Project Fortress:

A Multicore Language for Scientists and Engineers

Sukyoung Ryu

Department of Computer Science
Korea Advanced Institute of Science and Technology

September 12, 2010

Project Fortress

- A multicore language for scientists and engineers

Project Fortress

- A multicore language for scientists and engineers
- Run your whiteboard in parallel!

Project Fortress

- A multicore language for scientists and engineers
- Run your whiteboard in parallel!

$$v_{\mathrm{norm}} = v / \|v\|$$

$$\sum_{k \leftarrow 1:n} a_k \, x^k$$

$$C = A \cup B$$

$$y = 3x \sin x \cos 2x \log \log x$$

Project Fortress

- A multicore language for scientists and engineers
- Run your whiteboard in parallel!

$$v_{\mathrm{norm}} = \underline{v/\underline{\|v\|}}$$

$$\sum_{\underline{k \leftarrow 1:n}} \underline{a_k}\ \underline{x^k}$$

$$C = \underline{A \cup B}$$

$$y = \underline{\underline{3x}\ \underline{\sin x}}\ \underline{\cos \underline{2x}}\ \underline{\log \underline{\log x}}$$

Project Fortress

- A multicore language for scientists and engineers
- Run your whiteboard in parallel!

$$v_{\text{norm}} = \underline{v / \underline{\|v\|}}$$

$$\underline{\sum_{k \leftarrow 1:n}} \underline{a_k} \, \underline{x^k}$$

$$C = \underline{A \cup B}$$

$$y = \underline{3x} \, \underline{\sin x} \, \underline{\cos \underline{2x}} \, \underline{\log \underline{\log x}}$$

- "Growing a Language"

  Guy L. Steele Jr., keynote talk, OOPSLA 1998

Project Fortress

- Fortress is a growable, mathematically oriented, parallel programming language for scientific applications.
- Started under Sun/DARPA HPCS program, 2003–2006.
- Fortress is now an open-source project with international participation.
- The Fortress 1.0 release (March 2008) synchronized the specification and implementation.
- Moving forward, we are growing the language and libraries and developing a compiler.

Parallelism by Default

A Parallel Language

High productivity for multicore, SMP, and cluster computing

- Hard to write a program that isn't potentially parallel
- Support for parallelism at several levels
  - Expressions
  - Loops, reductions, and comprehensions
  - Parallel code regions
  - Explicit multithreading
- Shared global address space model with shared data
- Thread synchronization through atomic blocks and transactional memory

