Principles of Software Construction: Objects, Design, and Concurrency

Concurrency: More Design Tradeoffs

Christian Kästner Bogdan Vasilescu

School of Computer Science

institute for SOFTWARE RESEARCH

So far on concurrency

- Primitives (synchronized, wait/notify, …)
- Safety
  - immutable vs thread-local vs synchronized
  - fine-grained vs coarse-grained, safe vs guarded
  - documentation
- Concurrent libraries
- Structuring applications
  - Producer-Consumer, Fork-Join, Membrane, Thread Pool
  - Executor Service framework

Intro to Java

Git, CI

UML

Static Analysis

Performance

GUIs

GUIs

More Git

Design

Part 1: Design at a Class Level	Part 2: Designing (Sub)systems	Part 3: Designing Concurrent Systems
Design for Change: Information Hiding, Contracts, Design Patterns, Unit Testing Design for Reuse: Inheritance, Delegation, Immutability, LSP, Design Patterns	Understanding the Problem Responsibility Assignment, Design Patterns, GUI vs Core, Design Case Studies Testing Subsystems Design for Reuse at Scale: Frameworks and APIs	Concurrency Primitives, Synchronization Designing Abstractions for Concurrency Distributed Systems in a Nutshell

Learning Goals

- Apply executor services for parallelizing a task
- Understand real-world tradeoffs of concurrent computing
- Understand the abstractions of the Actor model to concurrency and its tradeoffs
- Pick the right abstractions for the task at hand

PUTTING THE PIECES TOGETHER: PARALLELIZING PREFIXSUM

Sorting Competition

Open to all Carnegie Mellon students.

The goal of this competition is to help develop the fastest parallel algorithms/implementations of sorting, especially for memory managed (garbage collected) languages, and to better understand how languages compare. Deadline for submissions is **May 4, 2017** (11:59pm) EST.

The Prizes:

MAIN PRIZE: The fastest code on a garbage-collected language (see below) will receive a new 13-inch MacBook pro.

LANGUAGE PRIZES: For each of the languages C/C++, Rust, Java, OCaml, Haskell, Go, Swift, Erlang, Scala, Python, Clojure, the fastest submission will win a $100 gift certificate to a local restaurant of your choice. However, the submission has to be at least 10x faster than the (best) sequential sorting library available in the language, and within a factor of 50 of the winner of the main prize. More languages can be considered by suggestion.

ELEGANCE PRIZE: The most elegant solution that is within a factor of 10 of our SML solution will receive a $100 gift certificate to the restaurant of your choice. This will be judged by a small committee, and will be based on how concise and clean the algorithm implementation is in the language of choice.

And you can always add these prizes to your resume.

The General Rules:

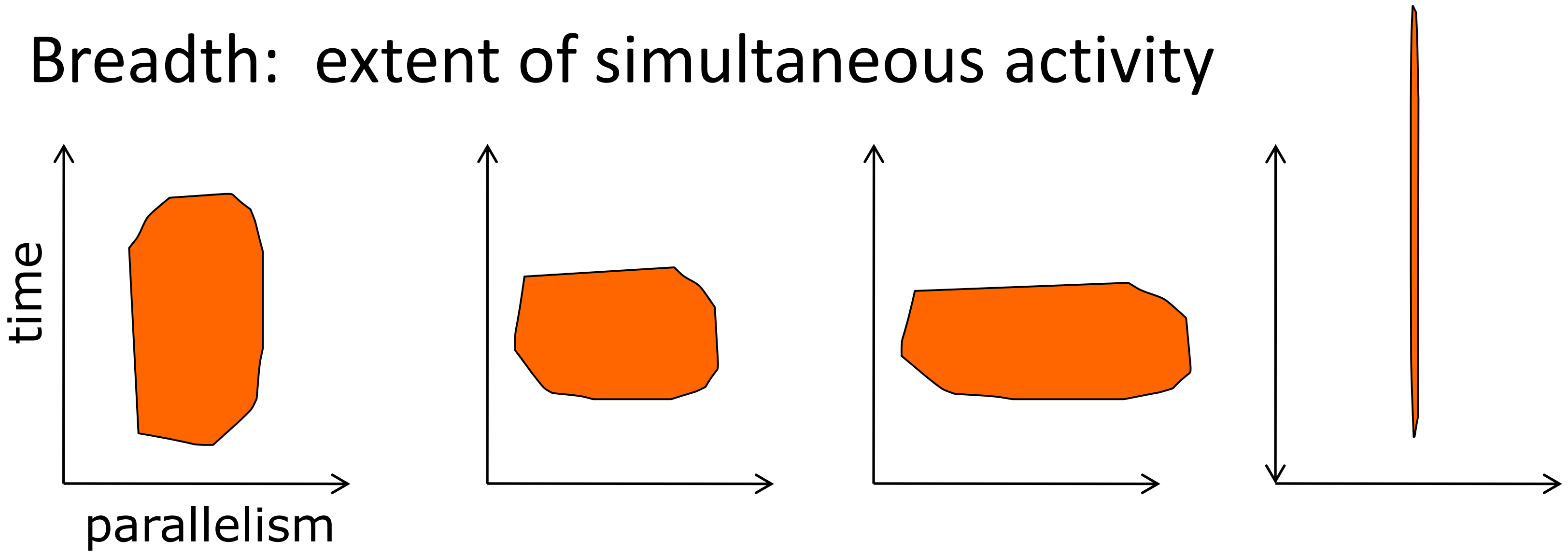
For the main prize you can use any language which garbage collects by default (except for SML, since we give the code for this). Examples of languages that count are: Java, Haskell, Go, Swift, Scala, and OCaml. Examples that do not are Fortran, C, C++ (and variants), and rust. Note that the participants can use non-garbage-collected languages for the language prizes.

