ORACLE®

ORACLE®

Laying Parallel Tracks

Victor Luchangco
Sun Labs, Oracle

The Future is Parallel

- Multicore chips imply ubiquitous parallelism
  - driven by power, complexity
  - already in desktops/laptops
- Parallel applications needed to exploit new hardware
  - Long studied by academics
  - Specialized expertise for scientific code, databases
  - But now “everyone” must write concurrent programs

Old Wineskins

- Lock-based synchronization
  - Introduces possibility of deadlock
  - Trade-off in granularity, both spatial and temporal
  - Fine-grained locking is error-prone
  - Not composable
- Explicit message passing among fixed number of processors
  - Multicore chips communicate via shared memory
  - Number of processors is not fixed
- Nonblocking techniques for shared-memory programming
  - Difficult to understand and implement
  - Code is fragile
  - Often high overhead
- Functional programming

Communication and Synchronization

- These are important *low-level* concepts
  - Required by library programmers building the “plumbing”
  - . . . but we should discourage their use by application programmers
- Why are transactions better than locks?
  - Avoid deadlock?
  - Optimistic execution is more efficient?
  - Big win: composable abstractions
- Avoid (minimize) need for communication and synchronization
  - Side-effect-free code
  - Immutable data structures
  - Functional programming

Tips for Programming for Parallelism

- Minimize shared mutable state
  - Avoid complicated locking schemes
  - Use transactions to manage access to shared data
- Provide “slack”
  - Identify/enable as much potential parallelism as possible
  - Work-stealing can efficiently exploit such slack
- Enable multiway decomposition and aggregation
  - Use data structures that can be “split”
- Raise level of abstraction
- Languages and tools should help

Tips for Programming for Parallelism

- Minimize shared mutable state
  - Avoid complicated locking schemes
  - Use transactions to manage access to shared data
- Provide “slack”
  - Identify/enable as much potential parallelism as possible
  - Work-stealing can efficiently exploit such slack
- Enable multiway decomposition and aggregation
  - Use data structures that can be “split”
- Raise level of abstraction
- Languages and tools should help
  - The Fortress programming language

Good Sequential Algorithms Lead Us Astray

- They minimize computation by reusing previously computed results
  - But redundant computation may reduce communication
- They minimize space usage by reusing storage
  - But extra storage may permit temporal decoupling
- They stress linear problem decomposition, processing one thing at a time and accumulating results
  - But good parallel code usually requires multiway problem decomposition and multiway aggregation of results

The bag of programming tricks

that has served us well

for the past 50 years

is

the wrong way to think

going forward and

must be thrown out.

A Simple Sequential Computation

$\texttt{var}\ sum : \mathbb{Z}32 := 0$

$\texttt{for}\ i \leftarrow seq(1 : 1\,000\,000)\ \texttt{do}$

$\quad sum \mathrel{+}= i$

$\texttt{end}$

(This is Fortress code.)

A Simple Sequential Computation

$\texttt{var}\ sum : \mathbb{Z}32 := 0$

$\texttt{for}\ i \leftarrow seq(1 : 1\,000\,000)\ \texttt{do}$

$\quad sum \mathrel{+}= i$

$\texttt{end}$

Can we just do the loop iterations in parallel?

A Simple Parallel Computation

$\texttt{var}\ sum : \mathbb{Z}32 := 0$

$\texttt{for}\ i \leftarrow 1 : 1\,000\,000\ \texttt{do}$

$\quad \texttt{atomic}\ sum \mathrel{+}= i$

$\texttt{end}$

Is this better?