Implicit Parallelism

- Tuples

  $(a, b, c) = \big(f(x), g(y), h(z)\big)$

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

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

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

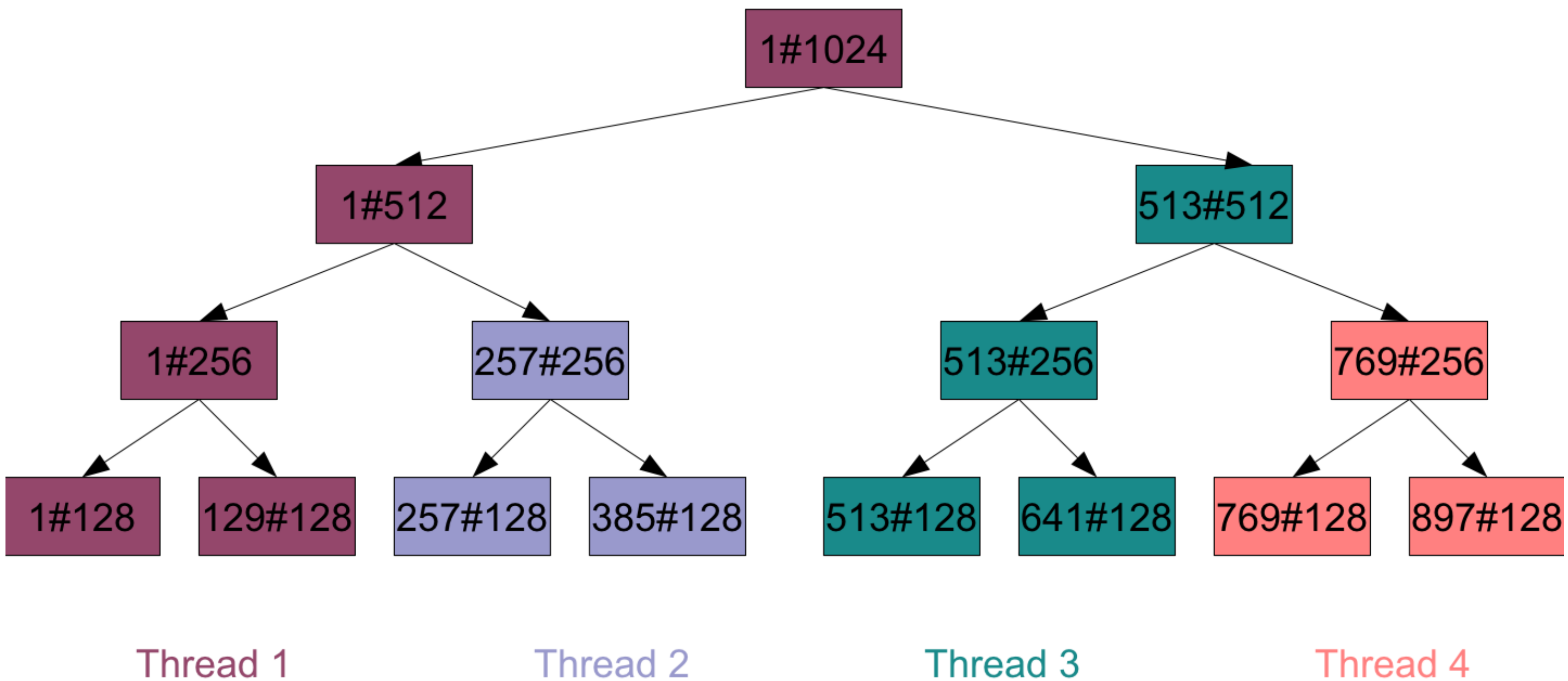
- Expressions with generators

  $$s = \sum_{k \leftarrow 1:n} c_k\, x^k$$

  $\{\, x^2 \mid x \leftarrow xs, x > 43 \,\}$

Recursive Subdivision and Work Stealing

$$\sum_{k \leftarrow 1\#1024} c_k\, x^k$$



With Multicore, a Profound Shift

- Parallelism is here, now, and in our faces
  - Academics have been studying it for 50 years
  - Serious commercial offerings for 25 years
  - But now it's in desktops and laptops
- Specialized expertise for science codes and databases and networking
- But soon general practitioners must go parallel
- An opportunity to make parallelism easier for everyone

KAIST Computer Science

The Big Messages

- Effective parallelism uses trees.
- Associative combining operators are good.
- MapReduce is good.
- There are systematic strategies for parallelizing superficially sequential code.
- We must lose the "accumulator" paradigm and emphasize "divide-and-conquer."

It Is All about Performance

The bag of programming tricks

that has served us so well

for the last 50 years

is

the wrong way to think

going forward and

must be thrown out.

Why?

- Good sequential code minimizes total number of operations.
  - > Clever tricks to reuse previously computed results.
  - > Good parallel code often performs redundant operations to reduce communication.
- Good sequential algorithms minimize space usage.
  - > Clever tricks to reuse storage.
  - > Good parallel code often requires extra space to permit temporal decoupling.
- Sequential idioms stress linear problem decomposition.
  - > Process one thing at a time and accumulate results.
  - > Good parallel code usually requires multiway problem decomposition and multiway aggregation of results.

Let's Add a Bunch of Numbers

```
DO I = 1, 1000000
  SUM = SUM + X(I)
END DO
```

Can it be parallelized?

KAIST Computer Science

Let's Add a Bunch of Numbers

```
SUM = 0                 !Oops!

DO I = 1, 1000000
  SUM = SUM + X(I)
END DO
```

Can it be parallelized?

This is already bad!

Clever compilers have to undo this.

What Does a Mathematician Say?

