Threads Redux

Changing thread semantics

Grace: Overview

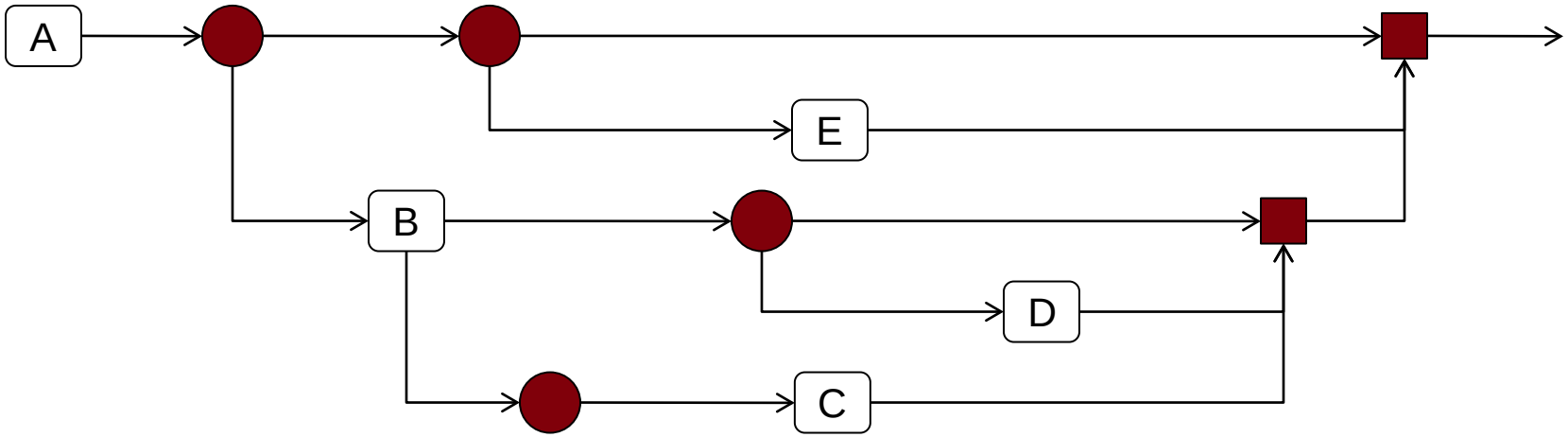
- Goal
  - Eliminate classes of concurrency errors
  - For applications using fork-join parallelism
  - Not appropriate for
    - Reactive systems (servers)
    - Systems with condition-based synchronization
- Approach
  - Fully isolated threads (turning threads into processes)
    - Leveraging virtual memory protections
    - No need for locks (turn locks into no-ops)
  - Sequential commit protocol
    - Threads commit in program order
    - Guarantees execution equivalent to serial execution
  - Speculative thread execution

Grace: Overview

- Features
  - Overhead amortized over lifetime of thread
  - Supports threads with irrevocable operations (e.g., I/O operations)
  - Less memory overhead than comparable transactional memory techniques

Concurrency error	Cause	Grace Prevention
Deadlock	Cyclic lock acquisition	No locking
Race condition	Unguarded updates	Updates committed deterministically
Atomicity violation	Interleaved updates	Threads run atomically
Order violation	Threads scheduled in unexpected order	Threads execute in program order