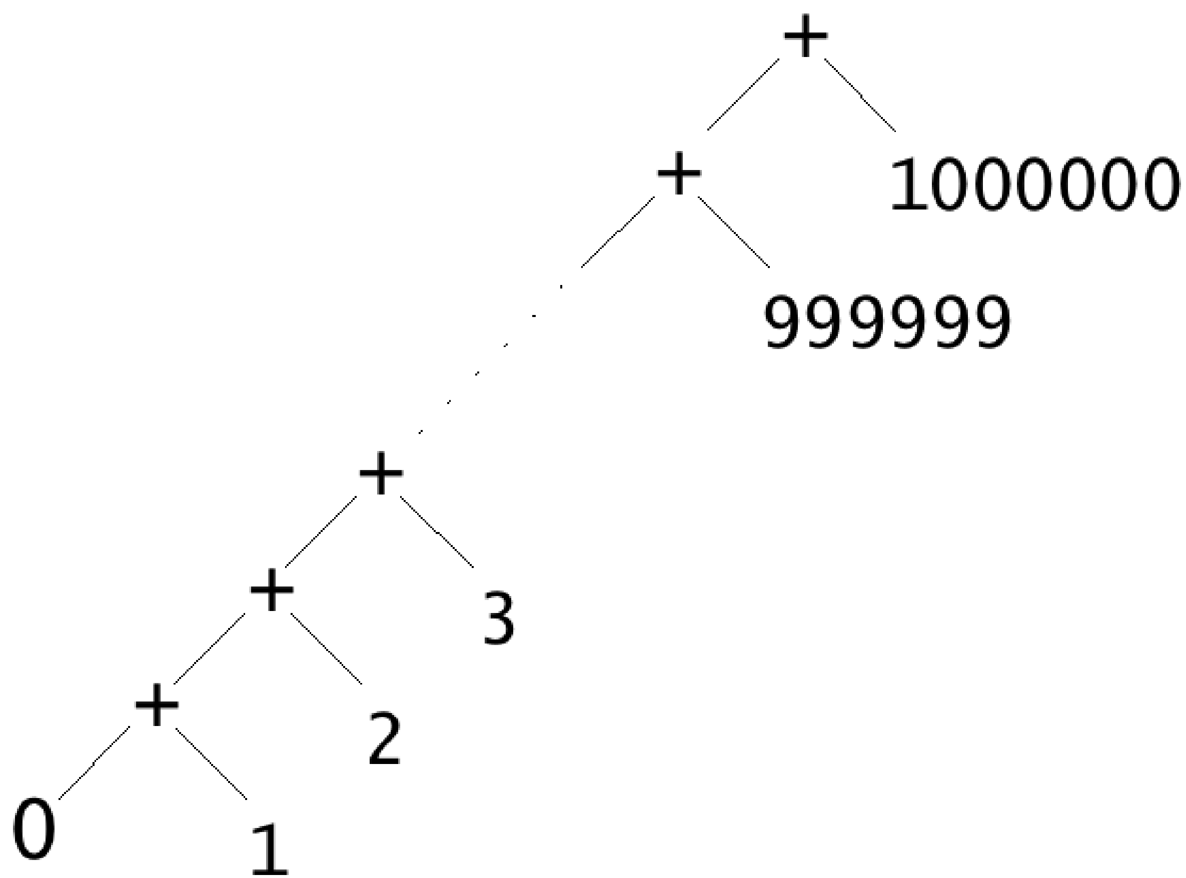
The Sequential Computation

$\texttt{var}\ sum : \mathbb{Z}32 := 0$

$\texttt{for}\ i \leftarrow seq(1 : 1000000)\ \texttt{do}$

$\quad sum \mathrel{+}= i$

$\texttt{end}$



The Parallel Computation

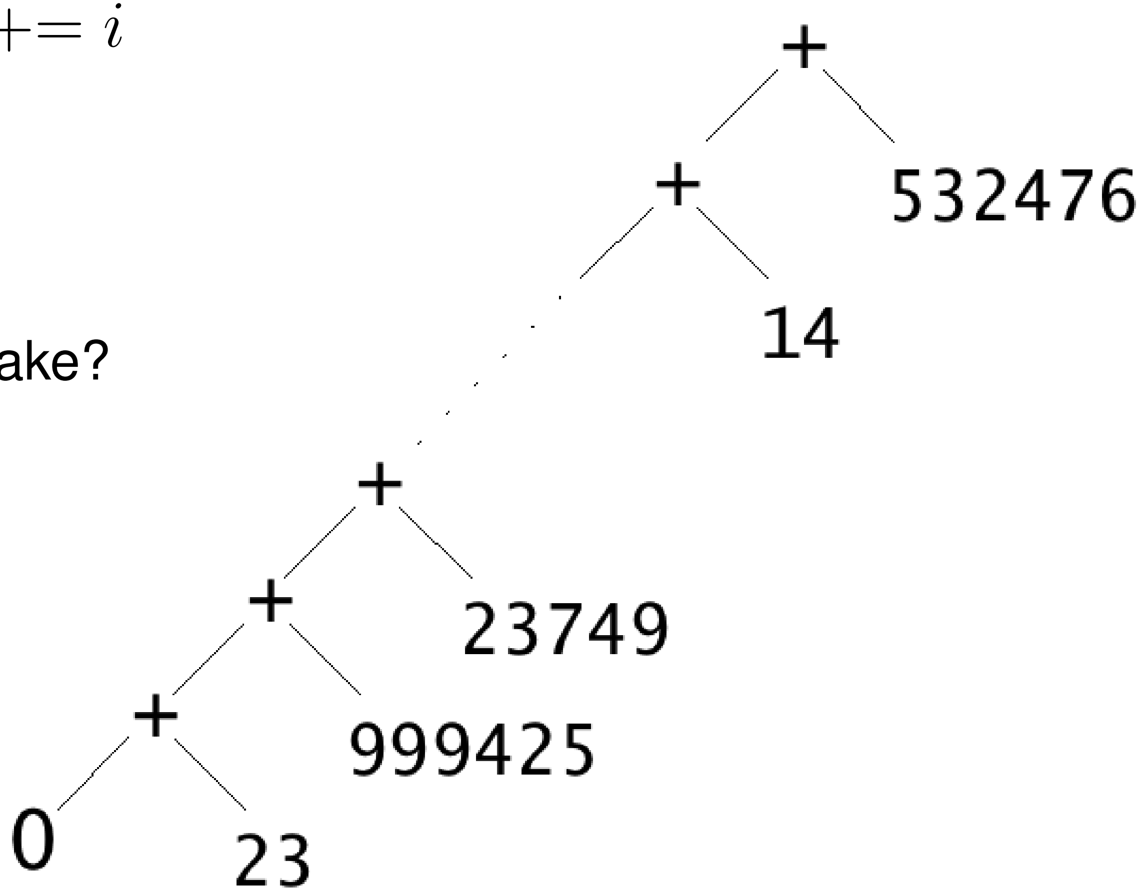
var $sum : \mathbb{Z}32 := 0$

for $i \leftarrow 1 : 1000000$ do

    atomic $sum \mathrel{+}= i$

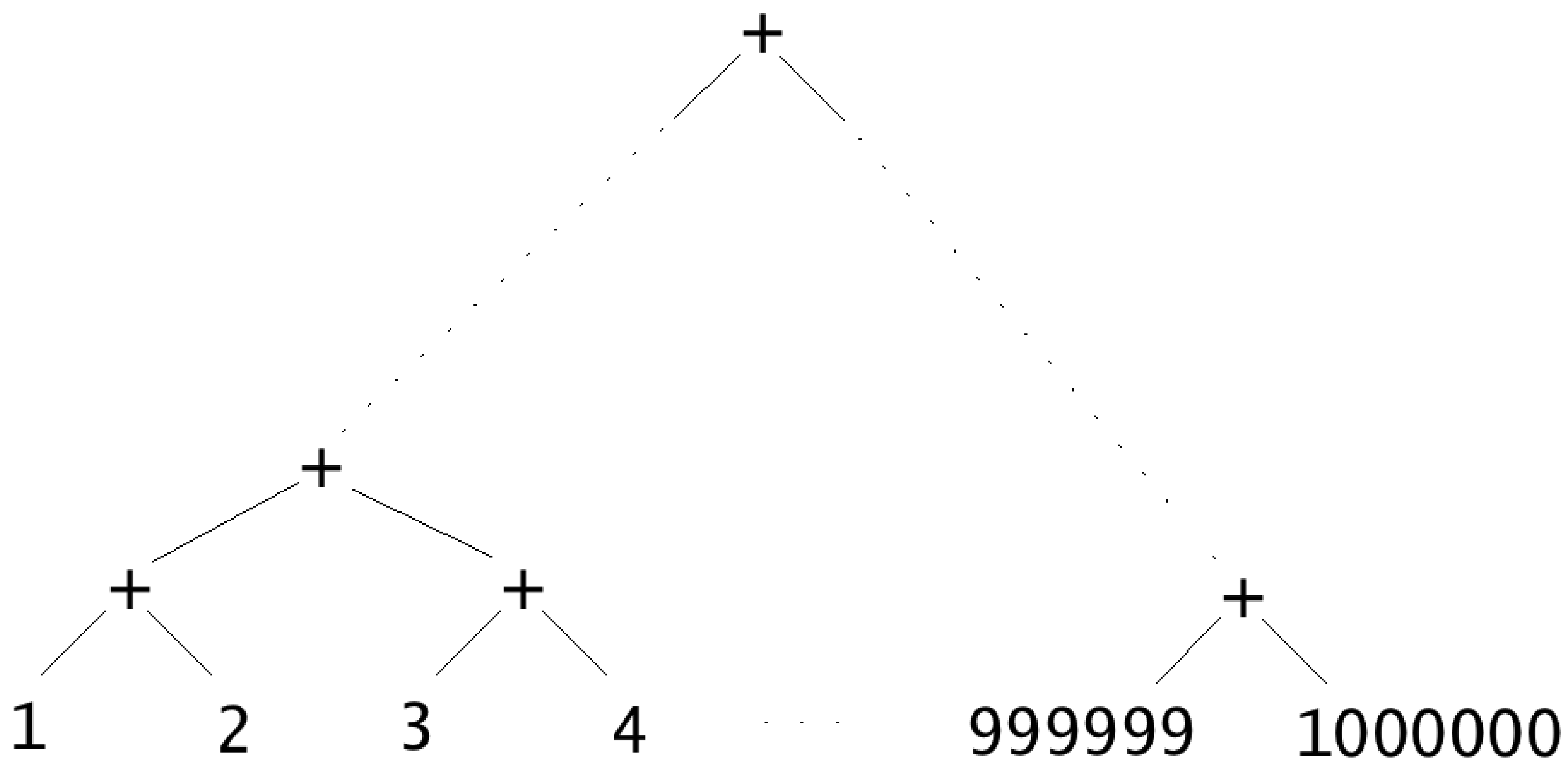
end

How long does this take?



Parallel Summation

We want a computation with this shape:



How can we achieve this?

What kind of code would we *like* to write?

What Do Mathematicians Write?

$$\sum_{i=1}^{1000000} x_i \quad \text{or maybe just} \quad \sum x$$

Compare Fortran 90 `SUM(X)`.

In Fortran we write:

$$\sum_{i \leftarrow 1:1000000} x_{\mathrm{i}} \quad \text{or} \quad \sum x_{1:1000000}$$

What, not how.

