

Lecture 25: Parallelism & Concurrency

CS 62

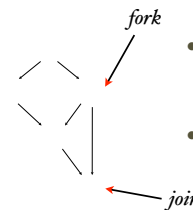
Fall 2015

Kim Bruce & Michael Bannister

*Some slides based on those from Dan Grossman,
U. of Washington*

Program Graph

- Program using fork and join can be seen as directed acyclic graph (DAG).
- Nodes: pieces of work
- Edges: dependencies - source must finish before start destination



- Fork command finishes node and makes two edges out:
 - New thread & continuation of old
- Join ends node & makes new node w/ 2 edges coming in

Performance

- Let T_P be running time if there are P processors
- Work = T_1 = sum of run-time of all nodes in DAG
- Span = T_∞ = sum of run-time of all nodes on most expensive path in DAG
- Speed-up on P processors = T_1/T_P

What does it mean?

- Guarantee: $T_P = O((T_1 / P) + T_\infty)$
 - No implementation can beat $O(T_\infty)$ by more than constant factor.
 - No implementation on P processors can beat $O((T_1 / P))$
 - So framework on average gives best can do, assuming user did best possible.
- Bottom line:
 - Focus on your algos, data structures, & cut-offs rather than # processors and scheduling.
 - Just need T_1 , T_∞ , and P to analyze running time

Examples

- Recall: $T_p = O((T_1 / P) + T_\infty)$
- For summing:
 - $T_1 = O(n)$
 - $T_\infty = O(\log n)$
 - So expect $T_p = O(n/P + \log n)$
- If instead:
 - $T_1 = O(n^2)$
 - $T_\infty = O(n)$
 - Then expect $T_p = O(n^2/P + n)$

Amdahl's Law

- Upper bound on speed-up!
 - Suppose the work (time to run w/one processor) is 1 unit time.
 - Let S be portion of execution that cannot be parallelized
 - $T_1 = S + (1 - S) = 1$
 - Suppose get perfect speedup on parallel portion.
 - $T_p = S + (1-S) / P$
 - Then overall speedup with P processors (Amdahl's law):
 - $T_1 / T_p = 1 / (S + (1-S) / P)$
 - Parallelism (∞ processors) is: $T_1 / T_\infty = 1 / S$

Bad News!

- $T_1 / T_\infty = 1 / S$
- If 33% of program is sequential, then millions of processors won't give speedup over 3.
- From 1980 - 2005, every 12 years gave 100x speedup
 - Now suppose clock speed is same but 256 processors instead of 1.
 - To get 100x speedup, need $100 \leq 1 / (S + (1-S) / P)$
 - Solve to get solution $S \leq .0061$, so need 99.4% perfectly parallel.

Moral

- May not be able to speed up existing algos much, but might find new parallel algos.
- Can change what we compute
 - Computer graphics now much better in video games with GPU's -- not much faster, but much more detail.

A Last Example: Sorting

- Quicksort, sequential, in-place, expected time $O(n \log n)$
 - Pick pivot elt $O(1)$
 - Partition data into $O(n)$
 - A: less than pivot
 - B: pivot
 - C: greater than pivot
 - Recursively sort A, C $2 * T(n/2)$
 - Now do in parallel, so $T(n/2)$
 - $n + n/2 + n/4 \dots = 2n$, which is $O(n)$
 - With work, can improve more and get $O(\log^2 n)$

Shared Memory Concurrency

Sharing Resources

- Have been studying parallel algorithms using fork-join
 - Reduce span via parallel tasks
- Algorithms all had a very simple structure to avoid race conditions
 - Each thread had memory “only it accessed”
 - Example: array sub-range
 - On fork, “loaned” some of its memory to “forkee” and did not access that memory again until after join on the “forkee”

But ...

- Strategy won't work well when:
 - Memory accessed by threads is overlapping or unpredictable
 - Threads are doing independent tasks needing access to same resources (rather than implementing the same algorithm)
- How do we control access?

Concurrent Programming

- **Concurrency:** Allowing simultaneous or interleaved access to shared resources from multiple clients
- Requires coordination, particularly synchronization to avoid incorrect simultaneous access: make somebody block
 - join is not what we want
 - block until another thread is “done using what we need” not “completely done executing”

Non-Deterministic Computation

- Even correct concurrent applications are usually highly *non-deterministic*: how threads are scheduled affects *what* operations from other threads they see and *when* they see them.
- Non-repeatability complicates testing and debugging

Examples

- **Multiple threads:**
 - Processing different bank-account operations
 - What if 2 threads change the same account at the same time?
- **Using a shared cache of recent files**
 - What if 2 threads insert the same file at the same time?
- **Creating pipeline w/ queue for handing work to next thread in sequence?**
 - What if enqueueer and dequeuer adjust a circular array queue at the same time?