You are free to use any standard libraries for the language. This includes, for example, pthreads, openMP, Cilk (in C++), Fork/Join (in Java) for parallelism.

You can use unsafe extensions as long as they are part of the standard libraries.

Vocabulary

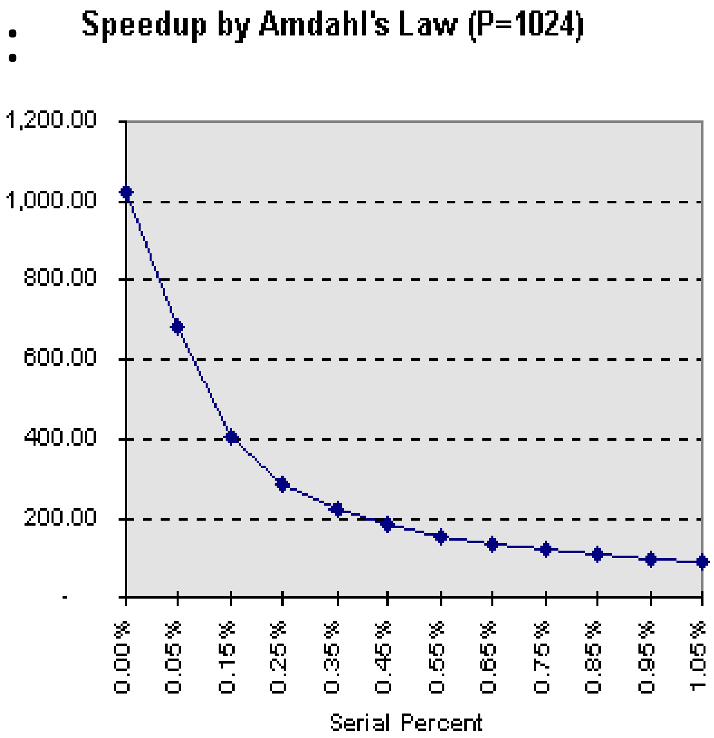
- Work: Total number of computation steps
- Depth: Longest sequence of sequential computation steps
- Breadth: extent of simultaneous activity



What are the typical goals in parallel algorithm design?

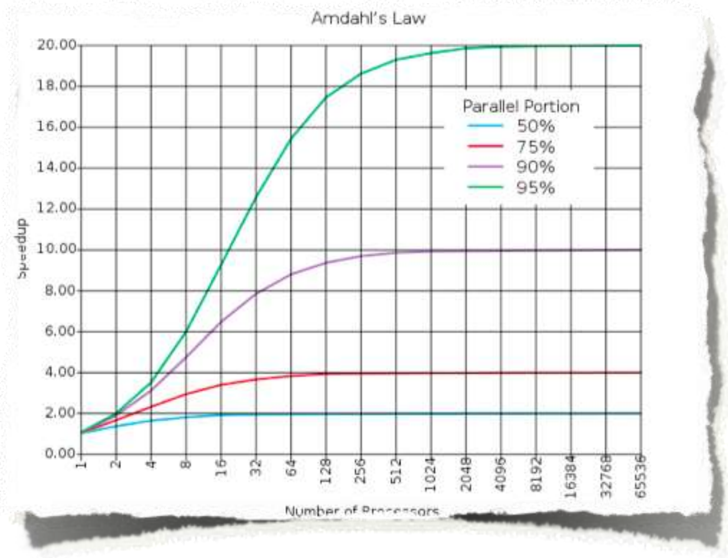
Amdahl's law: How good can the depth get?