Potentially Parallel Constructs in Fortress

- Expressions with generators:

$$\sum_{\substack{x \leftarrow xs \\ y \leftarrow ys}} x\, y$$

$$\texttt{for}\ i \leftarrow p^2 : upper : p\ \texttt{do}$$
$$\quad prime[\,i\,] := false$$
$$\texttt{end}$$

$$\langle\, x^2 \mid x \leftarrow xs, x > 43\,\rangle$$

$$prime[\,i\,] := false, i \leftarrow p^2 : upper : p$$

- Functions, operators, method call recipients, and their arguments:

$$e_1\, e_2 \qquad e_1 + e_2$$

$$e_1(e_2) \qquad e_1.method(e_2)$$

- Tuples: $(a, b, c) = \big(f(x), g(y), h(z)\big)$
- Parallel blocks: $\texttt{do}\ foo(a)\ \texttt{also}\ \texttt{do}\ foo(b)\ \texttt{end}$

Accumulation

- Start with null solution (e.g., zero, empty list)
- Update solution once for each input
  - Incremental update typically *asymmetric*, *effect-ful*
- Great for single processor
  - Linear time, typically low overhead
  - Saves space: use variables rather than construct data structures
- Difficult to parallelize
  - Reuse of space serializes access
  - Effect-ful computation requires synchronization

Divide-and-Conquer

- From each input, construct a *singleton* solution
- Merge solutions (typically pairwise)

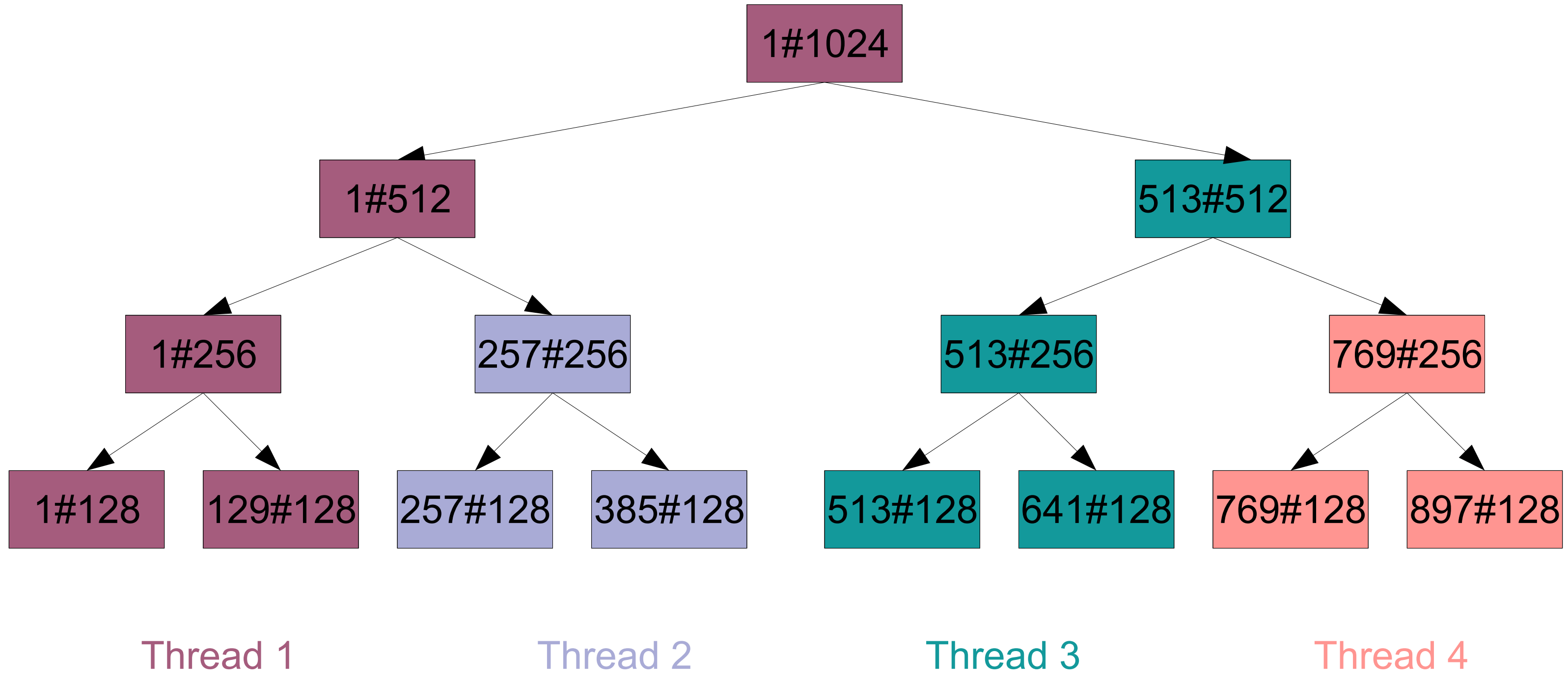
- Can take logarithmic time

- Takes more space
  - Intermediate solutions need to be heap-allocated
- Merge often more complicated than incremental update
  - Greater programming complexity, higher run-time overhead
- Requires algebraic properties of merge to exploit parallelism
  - Merge is typically associative (and often commutative)
  - Identifying appropriate merge operator is crucial

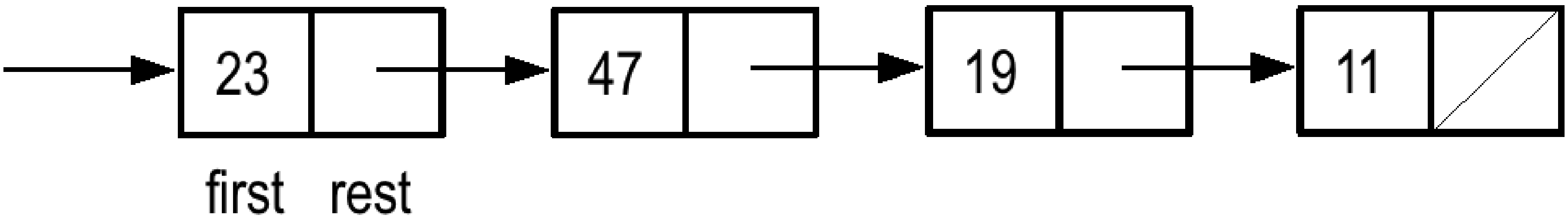
Dynamic Load Balancing: Work Stealing

- Goal: Be largely oblivious to machine size
- Express as much parallelism as possible (i.e., provide slack)
- System serializes excess parallelism

$$\sum(1 \# 1024)$$



Finding the Length of a Linked List



$$
\begin{aligned}
length(xs) = \texttt{if } (\mathit{first}, \mathit{rest}) \leftarrow xs.\mathit{extractLeft} \texttt{ then} \\
1 + length(rest) \\
\texttt{else} \\
0 \\
\texttt{end}
\end{aligned}
$$

Total work: $\Theta(n)$

Latency: $\Omega(n)$

Linearly linked data structures are inherently sequential
– analogous to using unary arithmetic!

Multi-way decomposition

- Use multi-way decomposition something like this:

$$length\,\langle\,\rangle = 0$$

$$length\,\langle x\rangle = 1$$

$$length\,(a \parallel b) = length\,a + length\,b$$

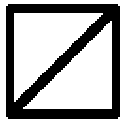
Splitting into independent, similar-sized subproblems

Total work: $\Theta(n)$

Latency: $\Omega(\log n), O(n)$ depending on how $a \parallel b$ is split;

(could be worse if splitting has worse than constant cost)

