What else looks like this?

- Saw summing an array went from $O(n)$ sequential to $O(\log n)$ parallel (assuming a lot of processors and very large $n$!)
  - An exponential speed-up in theory

- Anything that can use results from two halves and merge them in $O(1)$ time has the same property…
Examples

• Maximum or minimum element

• Is there an element satisfying some property (e.g., is there a 17)?

• Left-most element satisfying some property (e.g., first 17)
  – What should the recursive tasks return?
  – How should we merge the results?

• Corners of a rectangle containing all points (a “bounding box”)

• Counts, for example, number of strings that start with a vowel
  – This is just summing with a different base case
  – Many problems are!
Reductions

- Computations of this form are called reductions (or reduces?)
- They take a set of data items and produce a single result
- Note: Recursive results don’t have to be single numbers or strings. They can be arrays or objects with multiple fields.
  - Example: Histogram of test results
- While many can be parallelized due to nice properties like associativity of addition, some things are inherently sequential
  - How we process arr[i] may depend entirely on the result of processing arr[i-1]
Even easier: Data Parallel (Maps)

- While reductions are a simple pattern of parallel programming, maps are even simpler
  - Operate on set of elements to produce a new set of elements (no combining results)
  - For arrays, this is so trivial some hardware has direct support
- Canonical example: Vector addition

```java
int[] vector_add(int[] arr1, int[] arr2) {
    assert (arr1.length == arr2.length);
    result = new int[arr1.length];
    len = arr.length;
    FORALL (i=0; i < arr.length; i++) {
        result[i] = arr1[i] + arr2[i];
    }
    return result;
}
```
Maps in ForkJoin Framework

```java
class VecAdd extends RecursiveAction {
    int lo; int hi; int[] res; int[] arr1; int[] arr2;
    VecAdd(int l, int h, int[] r, int[] a1, int[] a2) {
    }

    protected void compute() {
        if (hi - lo < SEQUENTIAL_CUTOFF) {
            for (int i = lo; i < hi; i++)
                res[i] = arr1[i] + arr2[i];
        } else {
            int mid = (hi + lo) / 2;
            VecAdd left = new VecAdd(lo, mid, res, arr1, arr2);
            VecAdd right = new VecAdd(mid, hi, res, arr1, arr2);
            left.fork();
            right.compute();
        }
    }
}

static final ForkJoinPool fjPool = new ForkJoinPool();
int[] add(int[] arr1, int[] arr2) {
    assert (arr1.length == arr2.length);
    int[] ans = new int[arr1.length];
    fjPool.invoke(new VecAdd(0, arr.length, ans, arr1, arr2);
    return ans;
}
```
Maps in X10

```scala
def add(arr1: Array[Int], arr2: Array[Int]) = {
  val ans = Array.make[Int](arr1.length);
  finish foreach (i in ans.region) {
    ans(i) = arr1(i) + arr2(i);
  }
  return ans;
}
```

- **X10**
  - new OO programming language from IBM
  - designed for high-performance computing, multicore

- Fine-grained concurrency
- Distribution
  - *Partitioned* global address space
- Atomicity operations
X10

• http://x10-lang.org

• Current version 2.1
• Language still under development

• Compiles to Java
  • slow!

• Compiles to C++ on top of MPI, LAPI, sockets
  • fast!
X10 library

• Features I won’t talk about:
  • constrained types
  • generics
  • arrays and region algebra
X10 design (non-)goals

- Not trying to hide concurrency
  - this never works
- Not trying to hide the multiprocessor
  - this never works either
- Not trying to pretend concurrency is just a minor extension
  - this gives lousy concurrency
Basic X10 concurrency

- Three constructs:
  - `async S` – spawn a new activity
  - `finish S` – wait until all activities in `S` terminate
  - `atomic S` – do `S` without interruption

- `async` and `finish` similar to `spawn` and `sync` in Cilk
Funicular [this name will change!]

- X10 concurrency features as a Scala library
- http://ranger.uta.edu/~nystrom/funicular
- Examples below use Scala syntax
- To use:
  - import funicular._
def add(arr1: Array[Int], arr2: Array[Int]) = {
  val ans = Array.make[Int](arr1.length);
  finish for (i in ans.region)
    async {
      ans(i) = arr1(i) + arr2(i);
    }
  return ans;
}
Async

- `async S` – spawn a new activity
  - much lighter-weight than creating a thread
  - both syntactically and performance-wise
Finish

- **finish S** – wait until all activities spawned in S terminate
  - Also collects any exceptions thrown by spawned activities
Deadlock

• Using `async` and `finish` guarantees no deadlock

  • A parent activity can wait on spawned children
  • But, children can’t wait on parent
  • No wait-cycle => no deadlock
Parallel loops

• X10:

```java
var v: Int = 0
foreach ((i) in 1..100) {
    v += i;
}
```

Each loop iteration runs in parallel.
Loop spawns 100 activities.