- Ideal parallelism with N processors:
  – Speedup = N
- In reality, some work is always inherently sequential
  – Let F be the portion of the total task time that is inherently sequential
  – Speedup = $$\frac{1}{F + (1 - F)/N}$$
  – Suppose F = 10%. What is the max speedup? (you choose N)
    - As N approaches ∞, $1/(0.1 + 0.9/\mathrm{N})$ approaches 10.



Using Amdahl's law as a design guide

- For a given algorithm, suppose
  - `N` processors
  - Problem size `M`
  - Sequential portion `F`



- An obvious question:
  - What happens to speedup as `N` scales?
- A less obvious, important question:
  - What happens to `F` as problem size `M` scales?

"For the past 30 years, computer performance has been driven by Moore's Law; from now on, it will be driven by Amdahl's Law."

— Doron Rajwan, Intel Corp

Concurrency at the language level

- Consider:

```
Collection<Integer> collection = …;
int sum = 0;
for (int i : collection) {
   sum += i;
}
```

- In python:

```
collection = …
sum = 0
for item in collection:
   sum += item
```

Parallel quicksort in Nesl

```
function quicksort(a) =
  if (#a < 2) then a
  else
   let pivot   = a[#a/2];
       lesser  = {e in a| e < pivot};
       equal   = {e in a| e == pivot};
       greater = {e in a| e > pivot};
       result  = {quicksort(v): v in [lesser,greater]};
   in result[0] ++ equal ++ result[1];
```

- Operations in {} occur in parallel
- 210-esque questions: What is total work? What is depth?

Prefix sums (a.k.a. inclusive scan, a.k.a. scan)

- Goal: given array x[0…n-1], compute array of the sum of each prefix of x

  [sum(x[0…0]),
    sum(x[0…1]),
    sum(x[0…2]),
    …
    sum(x[0…n-1])]

- e.g., x = [13, 9, -4, 19, -6, 2, 6, 3]
- prefix sums: [13, 22, 18, 37, 31, 33, 39, 42]

Parallel prefix sums

- Intuition: If we have already computed the partial sums sum(x[0…3]) and sum(x[4…7]), then we can easily compute sum(x[0…7])
- e.g., x = [13, 9, -4, 19, -6, 2, 6, 3]

Parallel prefix sums algorithm, upsweep

- Compute the partial sums in a more useful manner
- [13, 9, -4, 19, -6, 2, 6, 3]
- [13, 22, -4, 15, -6, -4, 6, 9]

Parallel prefix sums algorithm, upsweep

- Compute the partial sums in a more useful manner
- [13, 9, -4, 19, -6, 2, 6, 3]
- [13, 22, -4, 15, -6, -4, 6, 9]
- [13, 22, -4, 37, -6, -4, 6, 5]

Parallel prefix sums algorithm, upsweep

- Compute the partial sums in a more useful manner
- [13, 9, -4, 19, -6, 2, 6, 3]
- [13, 22, -4, 15, -6, -4, 6, 9]
- [13, 22, -4, 37, -6, -4, 6, 5]
- [13, 22, -4, 37, -6, -4, 6, 42]