Split Lists

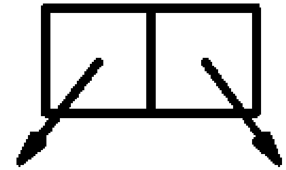


empty list

$\langle\,\rangle$

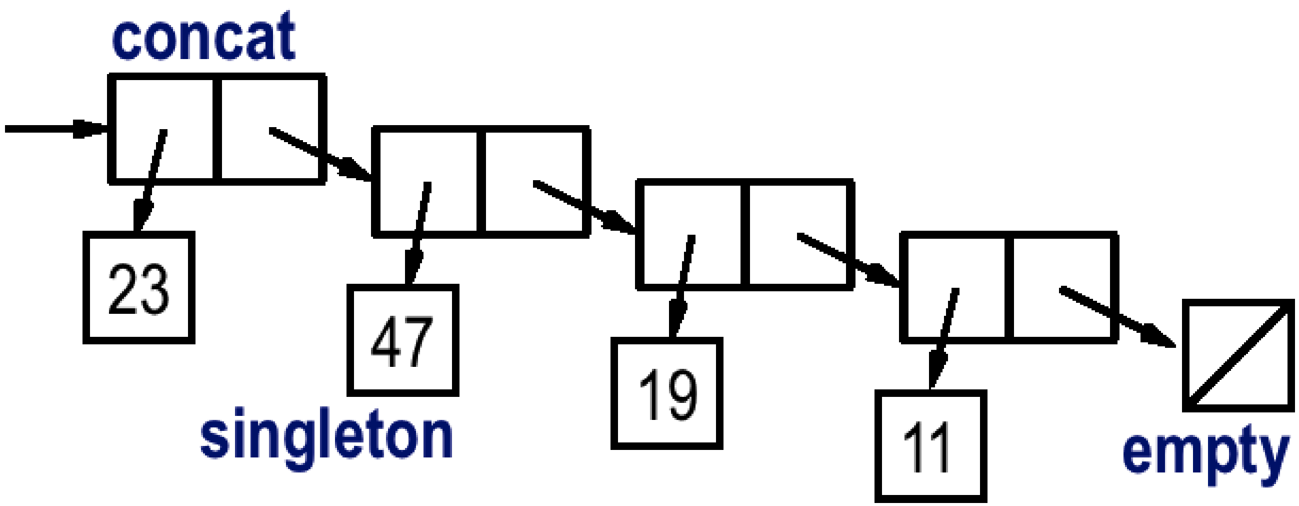
singleton

$\langle\, 23 \,\rangle$

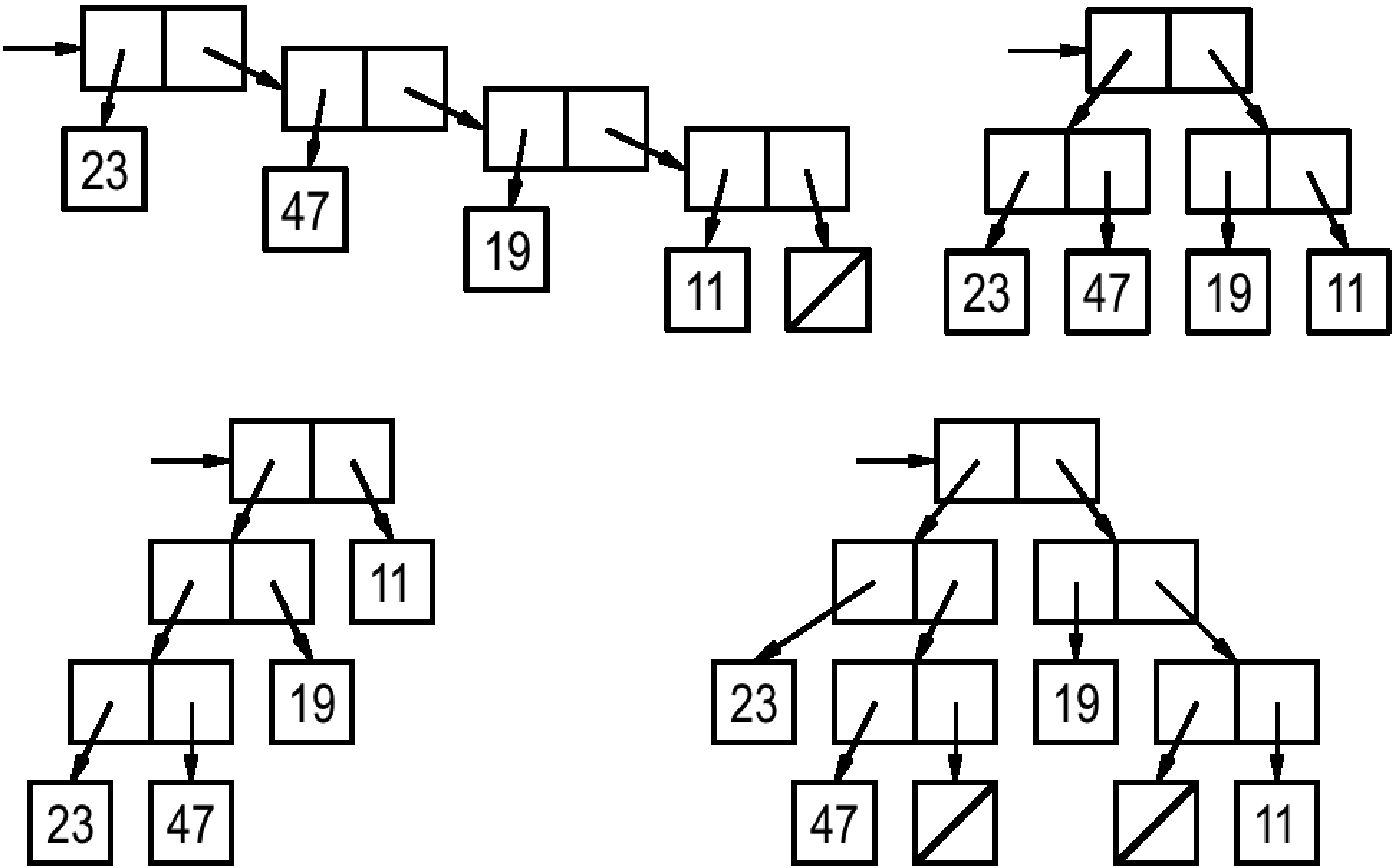


concatenation

$a \parallel b$



Split Lists for $\langle 23, 47, 19, 11 \rangle$



Primitives on Split Lists

notation	type	predicate	getters
$\langle\,\rangle$	$\mathrm{Empty}[\![T]\!]$	$isEmpty$	
$\langle x \rangle$	$\mathrm{Singleton}[\![T]\!]$	$isSingleton$	$item$
$l \parallel r$	$\mathrm{Concat}[\![T]\!]$	$isSplit$	$split$

$$\langle\, x.item \,\rangle = x$$

$$(l, r) = x.split \quad \text{implies} \quad l \parallel r = x$$

$$\langle\,\rangle \parallel a = a \parallel \langle\,\rangle = a$$

Split Lists: Length

```
trait List⟦T⟧ comprises { Empty⟦T⟧, NonEmpty⟦T⟧ }
  opr |self| : ℤ32 =
    if  self.isEmpty then 0
    elif  self.isSingleton then 1
    else
        (a, b) =  self.split
        |a| + |b| (* Parallel! *)
    end
end
```

Split Lists: Length

$\texttt{trait}\ \mathrm{List}[\![T]\!]\ \texttt{comprises}\ \{\ \mathrm{Empty}[\![T]\!], \mathrm{NonEmpty}[\![T]\!]\ \}$

$\quad\texttt{opr}\ |\texttt{self}| : \mathbb{Z}32 =$

$\quad\quad\texttt{if}\ \ \texttt{self}.\mathit{isEmpty}\ \texttt{then}\ 0$

$\quad\quad\texttt{elif}\ \ \texttt{self}.\mathit{isSingleton}\ \texttt{then}\ 1$

$\quad\quad\texttt{else}$

$\quad\quad\quad\quad(a, b) =\ \texttt{self}.\mathit{split}$

$\quad\quad\quad\quad|a| + |b|$ (* Parallel! *)

$\quad\quad\texttt{end}$

$\texttt{end}$

What else can we compute this way?