• Funicular:

```java
var v = 0
for (i <- 1 to 100 async) {
    v += i
}
```
def add(arr1: Array[Int], arr2: Array[Int]) = {
  val ans = Array.make[Int](arr1.length);
  finish foreach (i in ans.region) {
    ans(i) = arr1(i) + arr2(i);
  }
  return ans;
}
Determinate concurrency

- This is what to aim for
- Order of execution does not change result
- X10 does not guarantee determinacy
  - but can often get it
var v = 0
for (i <- 1 to 100 async) {
    v += i
}

Atomicity fail
Atomicity fail

Race condition: concurrent accesses to v.

```java
var v = 0
for (i <- 1 to 100 async) {
    v += i
}
```
Atomicity good

```javascript
var v = 0
for (i <- 1 to 100 async) {
  atomic {
    v += i
  }
}
```
Use **atomic** to guarantee only one activity at a time will execute code.

```scala
var v = 0
for (i <- 1 to 100 async) {
    atomic {
        v += i
    }
}
```
Atomic vs. synchronized

- Compare:
  ```java
  synchronized (this) {
      v += i
  }
  ```
- vs.
  ```java
  atomic {
      v += i
  }
  ```
- No need to specify which lock to acquire.
Atomicity

- Most operations are not atomic
  - atomicity is expensive
- Use atomic sparingly
Atomic and deadlock

- Can deadlock if code in an **atomic** statement can block
  - X10 prevents this from occurring
  - Cannot perform blocking operations within an **atomic** statement
    - e.g., I/O

- Funicular: no checking that **atomic** statements do not cause deadlock
  - A library function, not a language feature
Atomicity better

```scala
var v = new AtomicInteger
for (i <- 1 to 100 async) {
  v.getAndAdd(i)
}
```

- Use built-in atomic types for faster atomic operations
- Not as “clean” as `atomic`, though.
Array operations

• X10 intended for high-performance computing

• Supports parallel operations on arrays

• Usage in functional programming style

    // parallel fold:
    // add all elements of array A
    val sum = A.reduce(_ + _)

    // parallel map:
    // map all elements to their negation
    val neg = A.lift(0f - _)
Heat transfer

\[
A: \begin{bmatrix}
& & & & \\
& 1 & 1 & 1 & \\
& 1 & 0 & 1 & \\
& 1 & 1 & 1 & \\
& & & & \\
\end{bmatrix}
\]

\[
\sum \begin{bmatrix}
& & & \\
& 1 & 1 & \\
& 1 & 0 & \\
& 1 & 1 & \\
& & & \\
\end{bmatrix} \div 4
\]

repeat until max change < \(\varepsilon\)
Heat transfer

- Heat transfer is an example of a **stencil** computation
  - Performs a simple operation over a small window (the stencil) on an array
  - Easy to parallelize
import funicular._
import scala.util.Random

object Stencil {
  def main (args: Array[String]) = finish {
    // 100 x 100 array of random points
    val rnd = new Random(0)
    val N = 100
    val A = Array.tabulate(N)(
      _ => Array.tabulate(N)(_ => rnd.nextFloat)
    )
    transfer(A)
  }

  def transfer(A: Array[Array[Float]]) = ...
}

Heat transfer
def transfer(A: Array[Array[Float]]) = {
  val N = A.length
  var delta = Float.MaxValue
  var epsilon = 1e-4
  var iters = 0

  do {
    iters += 1
    finish for (i <- 1 until (N-1) async) {
      for (j <- 1 until (N-1) async) {
        val t = ( A(i)(j-1) + A(i)(j+1) + A(i-1)(j) + A(i+1)(j) ) / 4.f
        B(i)(j) = (A(i)(j) - t).abs
        A(i)(j) = t
      }
    }

    val D = B.mapPar[Float]((v: Array[Float]) => v.reduce(_ max _))
    delta = D.reduce((x:Float,y:Float) => x max y)
  } while (delta > epsilon);

  println("converged in " + iters + " iterations")
}
Futures

• Similar to `async`, but need to compute a `result`

• `future[T] e`
  • spawns an activity that computes expression `e`
  • returns a handle of type `Future[T]`
  • can `force` the future to get the result
    • blocks until result is available