Fork-Join Parallelism

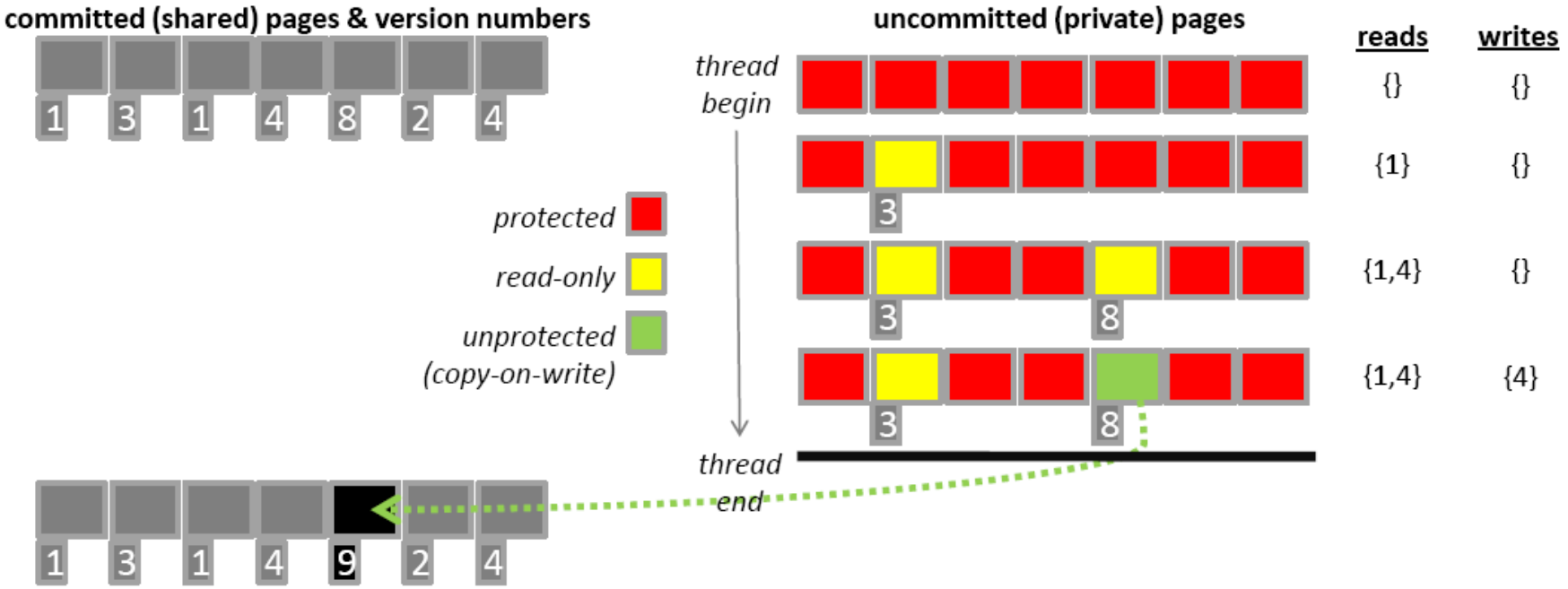



```
// Run f(x) and g(y) in parallel.
t1 = spawn f(x);
t2 = spawn g(y);
// Wait for both to complete.
sync;
```

```
// Run f(x) to completion, then g(y).
t1 = spawn f(x);
t2 = spawn g(y);
// Wait for both to complete.
sync;
```

- Serial elision used in Grace
- In Grace, fork-join parallelism is behaviorally equivalent to its sequential counterpart