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

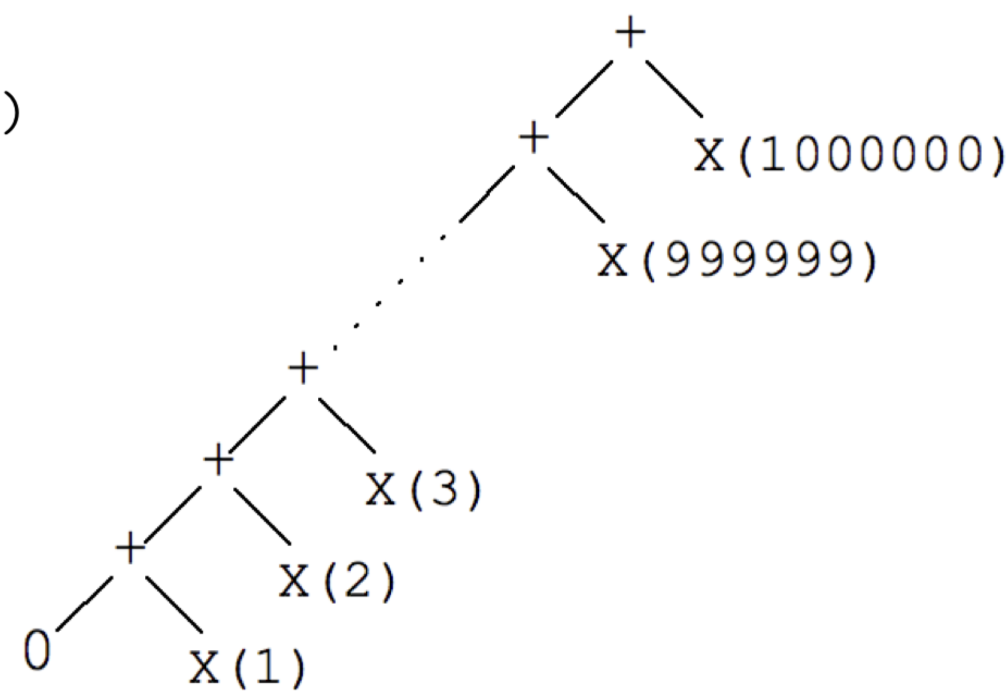
Compare Fortran 90 `SUM(X)`.

What, not how.

No commitment yet as to strategy. This is good.

Sequential Computation Tree

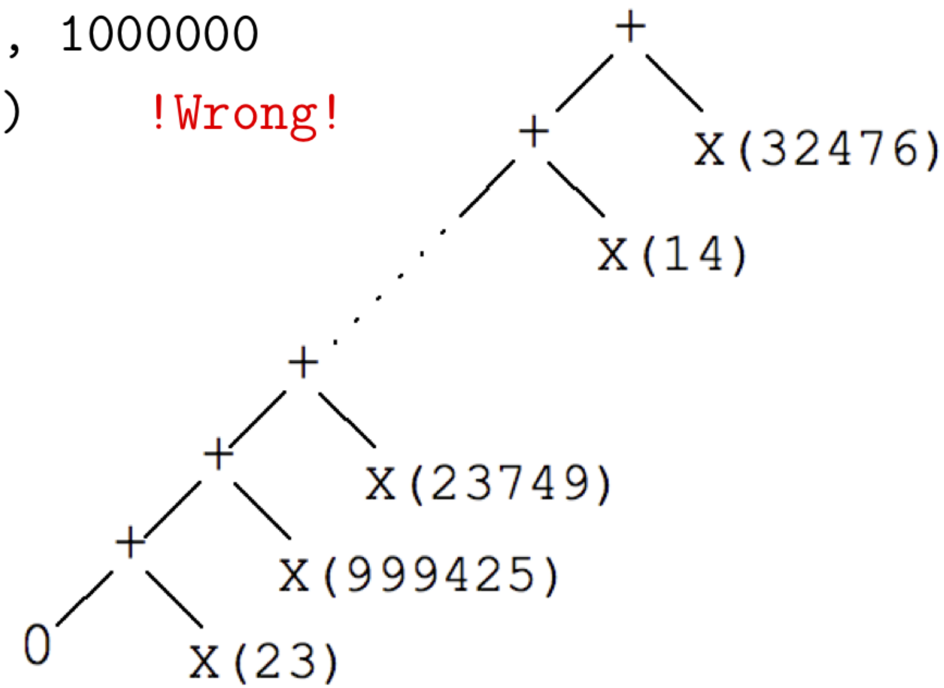
```
SUM = 0
DO I = 1, 1000000
  SUM = SUM + X(I)
END DO
```



Atomic Update Computation Tree (a)

