Practical Concurrent and Parallel Programming 7

Peter Sestoft
IT University of Copenhagen

Friday 2016-10-14
Plan for today

- **Threadsafe long integers with “striping”**
- Memory, cache and cache coherence
  - Shared mutable data on multicore is slow
- Graphical user interface toolkits, eg Swing
  - not thread-safe, access from event thread only
- Using SwingWorker for long-running work
  - Progress bar
  - Cancellation
  - Display results as they are generated
- A thread-based lift simulator with GUI
Flashback week 6: Maintaining hashmap size information

• Using a single AtomicLong limits scalability

• We used one size component per stripe:

```java
class StripedMap<K,V> implements OurMap<K,V> {
    private final int[] sizes;
    public int size() {
        ... for-loop summing sizes[i] ...
    }
    ...
}
```

• Fast updates but slow size() queries
  – Very slow if millions of stripes, as in Java 8 CHM

• Java 8 ConcurrentHashMap uses a LongAdder
  – A long counter that scales well with many threads
  – If there are more writes (add) than reads (get)
Thread-safe longs: simple, striped, ...

- Vastly different scalability
  - (a) Java 5’s AtomicLong
  - (b) Home-made single-lock LongCounter (week 1)
  - (c) Home-made striped long using AtomicLongArray
  - (d) Home-made striped long using AtomicLong[]
  - (e) Home-made striped long with scattered allocation
  - (f) Java 8’s LongAdder

- Ideas
  - (c,d,e) Use thread’s hashCode to reduce update collisions
  - (e) Scatter AtomicLongs to avoid false cache line sharing
  - Difference d-e is scattering of AtomicLong objects in memory

<table>
<thead>
<tr>
<th></th>
<th>i7 4c</th>
<th>AMD 32c</th>
<th>Xeon 48c</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>974</td>
<td>3011</td>
<td>667</td>
</tr>
<tr>
<td>(b)</td>
<td>499</td>
<td>14921</td>
<td>815</td>
</tr>
<tr>
<td>(c)</td>
<td>422</td>
<td>1611</td>
<td>956</td>
</tr>
<tr>
<td>(d)</td>
<td>183</td>
<td>-</td>
<td>296</td>
</tr>
<tr>
<td>(e)</td>
<td>114</td>
<td>922</td>
<td>135</td>
</tr>
<tr>
<td>(f)</td>
<td>64</td>
<td>54</td>
<td>22</td>
</tr>
</tbody>
</table>

Wall clock time (ms) for 32 threads making 1 million additions each
Dividing a long into 31 “stripes”

- Two threads unlikely to add to same stripe
- Each thread has a home stripe – “affinity”
  - if accessed by thread, likely to be accessed again
- So, quite fast despite the cost of `hashCode()`
The long adders’ memory layouts

(a) 17
(b) 17
(c) 9 3 5
(d) 9 3 5
(e) 9 3 5
(f) 9 3 5

AtomicLong
LongCounter, lock
AtomicLongArray
AtomicLong[]
AtomicLong[] and mem. scattering
Array of obj with internal padding
Plan for today

• Threadsafe long integers with “striping”
• **Memory, cache and cache coherence**
  – Shared mutable data on multicore is slow
• Graphical user interface toolkits, eg Swing
  – not thread-safe, access from event thread only
• Using SwingWorker for long-running work
  – Progress bar
  – Cancellation
  – Display results as they are generated
• A thread-based lift simulator with GUI
A typical multicore CPU with three levels of cache

- Floating-point register add or mul: 0.4 ns
- RAM access: > 80 ns (single-thread read)
Fix 1: Each processor core has a cache

- Cache = simple hardware hashtable
- Stores recently accessed values from RAM
- Cache is much faster than RAM

Two caches may have different values for a given memory address
Fix 2: Get all caches to agree

- Cache coherence; cache line state = M,E,S,I

<table>
<thead>
<tr>
<th>State</th>
<th>Cache line</th>
<th>Excl</th>
<th>RAM</th>
<th>To read</th>
<th>To write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>Modified by me</td>
<td>Y</td>
<td>stale</td>
<td>from cache</td>
<td>to cache</td>
</tr>
<tr>
<td>Exclusive</td>
<td>Not modified</td>
<td>Y</td>
<td>fresh</td>
<td>from cache</td>
<td>to cache -&gt; M</td>
</tr>
<tr>
<td>Shared</td>
<td>Others have it too</td>
<td>N</td>
<td>fresh</td>
<td>from cache</td>
<td>send invalidate</td>
</tr>
<tr>
<td>Invalid</td>
<td>Not in use by me</td>
<td>-</td>
<td>-</td>
<td>elsewhere</td>
<td>send invalidate</td>
</tr>
</tbody>
</table>