Thread Execution

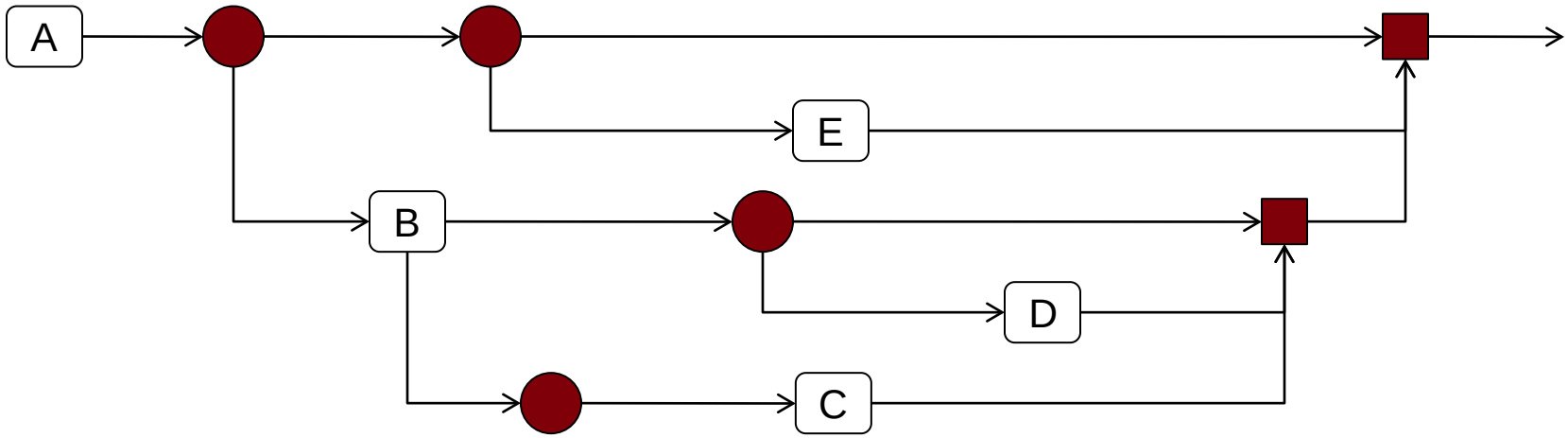


- Each thread has private copies of changed pages (guarantees thread isolation)
- Page protection mechanisms used to detect reads/writes
- Access tracking and conflict detection at page granularity
- Page version numbers allow detection of conflicts
- In case of conflict, thread aborts/restarts

Commit order

- Rules
  - Parent waits for
    - youngest (most recently created) child
  - Child waits for
    - youngest (most recently created) elder sibling, if it exists, or
    - the parent's youngest (most recently created) elder sibling
- Equivalent to post-order traversal of execution tree
- Guarantees equivalence to sequential execution

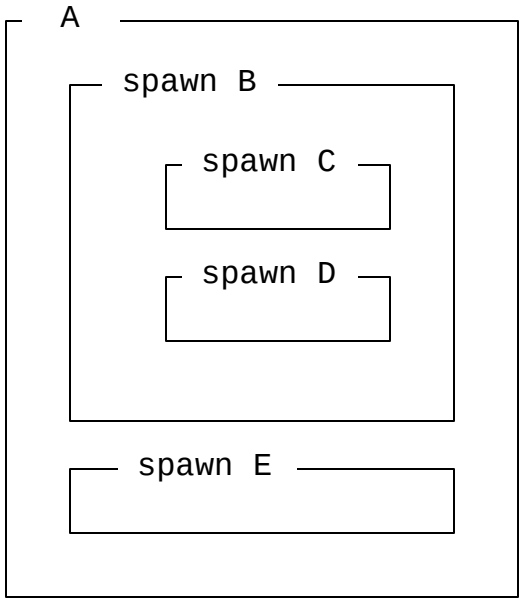
Serial elision and commit order

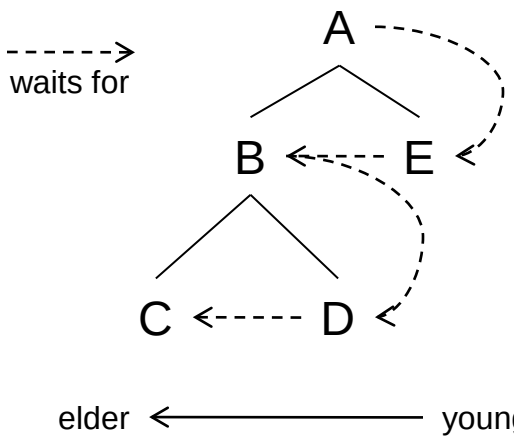



```
A{
    spawn B{
          spawn C {…};
          spawn D {…};
          synch;              //join for C and D
    }
    spawn E {…};
    synch;                    //join for B and E
}
```

