Principles of Software Construction: Objects, Design and Concurrency

The Perils of Concurrency, part 4

Can't live with it.
Can't live without it.

15-214
toad

Fall 2012

Jonathan Aldrich **Charlie Garrod**

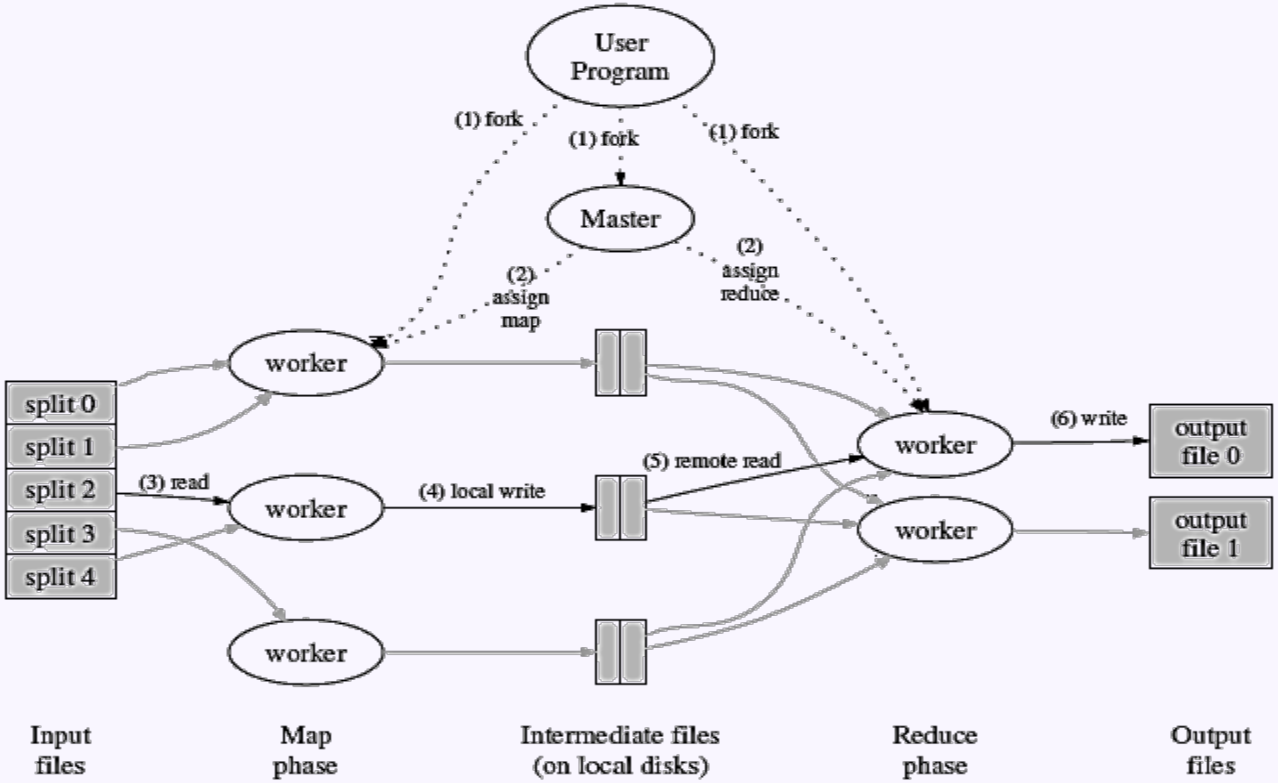
School of Computer Science
institute for SOFTWARE RESEARCH

Administrivia

- Problems with your Homework 6 partner?
  - Email me and/or Jonathan
- Homework 6c and Lab 7 due tonight
  - Using a late day allows you to turn in the second part of hw6c late, and also lab 7 late
- Homework 7 coming soon

Last time: parallel data algorithms and frameworks

- Quicksort
- Prefix sums
- Map / Reduce



Today

- The Java standard library concurrency framework

A better parallel prefix sums algorithm, part 1

- Computes the partial sums in a more useful manner

```
[13,    9,   -4,   19,   -6,    2,    6,    3]

[13,   22,   -4,   15,   -6,   -4,    6,    9]
```

A better parallel prefix sums algorithm, part 1

- Computes 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]
```

A better parallel prefix sums algorithm, part 1

- Computes 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]
```

⋮

A better parallel prefix sums algorithm, part 2

- Now unwinds to calculate the other sums

```
[13,   22,   -4,   37,   -6,   -4,    6,   42]

[13,   22,   -4,   37,   -6,   33,    6,   42]
```