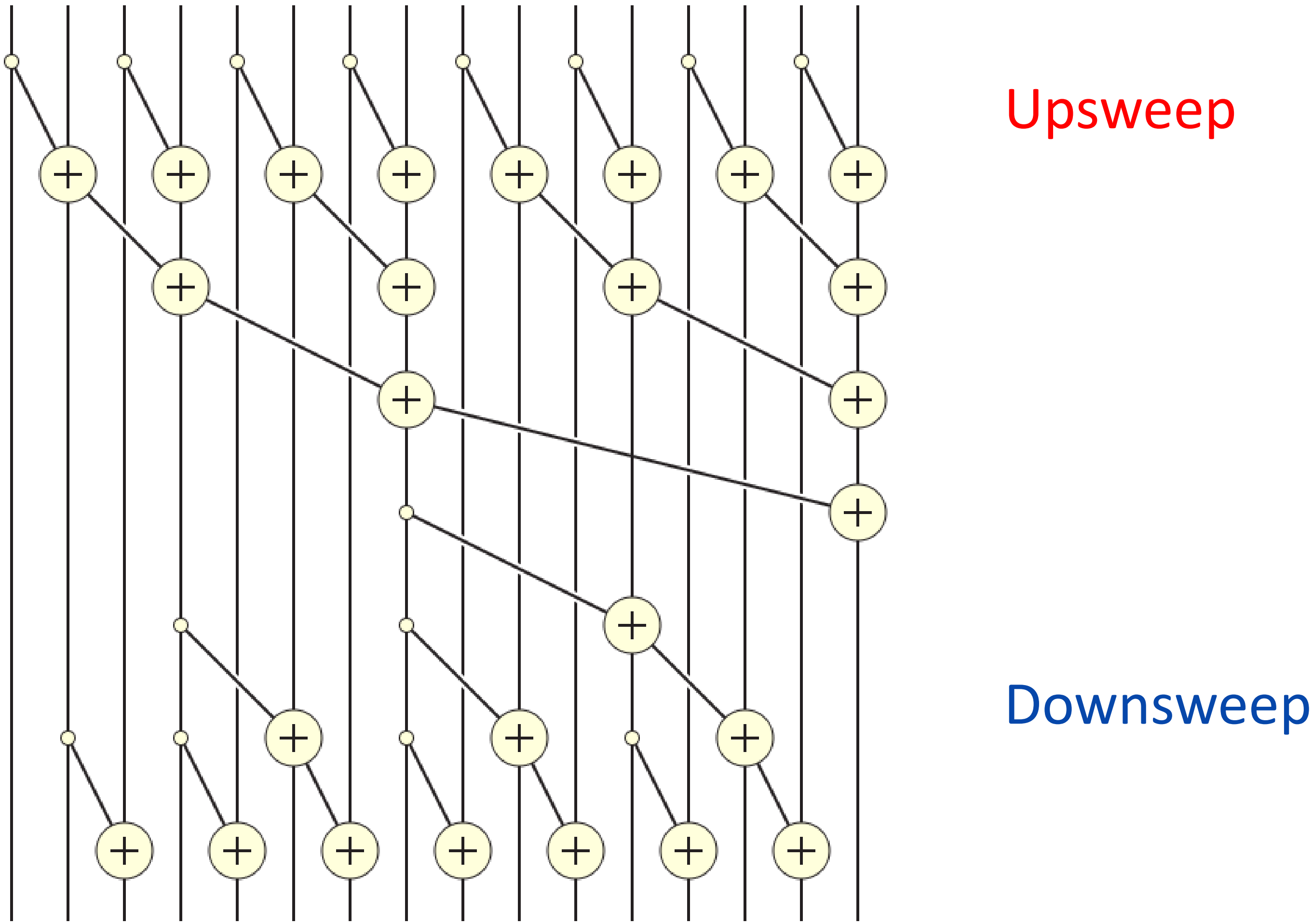
Parallel prefix sums algorithm, downsweep

- Now unwind to calculate the other sums
- [13, 22, -4, 37, -6, -4, 6, 42]
- [13, 22, -4, 37, -6, 33, 6, 42]

Parallel prefix sums algorithm, downsweep

- Now unwinds to calculate the other sums
- [13, 22, -4, 37, -6, -4, 6, 42]
- [13, 22, -4, 37, -6, 33, 6, 42]
- [13, 22, 18, 37, 31, 33, 39, 42]
- Recall, we started with:
  [13, 9, -4, 19, -6, 2, 6, 3]

Doubling array size adds two more levels



Parallel prefix sums

```
pseudocode:

// Upsweep
prefix_sums(x):
  for d in 0 to (lg n)-1:        // d is depth
    parallelfor i in 2d-1 to n-1, by 2d+1:
      x[i+2d] = x[i] + x[i+2d]

// Downsweep
for d in (lg n)-1 to 0:
  parallelfor i in 2d-1 to n-1-2d, by 2d+1:
    if (i-2d >= 0):
      x[i] = x[i] + x[i-2d]
```