Serial elision and commit order

```
A{
    spawn B{
          spawn C {…};
          spawn D {…};
          synch;
    }
    spawn E {…};
    synch;
}
```





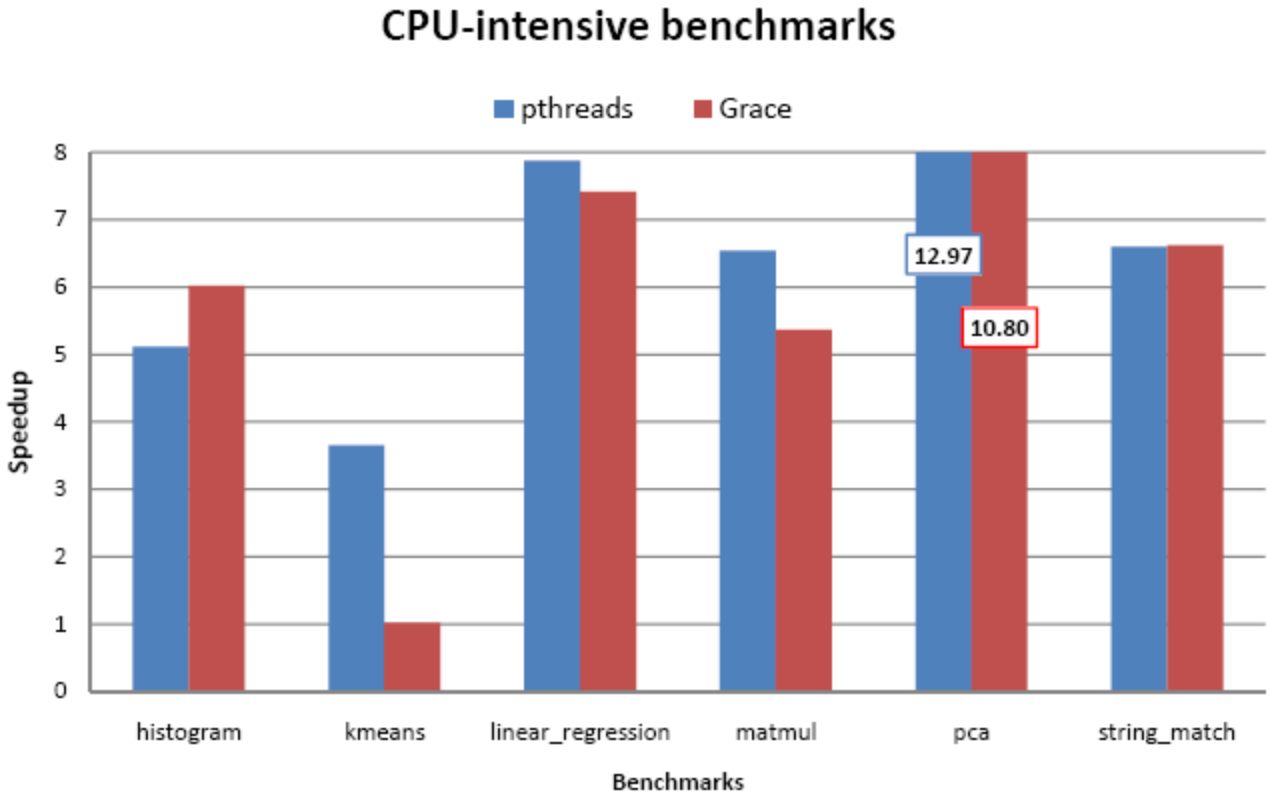
postorder traveral

elder younger

Handling irrevocable I/O operations

- Each thread
  - buffers I/O operations
  - commits I/O operations with memory updates
- Thread attempting irrevocable I/O operation
  - Waits for its immediate predecessor to commit
  - Checks for consistency with committed state
    - If consistent, perform irrevocable I/O operation
    - Else, restart and perform irrevocable I/O operation as part of new execution