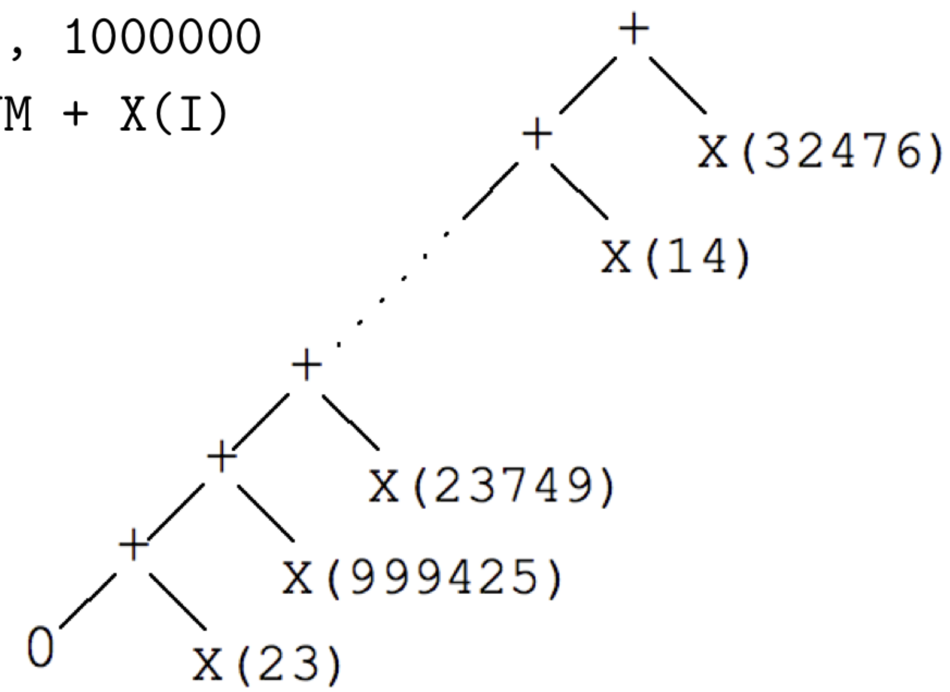
```
SUM = 0
PARALLEL DO I = 1, 1000000
  SUM = SUM + X(I)      !Wrong!
END DO
```

Race condition
can cause updates
to be lost.



Atomic Update Computation Tree (b)

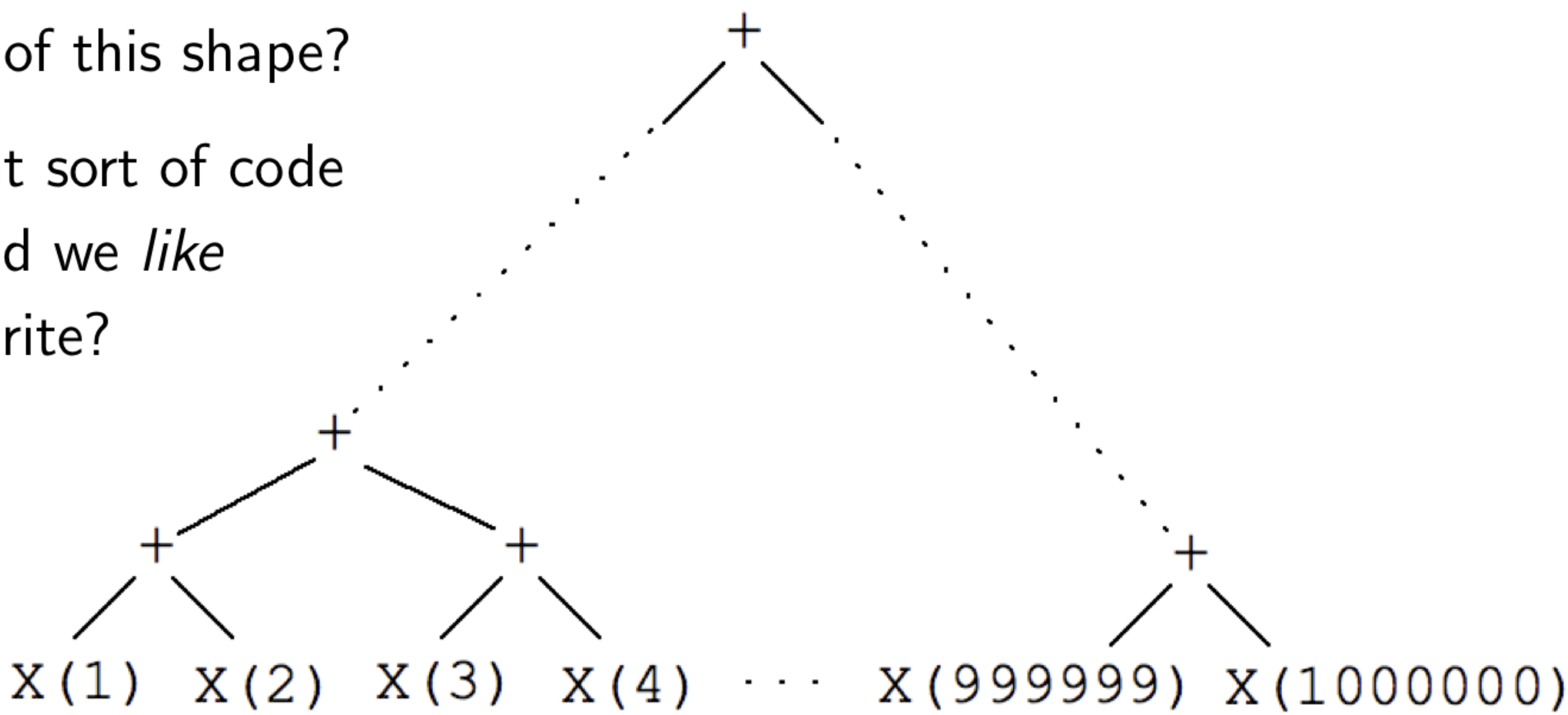
```
SUM = 0
PARALLEL DO I = 1, 1000000
  ATOMIC SUM = SUM + X(I)
END DO
```