- A cache line
  - has 4 possible states
  - and 12 transitions a-l

- Cache messages
  - sent by cores to others
  - via cache bus
  - to make caches agree

McKenney 2010: Memory barriers
## Transitions and messages

A write in a non-exclusive state requires acknowledge ack* from *all other* cores

Shared mutable state is slow on big machines

<table>
<thead>
<tr>
<th></th>
<th>Cause</th>
<th>I send</th>
<th>I receive</th>
<th>My response</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>a (Send update to RAM)</td>
<td>writeback</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>E</td>
<td>b Write</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>M</td>
<td>c Other wants to write</td>
<td>-</td>
<td>read inv</td>
<td>read resp, inv ack</td>
</tr>
<tr>
<td>I</td>
<td>d Atomic read-mod-write</td>
<td>read inv</td>
<td>read resp, inv ack*</td>
<td>-</td>
</tr>
<tr>
<td>S</td>
<td>e Atomic read-mod-write</td>
<td>read inv</td>
<td>inv ack*</td>
<td>-</td>
</tr>
<tr>
<td>M</td>
<td>f Other wants to read</td>
<td>-</td>
<td>read</td>
<td>read resp</td>
</tr>
<tr>
<td>E</td>
<td>g Other wants to read</td>
<td>-</td>
<td>read</td>
<td>read resp</td>
</tr>
<tr>
<td>S</td>
<td>h Will soon write</td>
<td>inv</td>
<td>inv ack*</td>
<td>-</td>
</tr>
<tr>
<td>E</td>
<td>i Other wants atomic rw</td>
<td>-</td>
<td>read inv</td>
<td>read resp, inv ack</td>
</tr>
<tr>
<td>I</td>
<td>j Want to write</td>
<td>read inv</td>
<td>read resp, inv ack*</td>
<td>-</td>
</tr>
<tr>
<td>I</td>
<td>k Want to read</td>
<td>read</td>
<td>read resp</td>
<td>-</td>
</tr>
<tr>
<td>S</td>
<td>l Other wants to write</td>
<td>-</td>
<td>inv</td>
<td>inv ack 11</td>
</tr>
</tbody>
</table>
Fast and slow cache cases

- The cache is **fast** when
  - the local core “owns” the data (state M or E), or
  - data is shared (S) but local core *only reads* it

- The cache is **slow** when
  - the data is shared (S) and we want to write it, or
  - the data is not in cache (I)
    - possibly because cache line “owned” by another core

<table>
<thead>
<tr>
<th></th>
<th>This core wants to</th>
<th>Messages</th>
<th>Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unshared</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mutable</td>
<td>M M Read cache line</td>
<td>0</td>
<td>fast</td>
</tr>
<tr>
<td></td>
<td>M M Write cache line</td>
<td>0</td>
<td>fast</td>
</tr>
<tr>
<td></td>
<td>E E Read cache line</td>
<td>0</td>
<td>fast</td>
</tr>
<tr>
<td></td>
<td>E M Write cache line</td>
<td>0</td>
<td>fast</td>
</tr>
<tr>
<td>Shared</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>immutable</td>
<td>S S Read cache line</td>
<td>0</td>
<td>fast</td>
</tr>
<tr>
<td></td>
<td>I S Read cache line</td>
<td>1+1</td>
<td>slow</td>
</tr>
<tr>
<td>Shared</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mutable</td>
<td>S M Write cache line</td>
<td>1+N</td>
<td>very slow</td>
</tr>
<tr>
<td></td>
<td>I M Write cache line</td>
<td>1+1+N</td>
<td>very slow</td>
</tr>
</tbody>
</table>
One more performance problem: “false sharing” because of cache lines