A better parallel prefix sums algorithm, part 2

- 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]
```

A better parallel prefix sums algorithm, in code

- An iterative implementation:

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

A better parallel prefix sums algorithm, in code

- A recursive implementation:

```
void computePrefixSumsRecursive(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];
  }

  computePrefixSumsRecursive(a, gap*2);

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

A better parallel prefix sums algorithm

- How good is this?
  - O(n) work, O(lg n) depth

A better parallel prefix sums algorithm

- How good is this?
  - O(n) work, O(lg n) depth
  - About 2n useful additions, plus extra additions for the loop indexes
  - Converts sequential memory access into non-sequential memory access
- See PrefixSums.java, PrefixSumsSequentialImpl.java, PrefixSumsNonSequentialImpl.java, and Main.java

Goal: parallelize PrefixSumsNonSequentialImpl

- 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];
}
```

Goal: parallelize PrefixSumsNonSequentialImpl

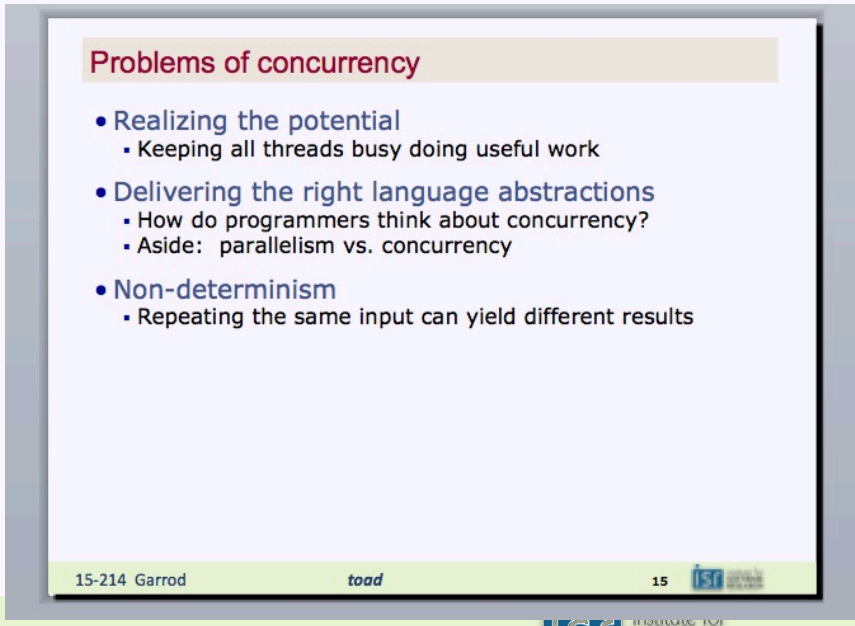
- Specifically, parallelize the parallelizable loops

```
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];
}
```

- Caveats:
  - We know we can't beat the sequential implementation on my 4-core computer



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();
```

- The `java.util.concurrent.ExecutorService` interface

```
Future          submit(Runnable task);
Future<V>       submit(Callable<V> task);
List<Future<V>> invokeAll(Collection<Callable<V>>
                                          tasks);

Future<V>       invokeAny(Collection<Callable<V>>
                                          tasks);
```

`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);
```

- See NetworkServer.java

Fork/Join: another common computational pattern

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

Fork/Join: another common computational pattern

- In a long computation:
  - Fork one (or more) thread(s) 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 RecursiveActionImpl
                                  extends RecursiveAction {
    public RecursiveActionImpl(…) {
      store the data fields we need
    }

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

      invokeAll(new RecursiveActionImpl(…),   // smaller
                new RecursiveActionImpl(…),   // tasks
                …);                           // …
    }
}
```

A ForkJoin example

- See PrefixSumsParallelImpl.java, PrefixSumsParallelLoop1.java, and PrefixSumsParallelLoop2.java
- See the processor go, go go!

Conclusions?

- Concurrency is powerful, dangerous, necessary
  - Necessary: Moore's law future
  - Powerful: from $O(n \log n)$ to $O(\log n)$
  - Dangerous: races, deadlocks, vulnerabilities
    - Intermittent errors, difficult to replicate and diagnose
    - Difficulties in development and for test and inspection

- There are many candidates for concurrency abstractions
  - Intrinsic concurrency = parallelism
    - Parallel loops. Map/reduce.
  - Explicit concurrency
    - Locks. Thread policies.
  - New abstractions continue to emerge
    - Atomic objects, transactional memory, many others

- Trade-offs in algorithm design and implementation
  - Generality vs. ease of use, predictability
  - Degree of concurrency vs. complexity