Split Lists: Filter

$\texttt{trait}\ \mathrm{List}[\![T]\!]\ \texttt{comprises}\ \{\,\mathrm{Empty}[\![T]\!], \mathrm{NonEmpty}[\![T]\!]\,\}$

$\quad \mathit{filter}(p : T \rightarrow \mathrm{Boolean}) \colon \mathrm{List}[\![T]\!] =$

$\qquad \texttt{if}\ \ \texttt{self}.\mathit{isEmpty}\ \texttt{then}\ \ \texttt{self}$

$\qquad \texttt{elif}\ \ \texttt{self}.\mathit{isSingleton}\ \texttt{then}$

$\qquad\qquad \texttt{if}\ p(\texttt{self}.\mathit{item})\ \texttt{then}\ \ \texttt{self}$

$\qquad\qquad \texttt{else}\ \langle\,\rangle\ \texttt{end}$

$\qquad \texttt{else}$

$\qquad\qquad (a, b) =\ \texttt{self}.\mathit{split}$

$\qquad\qquad a.\mathit{filter}(p) \parallel b.\mathit{filter}(p)$ (* Parallel! *)

$\qquad \texttt{end}$

$\texttt{end}$

Split Lists: Filter (continued)

$$
\mathit{reductionFilter}[\![E]\!]\big(p\colon E \rightarrow \mathrm{Boolean}, \mathit{xs}\colon \mathrm{List}[\![E]\!]\big)\colon \mathrm{List}[\![E]\!] =
$$

$$
\mathop{\Big\|}_{x \leftarrow \mathit{xs}} \Big(\texttt{if}\ p(x)\ \texttt{then}\ \langle\, x \,\rangle\ \texttt{else}\ \langle\,\rangle\ \texttt{end}\Big)
$$

Fortress has built-in support for filters in comprehension notation:

$$
\mathit{comprehensionFilter}[\![E]\!]\big(p\colon E \rightarrow \mathrm{Boolean}, \mathit{xs}\colon \mathrm{List}[\![E]\!]\big)\colon \mathrm{List}[\![E]\!] =
$$

$$
\Big\langle\, x \;\Big|\; x \leftarrow \mathit{xs}, p(x) \,\Big\rangle
$$

Point of Order

For filter, unlike summation, we maintain the original order of the elements in the input list.
(Both $\|$ and $+$ are associative, but only $+$ is commutative.)

Do not confuse the ordering of elements in the result list (a spatial order) with the order in which they are computed (a temporal order). Sequential programs often tie one to the other.

Parallel programming must decouple this unnecessary dependency. Both algorithms and data structures must change to achieve this.

This strategy for parallelism relies only on associativity, *not* commutativity.

Powerset of a List

Given: a list of integers
Result: a list of lists of integers, representing all subsets of the given list, preserving element order

Examples:

$$powerList(\langle\,\rangle) = \langle\,\langle\,\rangle\,\rangle$$

$$powerList(\langle\,1, 2\,\rangle) = \langle\,\langle\,\rangle, \langle\,2\,\rangle, \langle\,1\,\rangle, \langle\,1, 2\,\rangle\,\rangle$$

$$powerList(\langle\,1, 2, 3\,\rangle) = \langle\,\langle\,\rangle, \langle\,3\,\rangle, \langle\,2\,\rangle, \langle 2, 3\rangle, \langle\,1\,\rangle, \langle 1, 3\rangle, \langle\,1, 2\,\rangle, \langle 1, 2, 3\rangle\,\rangle$$

Powerset of a List

$$seqPowerList\left(x\colon \mathrm{List}[\![\mathbb{Z}32]\!]\right) =$$

$$\quad \texttt{if}\ |x| = 0\ \texttt{then}$$

$$\qquad \big\langle \, \langle\,\rangle \, \big\rangle$$

$$\quad \texttt{else}$$

$$\qquad (f, r) = x.extractLeft$$

$$\qquad z = seqPowerList(r)$$

$$\qquad z \parallel \big\langle \, \langle f \rangle \parallel y \;\big|\; y \leftarrow z \, \big\rangle$$

$$\quad \texttt{end}$$

Powerset of a List

$$\begin{array}{l}
parPowerList\big(x\colon \mathrm{List}[\![\mathbb{Z}32]\!]\big) = \\
\quad \texttt{if}\ |x| = 0\ \texttt{then} \\
\qquad \big\langle\, \langle\,\rangle \,\big\rangle \\
\quad \texttt{elif}\ |x| = 1\ \texttt{then} \\
\qquad \big\langle\, \langle\,\rangle, x \,\big\rangle \\
\quad \texttt{else} \\
\qquad (l, r) = x.split \\
\qquad (p, q) = \big(parPowerList(l), parPowerList(r)\big) \\
\qquad \big\langle\, a \parallel b \mid a \leftarrow p, b \leftarrow q \,\big\rangle \\
\quad \texttt{end}
\end{array}$$

Splitting a String into Words

Given: a string

Result: a list of strings, the words separated by spaces

- Words must be nonempty
- Words may be separated by more than one space
- String may or may not begin or end with spaces

Examples:

$$words(\text{“}\texttt{This\ \ is\ \ a\ \ sample}\text{”}) = \langle \text{“}\texttt{This}\text{”}, \text{“}\texttt{is}\text{”}, \text{“}\texttt{a}\text{”}, \text{“}\texttt{sample}\text{”} \rangle$$

$$words(\text{“}\texttt{\ \ \ \ Another\ \ \ sample\ \ \ }\text{”}) = \langle \text{“}\texttt{Another}\text{”}, \text{“}\texttt{sample}\text{”} \rangle$$

$$words(\text{“}\texttt{OneWord}\text{”}) = \langle \text{“}\texttt{OneWord}\text{”} \rangle$$

$$words(\text{“}\texttt{\ \ \ \ \ \ \ }\text{”}) = \langle \rangle$$

$$words(\text{“”}) = \langle \rangle$$

Can we do this in parallel?

Splitting a String into Words

```
Here is a sesquipedalian string of words
```

How should we split the input?

Splitting a String into Words

```
Here is a sesquipedalian string of words

Here is a sesquipedalian string of words

Here is a sesquipedalian string of words
```

How should we split the input?

How can we merge the intermediate solutions?

Splitting a String into Words

```
Here is a sesquipedalian string of words
```

```
Here is a sesquipedalian string of words
```

```
Here is a sesquipedalian string of words
```

How should we split the input?

How can we merge the intermediate solutions?

What ARE the intermediate solutions? Lists of words?

Generalizing length and filter: $\mathit{mapReduce}$

$$\mathit{mapReduce}[\![R]\!]\big(e\colon R, s\colon T \to R, r\colon (R,R) \to R\big)\colon R =$$

$$\quad \texttt{if}\ \ \texttt{self}.\mathit{isEmpty}\ \texttt{then}\ e$$

$$\quad \texttt{elif}\ \ \texttt{self}.\mathit{isSingleton}\ \texttt{then}\ s(\texttt{self}.\mathit{item})$$

$$\quad \texttt{else}$$

$$\qquad (a,b) =\ \texttt{self}.\mathit{split}$$

$$\qquad r\big(a.\mathit{mapReduce}(e,s,r), b.\mathit{mapReduce}(e,s,r)\big)$$

$$\quad \texttt{end}$$

$$\texttt{opr}\ \ |\texttt{self}|\colon \mathbb{Z}32 = \mathit{mapReduce}\big(0, \texttt{fn}\ i \Rightarrow 1, \texttt{fn}\ (x,y) \Rightarrow x+y\big)$$