Parallel prefix sums algorithm, in code

- An iterative Java-esque implementation:

```
void iterativePrefixSums(long[] a) {
  int gap = 1;
  for ( ; gap < a.length; gap *= 2) {
    parfor(int i=gap-1; i+gap < a.length; i += 2*gap) {
      a[i+gap] = a[i] + a[i+gap];
    }
  }
  for ( ; gap > 0; gap /= 2) {
    parfor(int i=gap-1; i < a.length; i += 2*gap) {
      a[i] = a[i] + ((i-gap >= 0) ? a[i-gap] : 0);
    }
  }
```

Parallel prefix sums algorithm, in code

- A recursive Java-esque implementation:

```
void recursivePrefixSums(long[] a, int gap) {
  if (2*gap – 1 >= a.length) {
    return;
  }

  parfor(int i=gap-1; i+gap < a.length; i += 2*gap) {
    a[i+gap] = a[i] + a[i+gap];
  }

  recursivePrefixSums(a, gap*2);

  parfor(int i=gap-1; i < a.length; i += 2*gap) {
    a[i] = a[i] + ((i-gap >= 0) ? a[i-gap] : 0);
  }
}
```

Parallel prefix sums algorithm

- How good is this?

Parallel prefix sums algorithm

- How good is this?
  - Work: O(n)
  - Depth: O(lg n)
- See PrefixSums.java, PrefixSumsSequentialWithParallelWork.java

Goal: parallelize the PrefixSums implementation

- Specifically, parallelize the parallelizable loops

```
parfor(int i = gap-1;  i+gap < a.length;  i += 2*gap) {
  a[i+gap] = a[i] + a[i+gap];
}
```

- Partition into multiple segments, run in different threads

```
for(int i = left+gap-1;  i+gap < right;  i += 2*gap) {
  a[i+gap] = a[i] + a[i+gap];
}
```

Recall the Java primitive concurrency tools

- The java.lang.Runnable interface
  - void run();
- The java.lang.Thread class
  - Thread(Runnable r);
  - void start();
  - static void sleep(long millis);
  - void join();
  - boolean isAlive();
  - static Thread currentThread();

Recall the Java primitive concurrency tools

- The java.lang.Runnable interface
  - void run();
- The java.lang.Thread class
  - Thread(Runnable r);
  - void start();
  - static void sleep(long millis);
  - void join();
  - boolean isAlive();
  - static Thread currentThread();
- The java.util.concurrent.Callable<V> interface
  - Like java.lang.Runnable but can return a value
  - V call();

A framework for asynchronous computation

- The java.util.concurrent.Future<V> interface
  - V get();
  - V get(long timeout, TimeUnit unit);
  - boolean isDone();
  - boolean cancel(boolean mayInterruptIfRunning);
  - boolean isCancelled();

A framework for asynchronous computation

- The java.util.concurrent.Future<V> interface:
  - V get();
  - V get(long timeout, TimeUnit unit);
  - boolean isDone();
  - boolean cancel(boolean mayInterruptIfRunning);
  - boolean isCancelled();
- The java.util.concurrent.ExecutorService interface:
  - Future<?> submit(Runnable task);
  - Future<V> submit(Callable<V> task);
  - List<Future<V>>
  - invokeAll(Collection<? extends Callable<V>> tasks);
  - Future<V>
  - invokeAny(Collection<? extends Callable<V>> tasks);
  - void shutdown();

Executors for common computational patterns

- From the java.util.concurrent.Executors class
  - static ExecutorService newSingleThreadExecutor();
  - static ExecutorService newFixedThreadPool(int n);
  - static ExecutorService newCachedThreadPool();
  - static ExecutorService newScheduledThreadPool(int n);

Fork/Join: another common computational pattern

- In a long computation:
  - Fork a thread (or more) to do some work
  - Join the thread(s) to obtain the result of the work

Fork/Join: another common computational pattern

- In a long computation:
  - Fork a thread (or more) to do some work
  - Join the thread(s) to obtain the result of the work
- The java.util.concurrent.ForkJoinPool class
  - Implements ExecutorService
  - Executes java.util.concurrent.ForkJoinTask<V> or

    java.util.concurrent.RecursiveTask<V> or
    java.util.concurrent.RecursiveAction

The RecursiveAction abstract class

- public class MyActionFoo extends RecursiveAction {
-    public MyActionFoo(…) {
-       store the data fields we need
-    }

-    @Override
-    public void compute() {
-       if (the task is small) {
-          do the work here;
-          return;
-       }

-       invokeAll(new MyActionFoo(…), // smaller
-               new MyActionFoo(…), // tasks
-               …);   // …
-    }
- }

A ForkJoin example

- See PrefixSumsParallelForkJoin.java
- See the processor go, go go!