Performance



- On 8 core system
- Minimal (1-16 lines) changes for Grace version
- Speedup for Grace comparable to pthreads but with guarantees of absence of concurrency errors

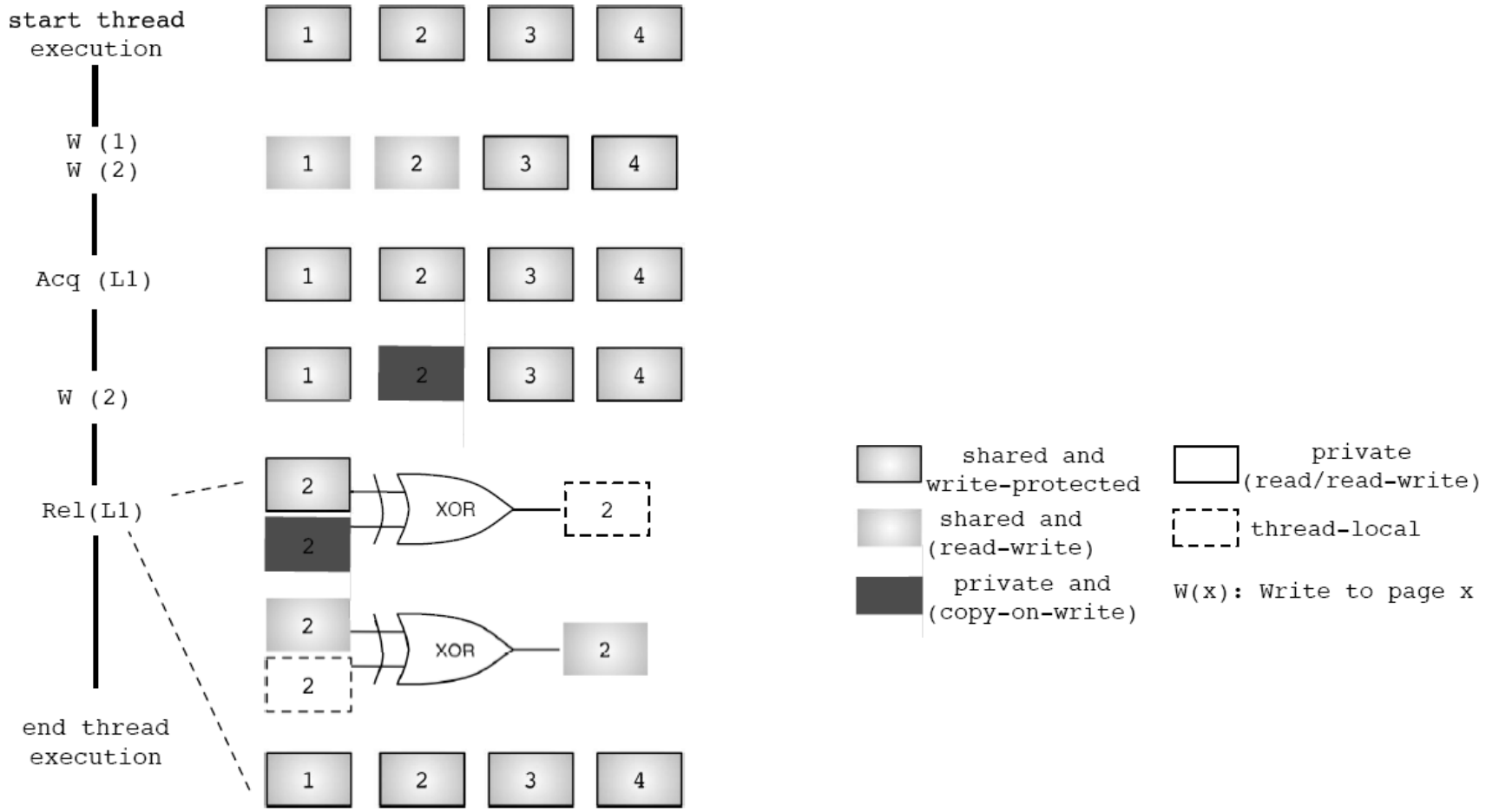
Sammati

- Goals
  - Eliminates deadlock in threaded codes
  - Transparent to application (no code changes)
  - Allows arbitrary use of locks for concurrency control
  - Achieves composability of lock based codes
  - Works for weakly typed languages (e.g., C/C++)
- Approach
  - Containment
    - Identify memory accesses associated with a lock
    - Keep updates private while lock is held
    - Make updates visible when lock is released
  - Deadlock handling
    - Automatic detection on lock acquisition
    - Resolves deadlock by restarting one thread

Sammati

- Key mechanisms
  - Transparent mechanism for privatizing memory updates within a critical section
  - Visibility rules that
    - preserve lock semantics
    - allow containment
  - Deadlock detection and recovery