- A cache line typically is 64 bytes
  - gives better memory bus utilization
  - prefetches data (in array) that may be needed next
- Thus invalidating one (8 byte) long may invalidate the neighboring 7 longs!
- Frequently written memory locations should not be on the same cache line!
  - even if apparently not shared between threads
- Attempts to fix this by “padding”
  - may look very silly (next slide)
  - are not guaranteed to help
  - yet are used in the Java class library code
Scattering the stripes of a long

class NewLongAdderPadded {
    private final static int NSTRIPES = 31;
    private final AtomicLong[] counters;

    public NewLongAdderPadded() {
        this.counters = new AtomicLong[NSTRIPES];
        for (int stripe=0; stripe<NSTRIPES; stripe++) {
            // Believe it or not, this sometimes speeds up the code,
            // presumably because avoids false sharing of cache lines:
            new Object(); new Object(); new Object(); new Object();
            counters[stripe] = new AtomicLong();
        }
    }
}

- Allocate many AtomicLongs
  - instead of AtomicLongArray

- Scatter the AtomicLongs
  - by allocating some Objects in between

Avoid side-by-side AtomicLong allocation

Unless JVM is too clever

Don’t do this at home
Plan for today

• Atomic longs with “striping” (week 6)
• Memory, cache and cache coherence
  – Shared mutable data on multicore is slow
• **Graphical user interface toolkits, Swing**
  – not thread-safe, access from event thread only
• Using SwingWorker for long-running work
  – Progress bar
  – Cancellation
  – Display results as they are generated
• A thread-based lift simulator with GUI
GUI toolkits are single-threaded

• Java Swing components are **not** thread-safe
  – This is intentional
  – Ditto .NET’s System.Windows.Forms and others

• Multithreaded GUI toolkits
  – are difficult to use
  – deadlock-prone, because actions are initiated both
    • *top-down*: from user towards operating system
    • *bottom-up*: from operating system to user interface
    • locking in different orders ... hence deadlock risk

• In Swing, at least two threads:
  – Main Thread – runs `main(String[] args)`
  – Event Thread – runs ActionListeners and so on
From Graham Hamilton’s blog post “Multithreaded toolkits: A failed dream?”

• “In general, GUI operations start at the top of a stack of library abstractions and go "down". I am operating on an abstract idea in my application that is expressed by some GUI objects, so I start off in my application and call into high-level GUI abstractions, that call into lower level GUI abstractions, that call into the ugly guts of the toolkit, and thence into the OS.

• In contrast, input events start off at the OS layer and are progressively dispatched "up" the abstraction layers, until they arrive in my application code.

• Now, since we are using abstractions, we will naturally be doing locking separately within each abstraction.

• And unfortunately we have the classic lock ordering nightmare: we have two different kinds of activities going on that want to acquire locks in opposite orders. So deadlock is almost inevitable.” (19 October 2004)

https://weblogs.java.net/blog/kgh/archive/2004/10/multithreaded_t.html
Java Swing GUI toolkit dogmas

• Dogma 1: “Time-consuming tasks should not be run on the Event Thread”
  – Otherwise the application becomes unresponsive

• Dogma 2: “Swing components should be accessed on the Event Thread only”
  – The components are not thread-safe

• But if another thread does long-running work, how can it show the results on the GUI?
  – Define the work in SwingWorker subclass instance
  – Use execute() to run it on a worker thread
  – The Event Thread can pick up the results
A short computation on the event thread

- Main thread may create GUI components
  - But should not change eg. background color later
- Event thread calls the ActionListener
  - And can change the background color

```java
final JFrame frame = new JFrame("TestButtonGui");
final JPanel panel = new JPanel();
final JButton button = new JButton("Press here");
frame.add(panel);
panel.add(button);
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        panel.setBackground(new Color(random.nextInt()));
    }
});
frame.pack(); frame.setVisible(true);
```
Main thread and event thread

**Main thread**
- create panel
- create button
- panel.add(button)
- addActionListener()
- repaint()
- repaint()

**Event thread**
- actionPerformed()
- setBackground()
- repaint()
- paint(g)
- paint(g)

Click
Using the main thread for blinking

```java
final JPanel panel = new JPanel() {
    public void paint(Graphics g) {
        super.paint(g);
        if (showBar) {
            g.setColor(Color.RED);
            g.fillRect(0, 0, 10, getHeight());
        }
    }
};
final JButton button = ...
frame.pack(); frame.setVisible(true);
while (true) {
    try { Thread.sleep(800); } // milliseconds
    catch (InterruptedException exn) { }
    showBar = !showBar;
    panel.repaint();
}
```

- `repaint()` may be called by any thread
- Causes event thread to call `paint(g)` later
Fetching webpages on event thread

```java
fetchButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        for (String url : urls) {
            System.out.println("Fetching " + url);
            String page = getPage(url, 200);
            textArea.append(String.format(..., url, page.length()));
        }
    }
});
```