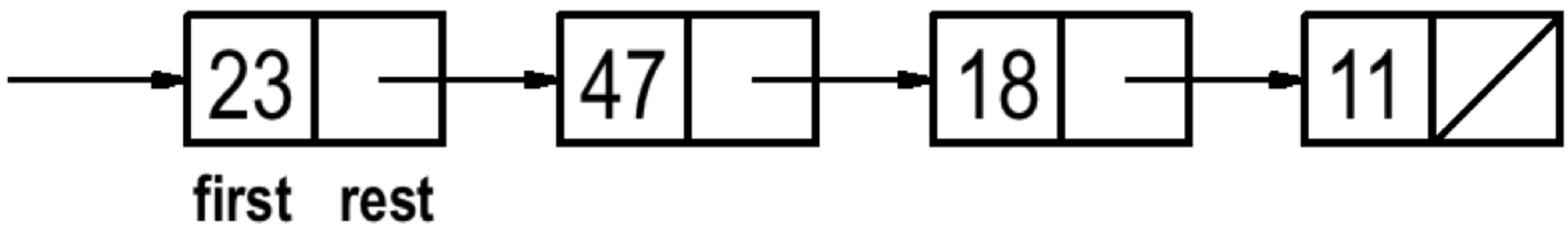
Parallel Computation Tree

What sort of code should we write to get a computation tree of this shape?

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



Finding the Length of a LISP List



Recursive:

```
(define length (list)
  (cond ((null list) 0)
        (else (+ 1 (length (rest list))))))
```

Total work: $\Theta(n)$

Delay: $\Omega(n)$

Linear versus Multiway Decomposition

- Linearly linked lists are inherently sequential.
  - Compare Peano arithmetic: $5 = ((((0+1)+1)+1)+1)+1$
  - Binary arithmetic is much more efficient than unary!
- We need a multiway decomposition paradigm:

```
length [ ] = 0
length [a] = 1
length (a++b) = (length a) + (length b)
```

This is just a summation problem: adding up a bunch of 1's!

Total work: $\Theta(n)$

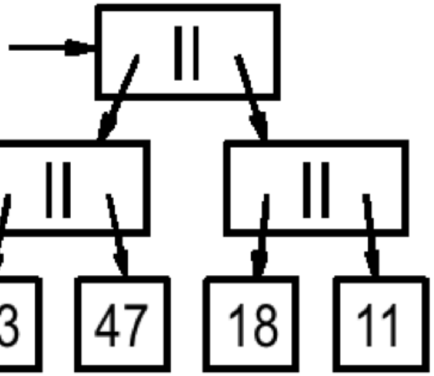
Delay: $\Omega(\log n)$, $O(n)$ depending on how a++b is split;

even worse if splitting has worse than constant cost

Conc Lists

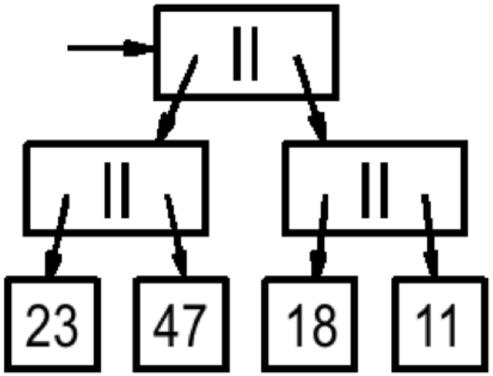
empty list	singleton	concatenation
$\langle\,\rangle$	$\langle\, 23 \,\rangle$	$a \parallel b$