Privatizing memory



Visibility rules

- Locks not nested

```
x=y=0;

acquire (L1);
  x++;
release (L1);
```

Begin privatizing changes to x when lock is acquired.

Allow changes to x to become (globally) visible when lock is released.

Visibility rules

- Nested locks

```
x=y=0;

acquire (L1);
  acquire(L2);
    x++;
  release(L2);
  acquire(L3);
    x++;
    y++;
  release(L3);
release (L1);
```

Cannot allow changes to x to become (globally) visible when L2 is released because of possible rollback to L1.

Cannot allow changes to x or y to become (globally) visible when L3 is released because of possible rollback to L1.

Rule: make changes visible when all locks released.

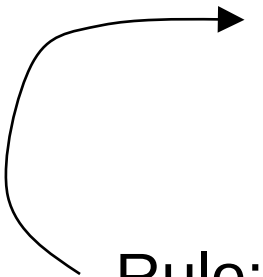
Visibility rules

- Overlapping (unstructured) locks

```
x=y=0;

acquire (L1);
  x++;
  acquire(L2);
    y++;
release(L1);
  release (L2);
```

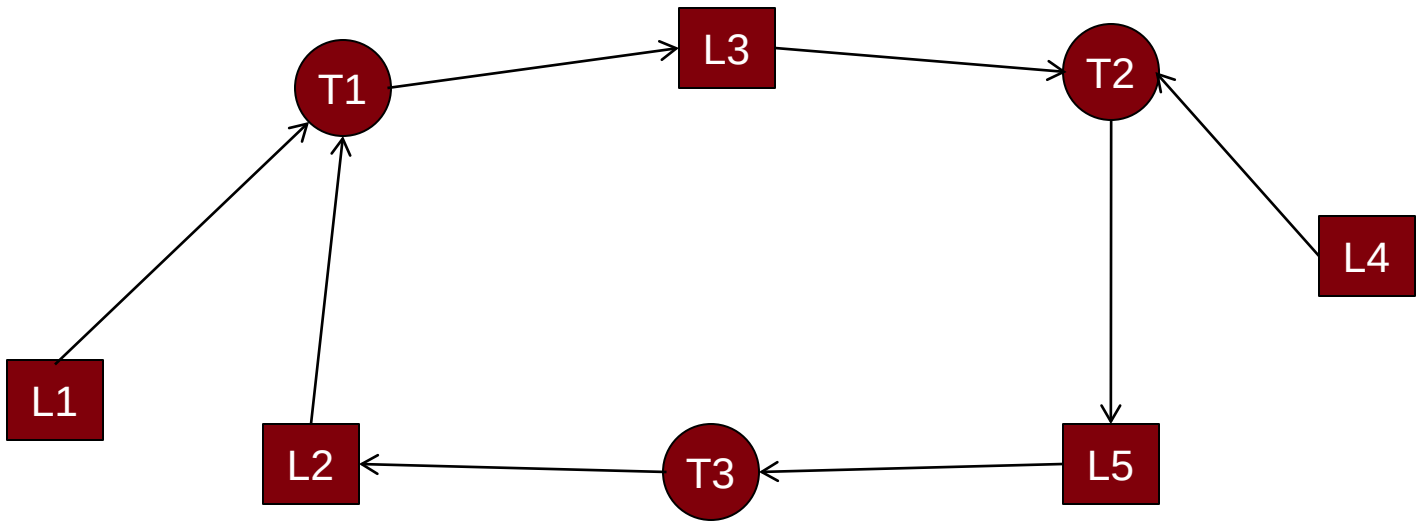
Cannot determine transparently with which lock(s) the data should be associated.