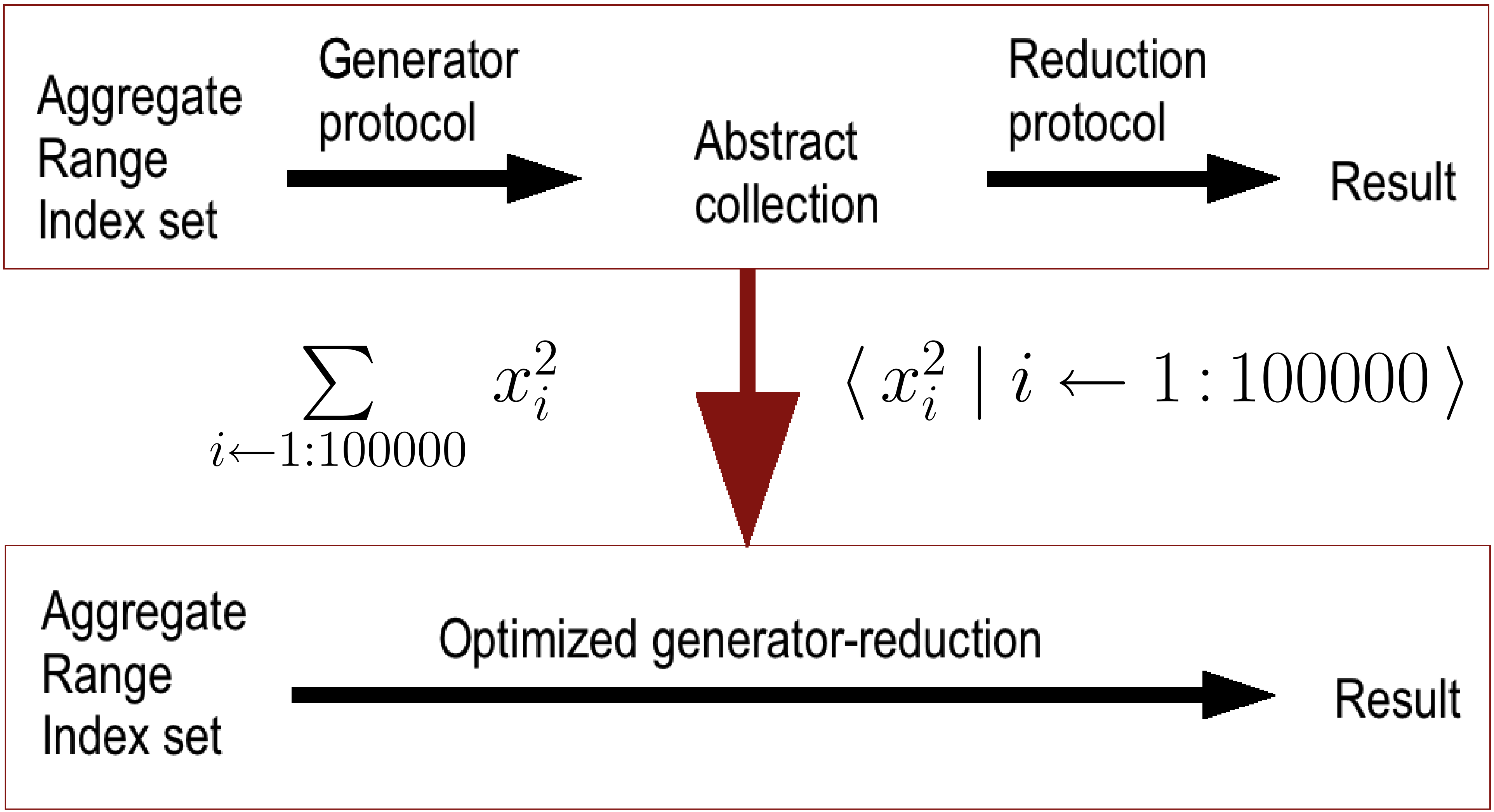
$$\mathit{filter}(p\colon T \to \text{Boolean})\colon \text{List}[\![T]\!] =$$

$$\quad \mathit{mapReduce}(\langle\,\rangle,$$

$$\qquad\qquad \texttt{fn}\ (\mathit{item}) \Rightarrow \texttt{if}\ p(\mathit{item})\ \texttt{then}\ \langle\, \mathit{item}\,\rangle\ \texttt{else}\ \langle\,\rangle\ \texttt{end},$$

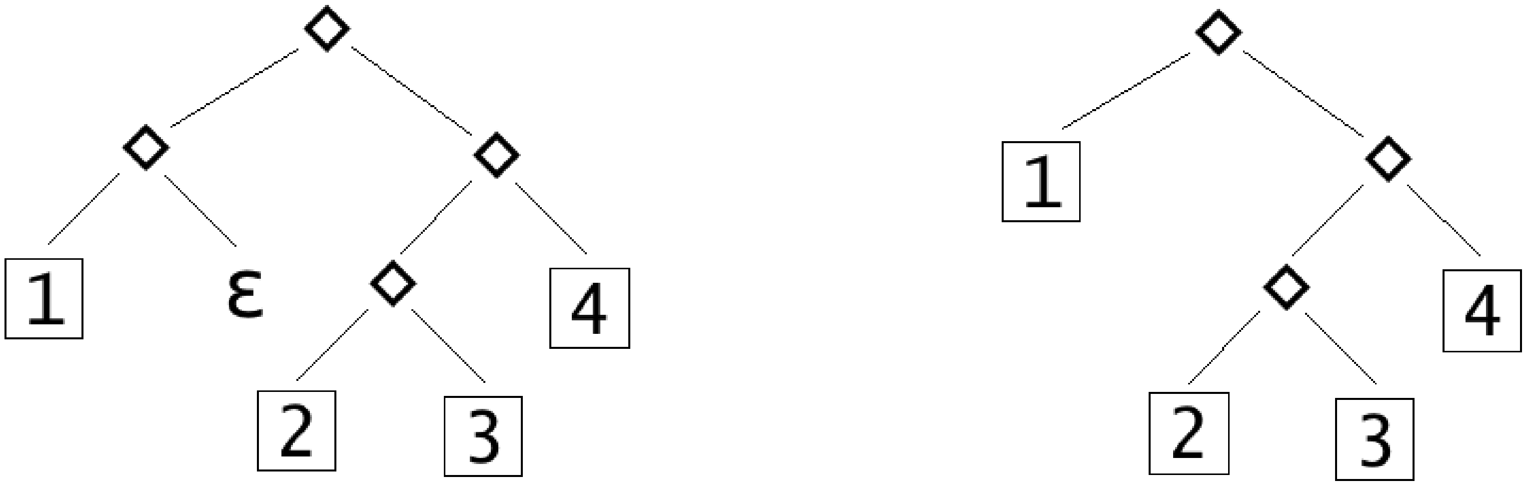
$$\qquad\qquad \texttt{fn}\ (a,b) \Rightarrow a \,\|\, b)$$

Big Idea: Abstract Collections



Representation of Abstract Collections

Binary operator ◇
Leaf operator (“unit”) □
Optional empty collection (“zero”) ε
that is the identity for ◇



Algebraic Properties

- Associative: grouping doesn’t matter
- Commutative: order doesn’t matter
- Idempotent: repetition doesn’t matter
- Identity: this value doesn’t matter
- Zero: other values don’t matter