Parallel prefix sums algorithm

- How good is this?
  - Work: O(n)
  - Depth: O(lg n)
- See PrefixSumsParallelArrays.java

Parallel prefix sums algorithm

- How good is this?
  - Work: O(n)
  - Depth: O(lg n)
- See PrefixSumsParallelArrays.java
- See PrefixSumsSequential.java

Parallel prefix sums algorithm

- How good is this?
  - Work: O(n)
  - Depth: O(lg n)
- See PrefixSumsParallelArrays.java
- See PrefixSumsSequential.java
  - n-1 additions
  - Memory access is sequential
- For PrefixSumsSequentialWithParallelWork.java
  - About 2n useful additions, plus extra additions for the loop indexes
  - Memory access is non-sequential
- The punchline:
  - Don't roll your own
  - Cache and constants matter

THE ACTOR MODEL

Concurrency Challenges

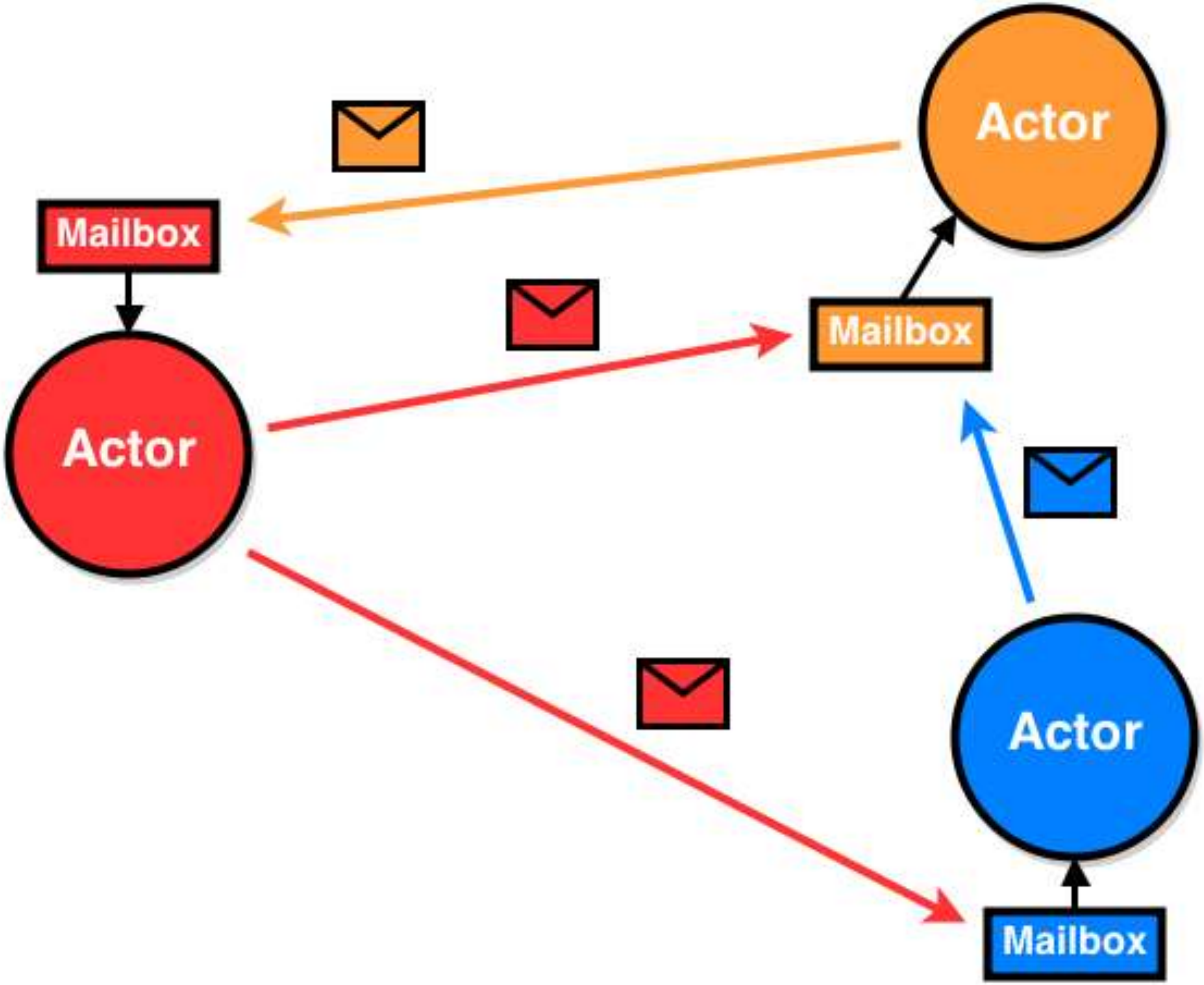
- Java's shared-memory model "is unnatural for developers", error-prone & unscalable
- Requires careful tracking of how state is shared and how it is synchronized
- Faults difficult to detect