Threads again?!?

- **Not about speed, but**
 - Code structure for responsiveness
 - Example: Respond to GUI events in one thread while another thread is performing an expensive computation
 - Processor utilization (mask I/O latency)
 - If 1 thread “goes to disk,” have something else to do
 - Failure isolation
 - Convenient structure if want to interleave multiple tasks and don't want an exception in one to stop the other

Sharing is the Key

- Common to have:
 - Different threads access the same resources in an unpredictable order or even at about the same time
 - But program correctness requires that simultaneous access be prevented using synchronization
 - Simultaneous access is rare
 - Makes testing difficult
 - Must be much more disciplined when designing / implementing a concurrent program
 - Will discuss common idioms known to work

Canonical Example

- Several ATM's accessing same account.
 - See ATM₂

Bad Interleavings

Interleaved **changeBalance(-100)** calls on the same account
– Assume initial **balance 150**

	Thread 1	Thread 2
Time ↓	<code>int nb = b + amount;</code>	<code>int nb = b + amount;</code>
	<code>if(nb < 0)</code> <code> throw new ...;</code> <code>balance = nb;</code>	<code>if(nb < 0)</code> <code> throw new ...;</code> <code>balance = nb;</code>
	<code>if(nb < 0)</code> <code> throw new ...;</code> <code>balance = nb;</code>	<code>if(nb < 0)</code> <code> throw new ...;</code> <code>balance = nb;</code>
	<code>if(nb < 0)</code> <code> throw new ...;</code> <code>balance = nb;</code>	<code>if(nb < 0)</code> <code> throw new ...;</code> <code>balance = nb;</code>

“Lost withdraw” –
unhappy bank

Interleaving is the Problem

- Suppose:
 - Thread T₁ calls `changeBalance(-100)`
 - Thread T₂ calls `changeBalance(-100)`
- If second call starts before first finishes, we say the calls **interleave**
 - Could happen even with one processor since a thread can be pre-empted at any point for time-slicing
- If x and y refer to different accounts, **no problem**
 - “You cook in your kitchen while I cook in mine”
 - But if x and y alias, possible trouble...

Problems with Account

- Get wrong answers!
- Try to fix by getting balance again, rather than using `newBalance`.
 - Still can have interleaving, though less likely
 - Can go negative w/ wrong interleaving!

Solve with Mutual Exclusion

- At most one thread withdraws from account A at one time.
- Areas where don't want two threads executing called *critical sections*.
- Programmer needs to decide where, as compiler doesn't know intentions.

Java Solution

- *Re-entrant locks* via synchronized blocks
- Syntax:
 - **synchronized (expression) {statements}**
- Evaluates expression to an object and tries to grab it as a lock
 - If no other process is holding it, grabs it and executes statements. Releasing when finishes statements.
 - If another process is holding it, waits until it is released.
- Net result: Only one thread at a time can execute a synchronized block w/same lock

Correct Code

```
public class Account {
    private myLock = new Object();

    ...
    // return balance
    public int getBalance() {
        synchronized(myLock){ return balance; }
    }

    // update balance by adding amount
    public void changeBalance(int amount) {
        synchronized(myLock) {
            int newBalance = balance + amount;
            display.setText("" + newBalance);
            balance = newBalance;
        }
    }
}
```

Better Code

```
public class Account {
    ...
    // return balance
    public int getBalance() {
        synchronized(this){ return balance; }
    }

    // update balance by adding amount
    public void changeBalance(int amount) {
        synchronized(this) {
            int newBalance = balance + amount;
            display.setText("" + newBalance);
            balance = newBalance;
        }
    }
}
```

Best Code

```
public class Account {
    ...
    // return balance
    synchronized public int getBalance() {
        return balance;
    }

    // update balance by adding amount
    synchronized public void changeBalance(int amount) {
        int newBalance = balance + amount;
        display.setText("" + newBalance);
        balance = newBalance;
    }
}
```

Reentrant Locks

- If thread holds lock when executing code, then further method calls within block don't need to reacquire same lock.
 - E.g., Methods m and n are both synchronized with same lock (e.g., with *this*), and execution of m results in calling n. Then once thread has the lock executing m, no delay in calling n.