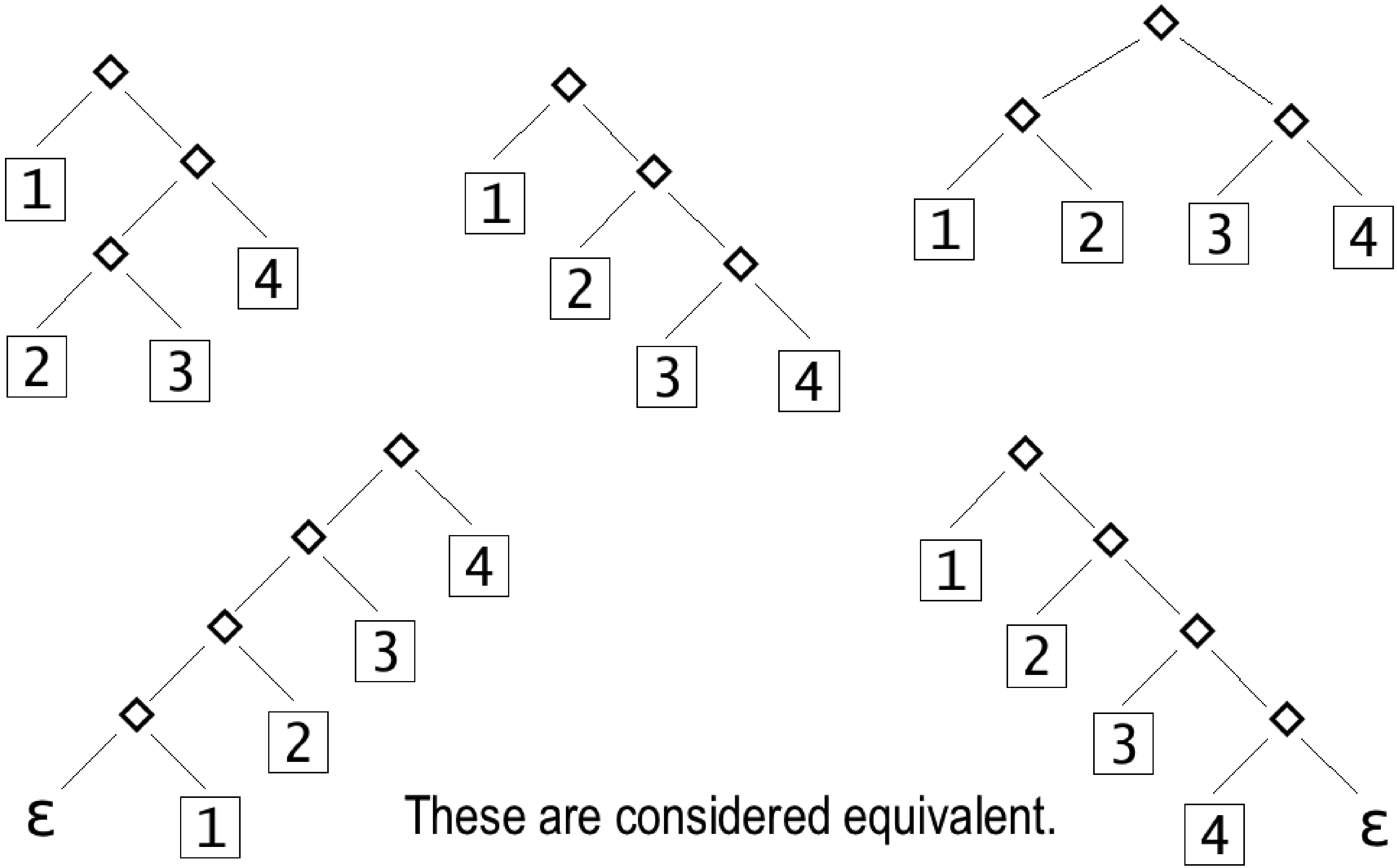
These properties give implementations *representational flexibility*
– Multiple equivalent representations of same value

The parallelism strategy described in this talk relies on associativity.

The Boom Hierarchy

Associative	Commutative	Idempotent	
			binary trees
		✓	
	✓		mobiles
	✓	✓	
✓			lists
✓		✓	
✓	✓		multisets
✓	✓	✓	sets

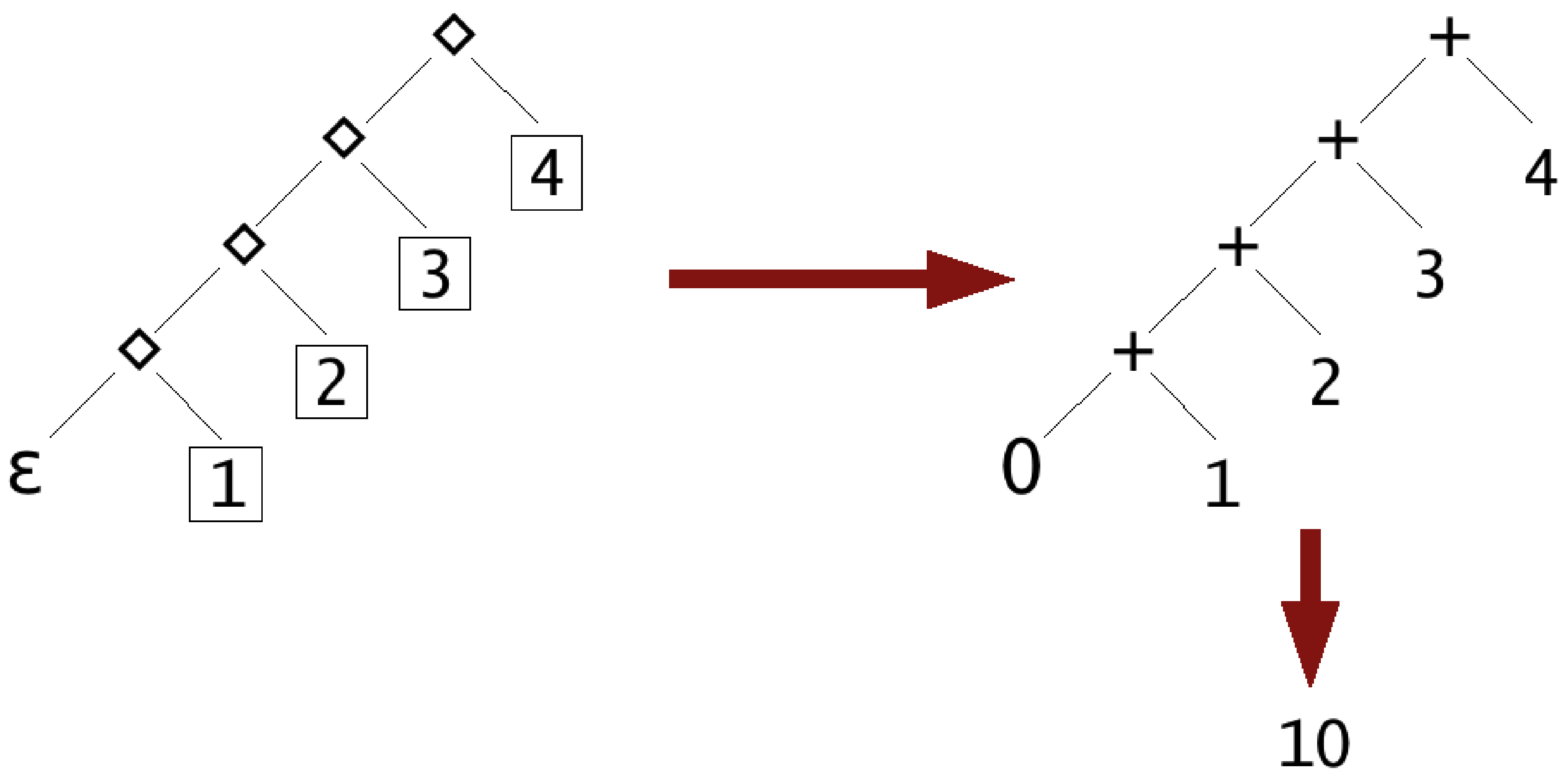
Associativity



These are considered equivalent.