- Occupies event thread for many seconds
  - The GUI is unresponsive in the meantime
  - Results not shown as they become available
    - GUI gets updated only after all fetches
  - Cancellation would not work
    - Cancel button event processed only after all fetches
  - A progress bar would not work
    - Gets updated only after all fetches
static class DownloadWorker extends SwingWorker<String,String> {
    private final TextArea textArea;

    public String doInBackground() {
        StringBuilder sb = new StringBuilder();
        for (String url : urls) {
            String page = getPage(url, 200),
            result = String.format("%-40s%7d\n", url, page.length());
            sb.append(result);
        }
        return sb.toString();
    }

    public void done() {
        try { textArea.append(get()); }
        catch (InterruptedException exn) { }
        catch (ExecutionException exn) { throw new RuntimeExc...; }
    }
}

SwingWorker<T,V> implements Future<T>
.NET has similar System.ComponentModel.BackgroundWorker
Fetching web with SwingWorker

```java
DownloadWorker downloadTask = new DownloadWorker(textArea);
fetchButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        downloadTask.execute();
    }
});
```

- Event thread runs `execute()`
- Worker thread runs `doInBackground()`
  - which returns the full result when computed
- Event thread runs `done()`
  - obtains the already-computed result with `get()`
  - and writes the result to the `textArea`
Worker thread and event thread

Worker thread

downloadTask
downloadTask.execute()
doInBackground()
getPage() "A"
getPage() "B"
getPage() "C"

"A B C"
done()
append("A B C")

event thread
textArea
Fetch

IT University of Copenhagen
Add progress notification

```java
static class DownloadWorker extends SwingWorker<String, String> {
    public String doInBackground() {
        int count = 0;
        StringBuilder sb = new StringBuilder();
        for (String url : urls) {
            String page = getPage(url, 200),
            result = String.format("%-40s%7d\n", url, page.length());
            sb.append(result);
            setProgress((100 * ++count) / urls.length);
        }
        return sb.toString();
    }
}
```

- In the GUI setup, add:
```java
downloadTask.addPropertyChangeListener(new PropertyChangeListener() {
    public void propertyChange(PropertyChangeEvent e) {
        if ("progress".equals(e.getPropertyName())) {
            progressBar.setValue((Integer)e.getNewValue());
        }
    }
});
```
In the GUI setup, add:

```java
cancelButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        downloadTask.cancel(false);
    }
});
```

`On event thread`

```java
static class DownloadWorker extends SwingWorker<String, String> {
    public String doInBackground() {
        for (String url : urls) {
            if (isCancelled())
                break;
            ...
            sb.append(result);
        }
        return sb.toString();
    }
}
```

`On worker thread`

```java
public void done() {
    try { textArea.append(get()); }
    catch (InterruptedException exn) { }
    catch (ExecutionException exn) { throw new RuntimeException(...); }
    catch (CancellationException exn) { textArea.append("Yrk"); }
}
```
Progress and cancellation

Worker thread

downloadTask

doInBackground()

getPage() "A"
setProgress(...)  
getPage() "B"
setProgress(...)

"A B"

textArea

done()

append("Yrk")

event thread

downloadTask.execute()

Fetch

cancel(false)

progressBar.setValue(...)

Canc

IT University of Copenhagen
Worker thread calls **`publish(...)`** a few times.

Event thread calls **`process`** with results from the calls to **`publish`** since last call to **`process`**.
SwingUtilities static methods

• May be called from any thread:
  – boolean isEventDispatchThread()
    • True if executing thread is the Event Thread
  – void invokeLater(Runnable cmd)
    • Execute cmd.run() asynchronously on the Event Thread
  – void invokeAndWait(Runnable command)
    • Execute cmd.run() on the Event Thread, wait to complete

• SwingWorker = these + Java executors
  – Goetz Listings 9.2 and 9.7 indicate how

• Other methods that any thread may call:
  – adding and removing listeners on components
    • but the listeners are called only on the Event Thread
  – comp.repaint() and comp.revalidate()
Avoids interaction with a partially constructed GUI
- because the Event Thread is busy constructing the GUI
Plan for today

• Graphical user interface toolkits, eg Swing
  – not thread-safe, access from event thread only

• Using SwingWorker for long-running work
  – Progress bar
  – Cancellation
  – Display results as they are generated

• A thread-based lift simulator with GUI

• Atomic long with ”thread striping” (week 7)