Responsiveness

Maze Program

- Uses stack to solve a maze.
- When user clicks “solve maze” button, spawns Thread to solve maze.
- What happens if send “run” instead of “start”?

Non-Event-Driven Programming

- Program in control.
- Program can ask for input at any point, with program control depending on input.
- But user can't interrupt program
 - Only give input when program ready

Event-Driven Programming

- Control inverted.
 - User takes action, program responds
- GUI components (buttons, mouse, etc.) have “listeners” associated with them that are to be notified when component generates an event.
- Listeners then take action to respond to event.

Event-Driven Programming in Java

- When an event occurs, it is posted to appropriate event queue.
 - Java GUI components share an event queue.
 - Any thread can post to the queue
 - Only the “event thread” can remove event from the queue.
- When event removed from queue, thread executes the appropriate method of listener w/ event as parameter.

Example: Maze-Solver

- Start button ⇒ StartListener object
- Clear button ⇒ ClearAndChooseListener
- Maze choice ⇒ ClearAndChooseListener
- Speed slider ⇒ SpeedListener

Listeners

- Different kinds of GUI items require different kinds of listeners:
 - Button -- ActionListener
 - Mouse -- MouseListener, MouseMotionListener
 - Slider -- ChangeListener
- See GUI cheatsheet on documentation web page

Event Thread

- Removes events from queue
- Executes appropriate methods in listeners
- Also handles repaint events
- Must remain responsive!
 - Code must complete and return quickly
 - If not, then spawn new thread!

Why did Maze Freeze?

- Solver animation was being run by event thread
- Because didn't return until solved, was not available to remove events from queue.
 - Could not respond to GUI controls
 - Could not paint screen

Off to the Races

- A *race* condition occurs when the computation result depends on scheduling (how threads are interleaved). Answer depends on shared state.
- Bugs that exist only due to concurrency
 - No interleaved scheduling with 1 thread
- Typically, problem is some intermediate state that “messes up” a concurrent thread that “sees” that state

Example

```
class Stack<E> {  
    ...  
    synchronized void push(E val) { ... }  
    synchronized E pop() {  
        if(isEmpty())  
            throw new StackEmptyException();  
        ...  
    }  
  
    E peek() {  
        E ans = pop();  
        push(ans);  
        return ans;  
    }  
}
```

Sequentially Fine

- Correct in sequential world
- May need to write this way, if only have access to push, pop, & isEmpty methods.
- peek() has no overall effect on data structure
 - reads rather than writes

Concurrently Flawed

- Way it's implemented creates an inconsistent intermediate state
 - Even though calls to push and pop are synchronized so no data races on the underlying array/list/whatever
 - (A data race is simultaneous (unsynchronized) read/write or write/write of the same memory: more on this soon)
- This intermediate state should not be exposed
 - Leads to several wrong interleavings...

Lose Invariants

- Want: If there is at least one push and no pops, then isEmpty always returns false.
- Fails with two threads if one is doing a peek, other isEmpty, & unlucky.
- Gets worse: Can lose LIFO property
 - Problem do push while doing peek.
- Want: If # pushes > # pops then peek never throws an exception.
 - Can fail if two threads do simultaneous peeks

Solution

- Make peek synchronized (w/same lock)
 - No problem with internal calls to push and pop because locks reentrant
- Just because all changes to state done within synchronized pushes and pops doesn't prevent exposing intermediate state.

- Re-entrant locks allows calls to push and pop if use same lock

From within Stack

```
class Stack<E> {  
    synchronized E peek() {  
        E ans = pop();  
        push(ans);  
        return ans;  
    }  
}
```

From outside Stack

```
class C {  
    <E> E myPeek(Stack<E> s) {  
        synchronized (s) {  
            E ans = s.pop();  
            s.push(ans);  
            return ans;  
        }  
    }  
}
```

Beware of Accessing Changing Data

- Even if unsynchronized methods don't change it.

```
class Stack<E> {  
    private E[] array = (E[])new Object[SIZE];  
    int index = -1;  
    boolean isEmpty() { // unsynchronized: wrong?!  
        return index==-1;  
    }  
    synchronized void push(E val) {  
        array[++index] = val;  
    }  
    synchronized E pop() {  
        return array[index--];  
    }  
    E peek() { // unsynchronized: wrong!  
        return array[index];  
    }  
}
```

Providing Safe Access

- For every memory location (e.g., object field) in your program, you must obey at least one of the following:
 - Thread-local: Don't access the location in > 1 thread
 - Immutable: Don't write to the memory location
 - Synchronized: Use synchronization to control access to the location