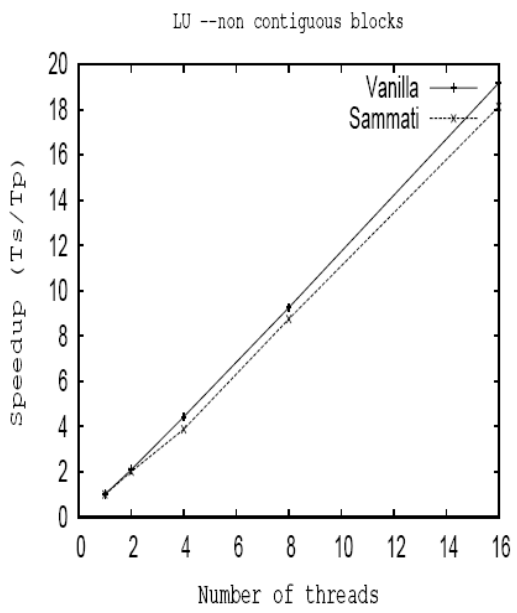
Rule: make changes visible when all locks released.

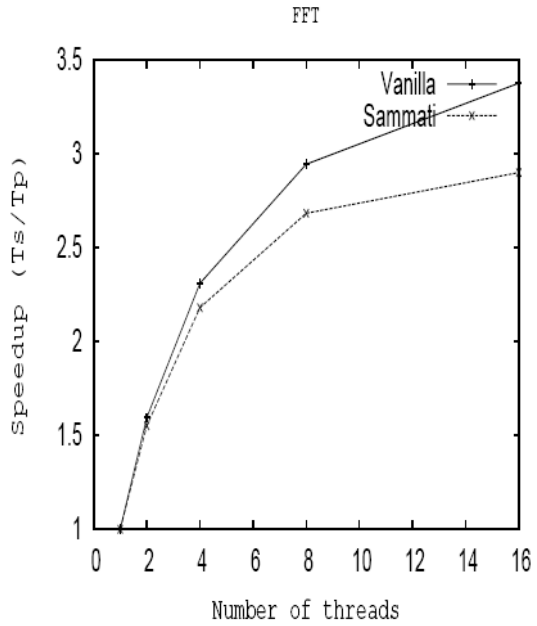
Deadlock detection

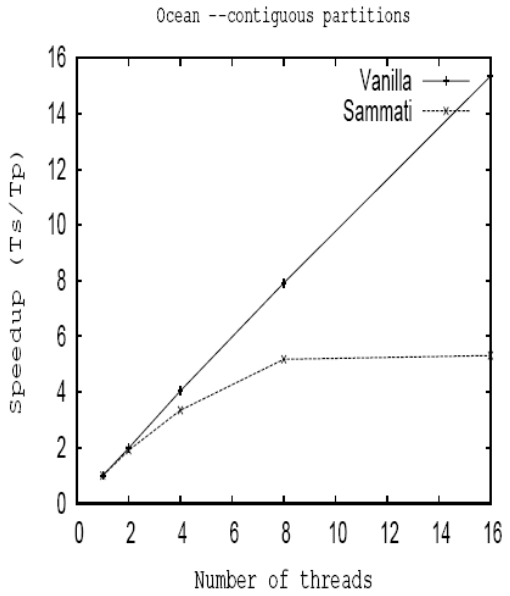
- A thread may only wait for (be blocked trying to acquire) one lock at a time
- Because thread state is privatized, deadlock can be resolved by rolling back and restarting one thread.