• `val f = future[T] e`
• `val x = f.force`
Futures and deadlock

- var f1 = future[Int] f2.force
- var f2 = future[Int] f1.force

- f1.force -> f2.force -> f1.force -> ...
Future example

import funicular._

object Fib {
    if (n <= 2) 1
    else fib(n-1).force + fib(n-2).force
  }

  def main(args: Array[String]) = finish {
    val n = if (args.length > 0) args(0).toInt else 10
    println("fib(" + n + ") = " + fib(n).force)
  }
}
Outline

Now:

- Clever ways to parallelize more than is intuitively possible
  - Parallel prefix:
    • This “key trick” typically underlies surprising parallelization
    • Enables other things like packs
  - Parallel sorting: quicksort (not in place) and mergesort
    • Easy to get a little parallelism
    • With cleverness can get a lot
The prefix-sum problem

Given int[] input, produce int[] output where output[i] is the sum of input[0]+input[1]+...input[i]

Sequential is easy enough for a CS1 exam:

```java
int[] prefix_sum(int[] input){
    int[] output = new int[input.length];
    output[0] = input[0];
    for(int i=1; i < input.length; i++)
        output[i] = output[i-1]+input[i];
    return output;
}
```

This does not appear to be parallelizable -- $O(n)$ work

– This algorithm is sequential, but we can design a different algorithm with parallelism (surprising)
Parallel prefix-sum

The parallel-prefix algorithm has $O(n)$ work but a span of $2\log n$

- So span is $O(\log n)$ and parallelism is $n/\log n$, an exponential speedup just like array summing

- The 2 is because there will be two “passes” one “up” one “down”

- Historical note:
  - Original algorithm due to R. Ladner and M. Fischer at the University of Washington in 1977
Parallel prefix, generalized

Just as sum-array was the simplest example of a pattern that matches many, many problems, so is prefix-sum

- Minimum, maximum of all elements to the left of $i$
- Is there an element to the left of $i$ satisfying some property?
- Count of all elements to the left of $i$ satisfying some property
  - This last one is perfect for an efficient parallel pack…
  - Perfect for building on top of the “parallel prefix trick”
- We did an *inclusive* sum, but *exclusive* is just as easy
Pack (filter)

[Non-standard terminology]

Given an array \textbf{input}, produce an array \textbf{output} containing only elements such that \( f(\text{elt}) \) is \textbf{true}

Example: \textbf{input} \( [17, 4, 6, 8, 11, 5, 13, 19, 0, 24] \)

\( f: \text{ is elt} > 10 \)

\textbf{output} \( [17, 11, 13, 19, 24] \)

Looks hard to parallelize

– Finding elements for the output is easy

– But getting them in the right place is hard
Parallel prefix to the rescue

1. Use a parallel map to compute a bit-vector for true elements
   input  [17, 4, 6, 8, 11, 5, 13, 19, 0, 24]
   bits   [1, 0, 0, 0, 1, 0, 1, 1, 0, 1]

2. Do parallel-prefix sum on the bit-vector
   bitsum [1, 1, 1, 1, 2, 2, 3, 4, 4, 5]

3. Use a parallel map to produce the output
   output [17, 11, 13, 19, 24]
Parallel prefix to the rescue

1. Use a parallel map to compute a bit-vector for true elements
   - input: [17, 4, 6, 8, 11, 5, 13, 19, 0, 24]
   - bits: [1, 0, 0, 0, 1, 0, 1, 1, 0, 1]

2. Do parallel-prefix sum on the bit-vector
   - bitsum: [1, 1, 1, 1, 2, 2, 3, 4, 4, 5]

3. Use a parallel map to produce the output
   - output: [17, 11, 13, 19, 24]

```java
output = new array of size bitsum[n-1]
if(bitsum[0]==1) output[0] = input[0];
FORALL (i=1; i < input.length; i++)
    if(bitsum[i] > bitsum[i-1])
        output[bitsum[i]-1] = input[i];
```
Pack comments

• First two steps can be combined into one pass
  – Just using a different base case for the prefix sum
  – Has no effect on asymptotic complexity

• Parallelized packs will help us parallelize quicksort

• Analysis: $O(n)$ work, $O(\log n)$ span
  – 2 or 3 passes, but 3 is a constant