The Actor Model

- Actors: independent concurrent entities, communicating over asynchronous message passing
- Local mutable state; no synchronization required (think thread-confined state)
- No shared state (“shared nothing”, think processes, not threads)
- Message queue for incoming messages; one message processed at a time
- Messages contain only pure values, no references (“call by value”)

Actors – Mental Model

- People communicating over email
- May send email; reply later by another email
- Read and process one email at a time without multitasking
- People have memory but cannot access each other's memories


Actor
Mailbox
Mailbox
Actor
Actor
Mailbox

Akka Example

```
public class MyActor extends AbstractActor {
  private final LoggingAdapter log =
          Logging.getLogger(context().system(), this);

  public MyActor() {
    receive(ReceiveBuilder.
      match(String.class, s -> {
        log.info("Received String message: {}", s);
      }).
      matchAny(o -> log.info("received unknown message")).build()
    );
  }
}
```

Akka Example

```
public class DemoMessagesActor extends AbstractLoggingActor {
   static public class Greeting {
    private final String from;
    public Greeting(String from) {     this.from = from;    }
    public String getGreeter() {     return from;    }
  }
  DemoMessagesActor() {
    receive(ReceiveBuilder.
      match(Greeting.class, g -> {
        log().info("I was greeted by {}", g.getGreeter());
      }).build()
    );
  };
}
```

Akka sending messages

- tell(): "fire-and-forget", e.g. send a message asynchronously and return immediately.
- ask(): sends a message asynchronously and returns a Future representing a possible reply.

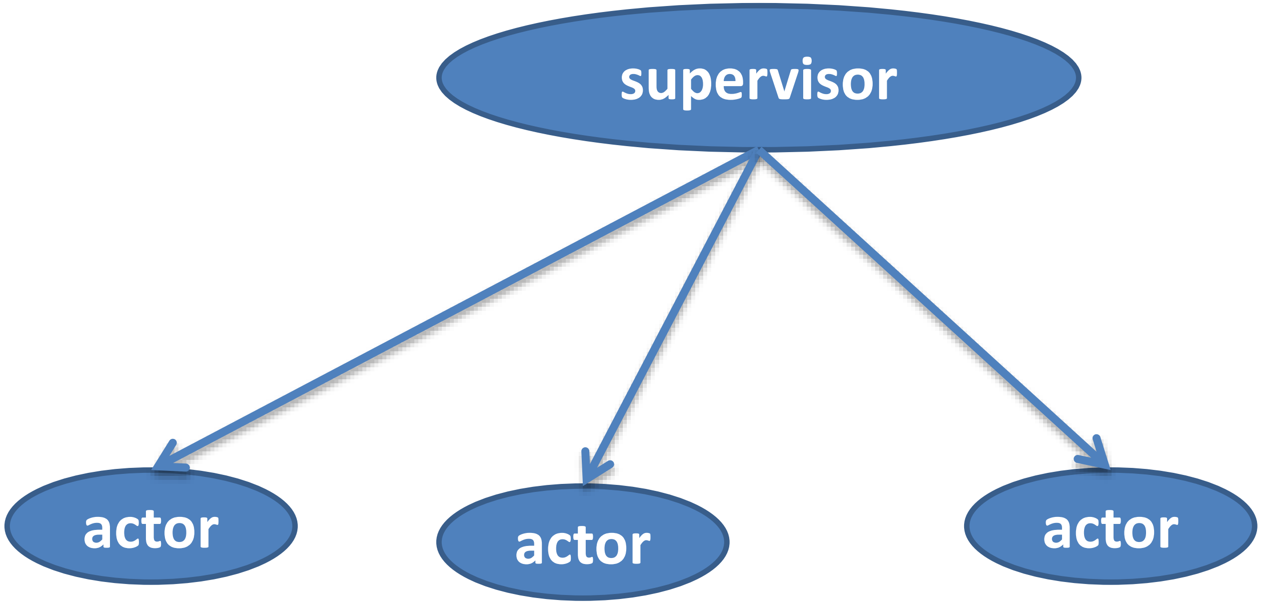
```
try {
  String result = operation();
  sender().tell(result, self());
} catch (Exception e) {
  sender().tell(new akka.actor.Status.Failure(e), self());
  throw e;
}
```