• Shared mutable data on multicore is slow
Example: 2 lifts, 7 floors, 26 buttons

Concrecy: 2 lift cages move 26 buttons pressed

Two lift threads + the event thread
Modeling and visualizing the lifts

• Use event thread for button clicks (obviously)
  – Inside requests: *this lift* must go to floor n
  – Outside requests: *some lift* must go to floor n, and then up (or down)

• An object for each lift
  – to hold current floor, and floors yet to be visited
  – to compute time to serve an outside request

• A thread for each lift
  – to update its state 16 times a second
  – to cause the GUI to display it

• A controller object
  – to decide which lift should serve an outside request
The lift simulator GUI

- Lift 1 shaft
- Buttons inside lift 1
- Lift 1 cage
- Buttons outside lifts
Lift controller algorithm

- When outside button Up on floor n is pressed
  - Ask each lift how long it would take to get to floor n while continuing up afterwards
  - Then order the fastest lift to serve floor n

```java
class LiftController {
    private final Lift[] lifts;
    ...
    public void someLiftTo(int floor, Direction dir) {
        double bestTime = Double.POSITIVE_INFINITY;
        int bestLift = -1;
        for (int i=0; i<lifts.length; i++) {
            double thisLiftTime = lifts[i].timeToServe(floor, dir);
            if (thisLiftTime < bestTime) {
                bestTime = thisLiftTime;
                bestLift = i;
            }
        }
        lifts[bestLift].customerAt(floor, dir);
    }
}
```
The state of a lift

- Current floor and direction (None, Up, Down)
- required stops and directions, \texttt{stops[n]}:
  - \texttt{null}: no need to stop at floor \texttt{n}
  - \texttt{None}: stop at floor \texttt{n}, don’t know future direction
  - \texttt{Down}: stop at floor \texttt{n}, then continue down
  - \texttt{Up}: stop at floor \texttt{n}, then continue up
  - \texttt{Both}: stop, then up, and later down; or vice versa

\begin{lstlisting}[language=Java]
class Lift implements Runnable {
  private double floor;
  private Direction direction; // None, Up, Down
  // @GuardedBy("this")
  private final Direction[] stops;
  ...
  public synchronized void customerAt(int floor, Direction thenDir) {
    setStop(floor, thenDir.add(getStop(floor)));
  }
}
\end{lstlisting}
The lift’s behavior when going Up

- If at a floor, check whether to stop here
  - If so, open+close doors and clear from stops table
- If not yet at highest requested stop
  - move up a bit and refresh display
  - otherwise stop moving

```
switch (direction) {
  case Up:
    if ((int)floor == floor) { // At a floor, maybe stop here
      Direction afterStop = getStop((int)floor);
      if (afterStop != null && (afterStop != Down || (int)floor == highestStop())) {
        openAndCloseDoors();
        subtractFromStop((int)floor, direction);
      }
    }
    if (floor < highestStop()) {
      floor += direction.delta / steps;
      shaft.moveTo(floor, 0.0);
    } else
      direction = Direction.None;
  break;
  case Down: ... dual to Up ...
  case None: ... if any stops[floor] != null, start moving in that direction ...
}
```
Lift GUI thread safety

• Dogma 1, no long-running on event thread:
  – `sleep()` happens on lift threads, not event thread

• Dogma 2, only event thread works on GUI:
  – Lift thread calls `shaft.moveTo`,
  – which calls `repaint()`,
  – so event thread later calls `paint(g)`, OK

• Lift and event threads access `stops[]` array
  – guarded by lock on lift instance `this`

• Only lift thread accesses `floor` and `direction`
  – not guarded by a lock
Lift modeling reflection

• Seems reasonable to have a thread per lift
  – because they move concurrently

• Why not a thread for the controller?
  – because activated only by the external buttons
  – but what about supervising the lifts, timeouts?
    E.g. if the lift sent to floor 4 going Up gets stuck at floor 3 by some fool blocking the open door?

• In Erlang, with message-passing, use
  – a “process” (task) for each lift
  – a “process” (task) for each floor, a “local controller”
  – no central controller

• Also Akka library, week 12-13

This week

• Reading this week
  – Goetz et al chapter 9
  – McKenney: *Memory barriers*, chapters 1-4

• Exercises
  – You can write responsive and correct user interfaces involving concurrency

• Read before next week’s lecture
  – Goetz chapter 12
  – Herlihy and Shavit chapter 3