Performance

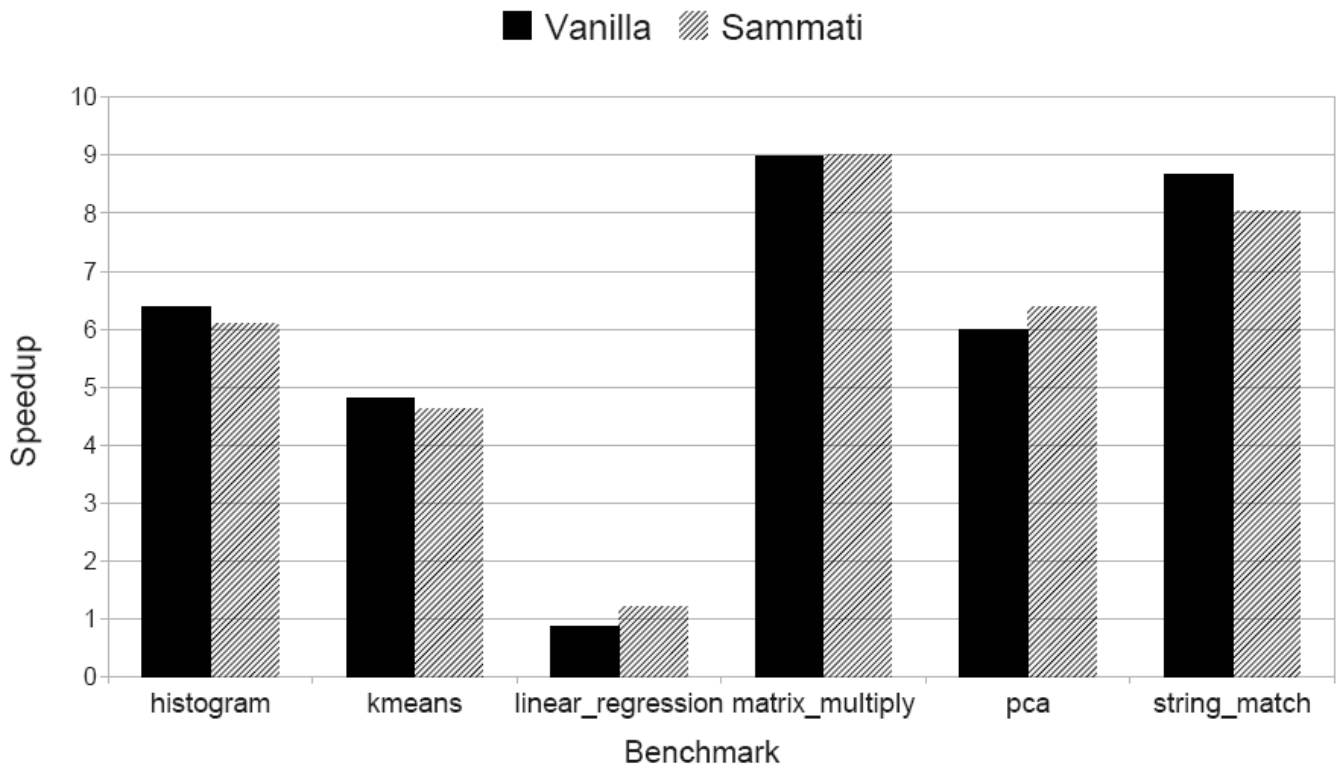






- SPLASH benchmark
- Summati performance generally comparable to pthreads (one notable exception)

Performance



- Phoenix benchmark
- Summati performance generally comparable to pthreads