Actor Frameworks

- Actors are simple, but frameworks can provide much shared functionality
- Transparent distributed computations on one or multiple machines
- Fair scheduling, error recovery, load balancing and scaling (horizontal and vertical)
- Abstractions for common tasks, synchronous messages

Fault Tolerance with Actors

- “let it crash” philosophy



Actor model – Properties

- Encapsulation
- Fair scheduling
- Location transparency
- Locality of reference
- Transparent migration

- Failure isolation

Build your own actor framework

- One thread per actor
- Producer-consumer pattern with queue for incoming messages
- Thread-local state for each actor
- Actors communicate only with queues not with other actors
- Only add immutable objects as parameters to messages or use defensive copying
- Write own request-reply mechanism (include unique ID in message and reply, wrap in future or use wait/notify)

As usual: don't build your own, but use one built by experts

Actors in Java

- Many frameworks
- No built-in language support
- State encapsulation, call-by-value etc often just conventions, not enforced (easy to cheat, easy to make mistakes)

	SALSA (v1.1.2)	Scala Actors (v2.7.3)	Kilim (v0.6)	Actor Architecture (v0.1.3)	JavAct (v1.5.3)	ActorFoundry (v1.0)	Jetlang (v0.1.7)
State Encapsulation	Yes	No	No	Yes	Yes	Yes	Yes
Safe Message-passing	Yes	No	No	Yes	No	Yes	No
Fair Scheduling	Yes	Yes	No	Yes	No	Yes	No
Location Transparency	Yes	No	No	Yes	Yes	Yes	Yes
Mobility	Yes	No	No	Yes	Yes	Yes	No

Design Discussion

- Shared memory (Java default) vs shared nothing (Actors)
  - performance vs simplicity, robustness
  - some functionality delegated to frameworks (eg., akka, executor services)
- Gateway to distributed systems and remote procedure calls
  - Includes distributed system problems (lost messages, timing issues, …)

DESIGN DISCUSSIONS:
WHY ARE GUIS SINGLE THREADED?

Concurrency and GUIs

- Earliest GUIs were programmed from main thread
- Modern GUI frameworks have single *event dispatch thread*

- Why not more concurrency, e.g., thread per button event?

Coarse-grained concurrency in GUIs

- Fine-grained concurrency is difficult to get right. Lots of deadlock and race problems in attempts
- Difficult to decide what to lock and when
- Environment-driven events (e.g., button clicked) and application-driven actions (update screen with computation results) interact at runtime
- Model-view-controller: Who interacts with whom and who holds the locks?
- Framework would expose all locking details to users

Single-threaded GUI

- Assumption: Thread confinement
- No locking required, since all access from *event dispatch thread*
- Pushes burden to developers to perform all actions on that event thread (i.e., GUI objects are confined to event dispatch thread)

Summary

- Apply concurrent design patterns for parallelizing computations
- Be mindful of real overhead of concurrent computations
- The Actor Model provides an alternative approach to Java’s shared memory model
  - Many tradeoffs, including performance and simplicity and robustness
  - Mostly retrofitted on JVM

Recommended Readings

- Goetz et al. Java Concurrency In Practice. Pearson Education, 2006
- Karmani, Rajesh K., Amin Shali, and Gul Agha. "Actor frameworks for the JVM platform: a comparative analysis." Proceedings of the 7th International Conference on Principles and Practice of Programming in Java. ACM, 2009.