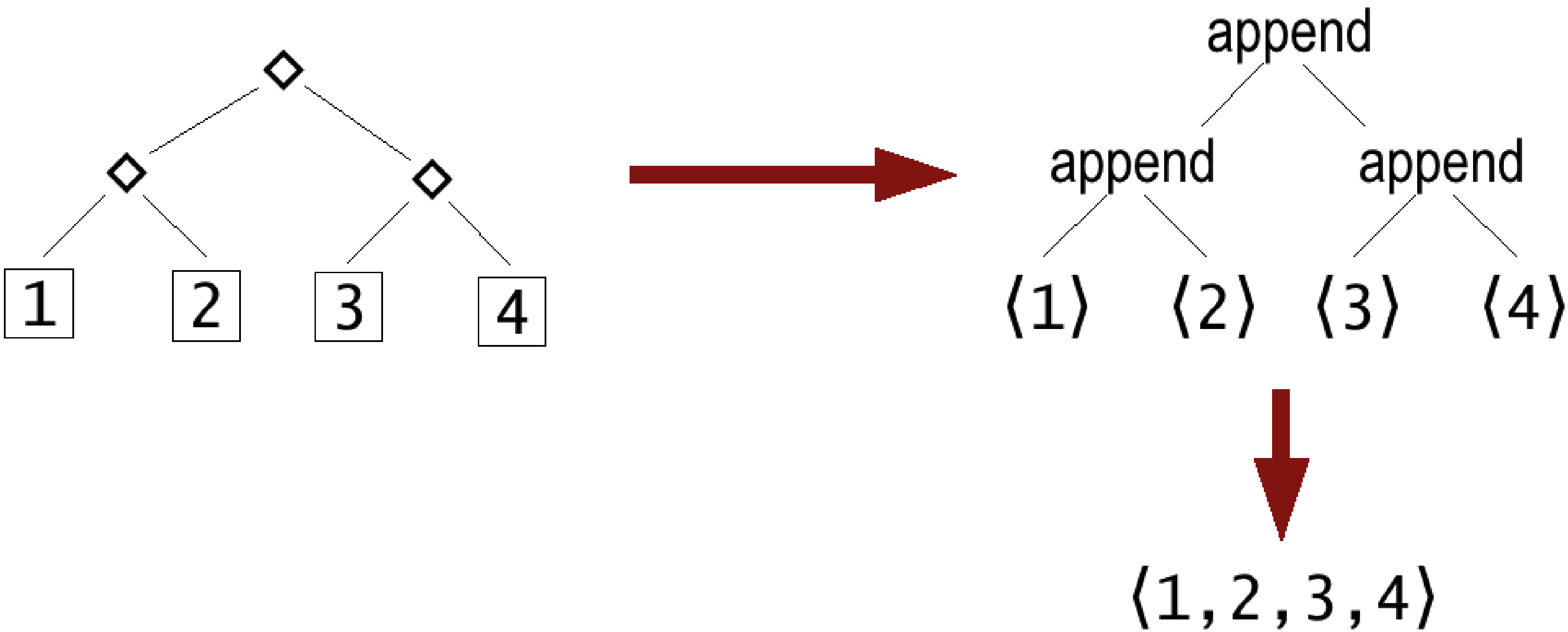
Summation

Replace ◇ □ ε with + identity 0



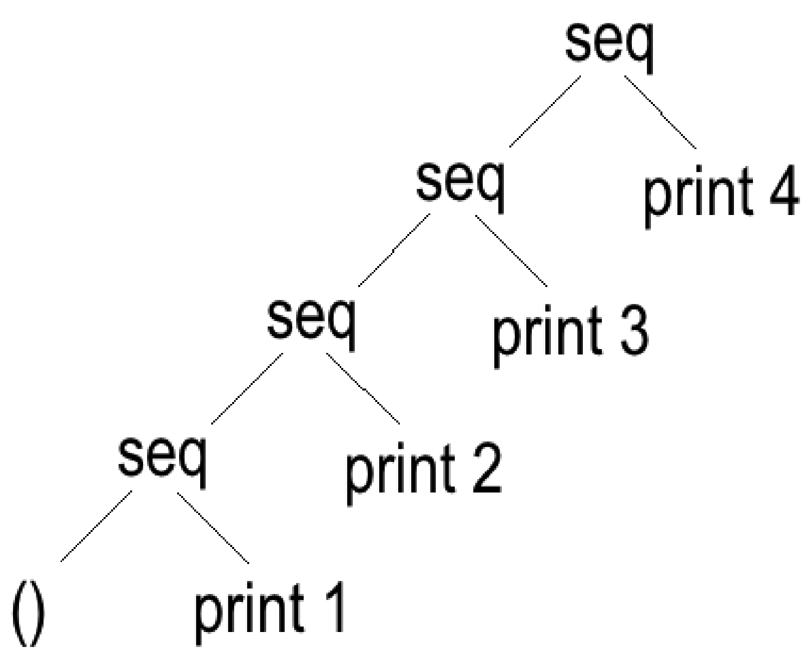
Lists

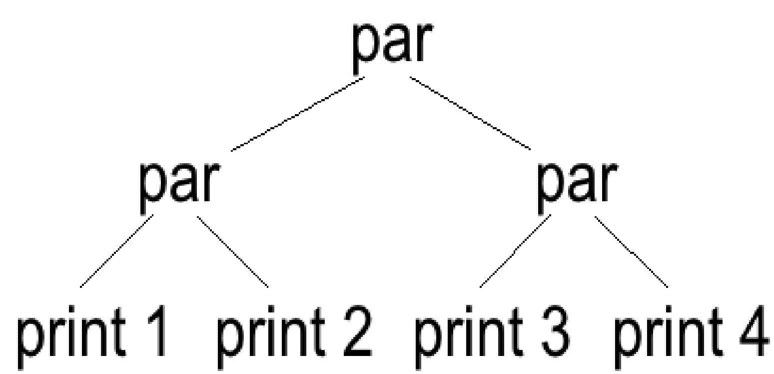
Replace $\diamond$ $\square$ ε with append $\langle - \rangle$ $\langle\rangle$



Loops

Replace $\Diamond$ $\Box$ ε with seq identity () or par identity () where seq: $(),() \rightarrow ()$ and par: $(),() \rightarrow ()$





Summary: Big Idea

- Summations and list constructors and loops are alike!

$$\sum_{i \leftarrow 1:1000000} x_{\mathrm{i}}^2$$

$$\langle\, x_{\mathrm{i}}^2 \mid i \leftarrow 1:1000000 \,\rangle$$

$$\begin{array}{l}\texttt{for } i \leftarrow 1:1000000 \texttt{ do} \\ \quad x_{\mathrm{i}} := x_{\mathrm{i}}^2 \\ \texttt{end}\end{array}$$

  - Generate an abstract collection
  - The *body* computes a function of each item
  - Combine the results (or just synchronize)
  - In other words: map and reduce
    - Contract for $mapReduce$ allows loop optimizations
- Whether to be sequential or parallel is a separable question
  - That's why they are especially good abstractions!
  - Make the decision on the fly, to use available resources

Out with the Old

- DO loops
- Linear linked lists
- Java-style iterators
- Even arrays are suspect

- Accumulators are BAD.
  - As soon as you say “first, SUM = 0”, you are in trouble.
- Don’t “process subproblems in order”

- We must give up many great tricks and programming idioms of the sequential past

In with the New

We need parallel strategies for problem decomposition, data structure design, and algorithm organization

- The top-down view:
  Don't split a problem into "the first" and "the rest." Instead, split a problem into roughly equal pieces; recursively solve subproblems, then combine sub-solutions.
  When dependencies exist, identify maximal independent chunks of computation and run in parallel within each chunk.
- The bottom-up view:
  Don't create a null solution, then successively update it;
  Instead, map inputs independently to singleton solutions,
  then merge the sub-solutions treewise.
- Combining sub-solutions is usually trickier than incremental update of a single solution.

Try Fortress

- Parallel interpreter written in Java
  - Easy to dash off a quick Fortress program like $sieve$
- Compiler to java bytecode partially complete
  - Small programs work, but still a ways to go yet

`http://projectfortress.sun.com/`

Subversion download gives the latest version of the code:

```
svn checkout https://projectfortress.sun.com/svn/Community/trunk PFC
```

ORACLE®