dimensions

- # of array distribution dimensions
- # of dimensions of processor grid
- 2-D processor grid

A (4,8)