Conc List for $\langle 23, 47, 18, 11 \rangle$ (1)

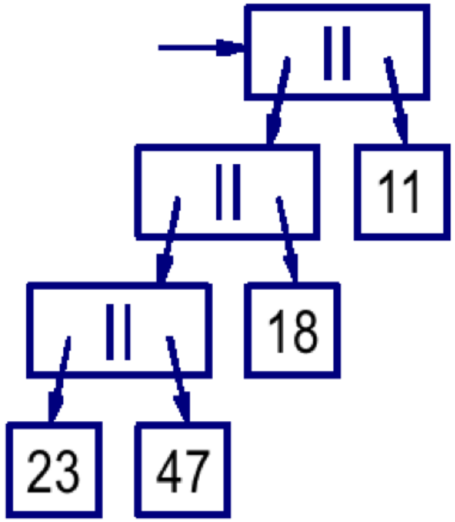


$$\Big(\langle\, 23 \,\rangle \parallel \langle\, 47 \,\rangle\Big) \parallel \Big(\langle\, 18 \,\rangle \parallel \langle\, 11 \,\rangle\Big)$$

Conc Lists for $\langle 23, 47, 18, 11\rangle$ (2)

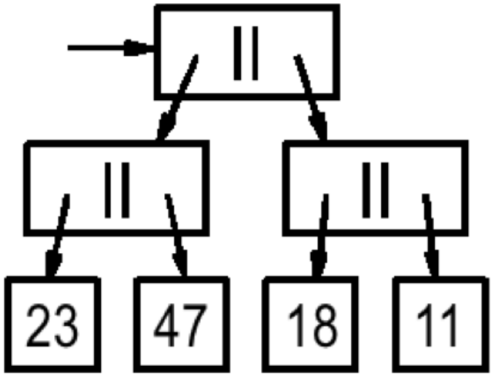


$$\Big(\langle\,23\,\rangle \parallel \langle\,47\,\rangle\Big) \parallel \Big(\langle\,18\,\rangle \parallel \langle\,11\,\rangle\Big)$$

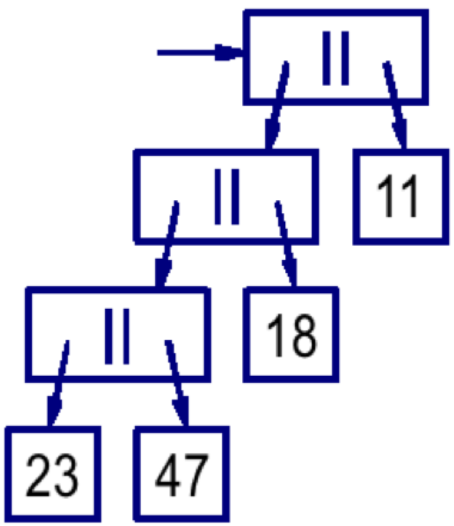


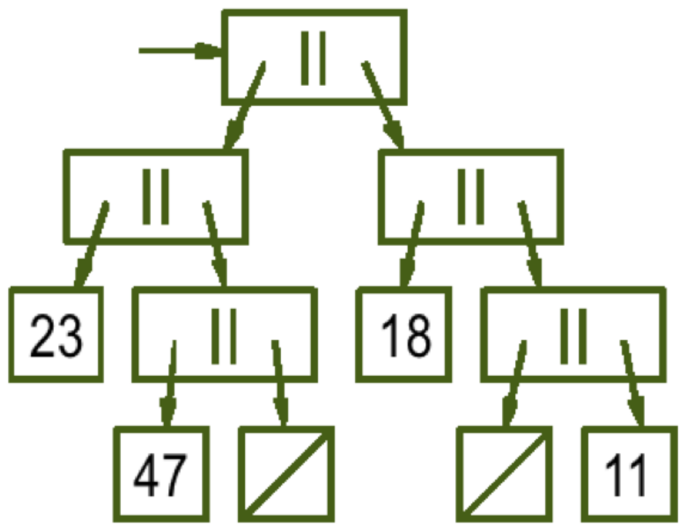
$$\Big(\Big(\langle\,23\,\rangle \parallel \langle\,47\,\rangle\Big) \parallel \langle\,18\,\rangle\Big) \parallel \langle\,11\,\rangle$$

Conc Lists for $\langle 23, 47, 18, 11\rangle$ (3)

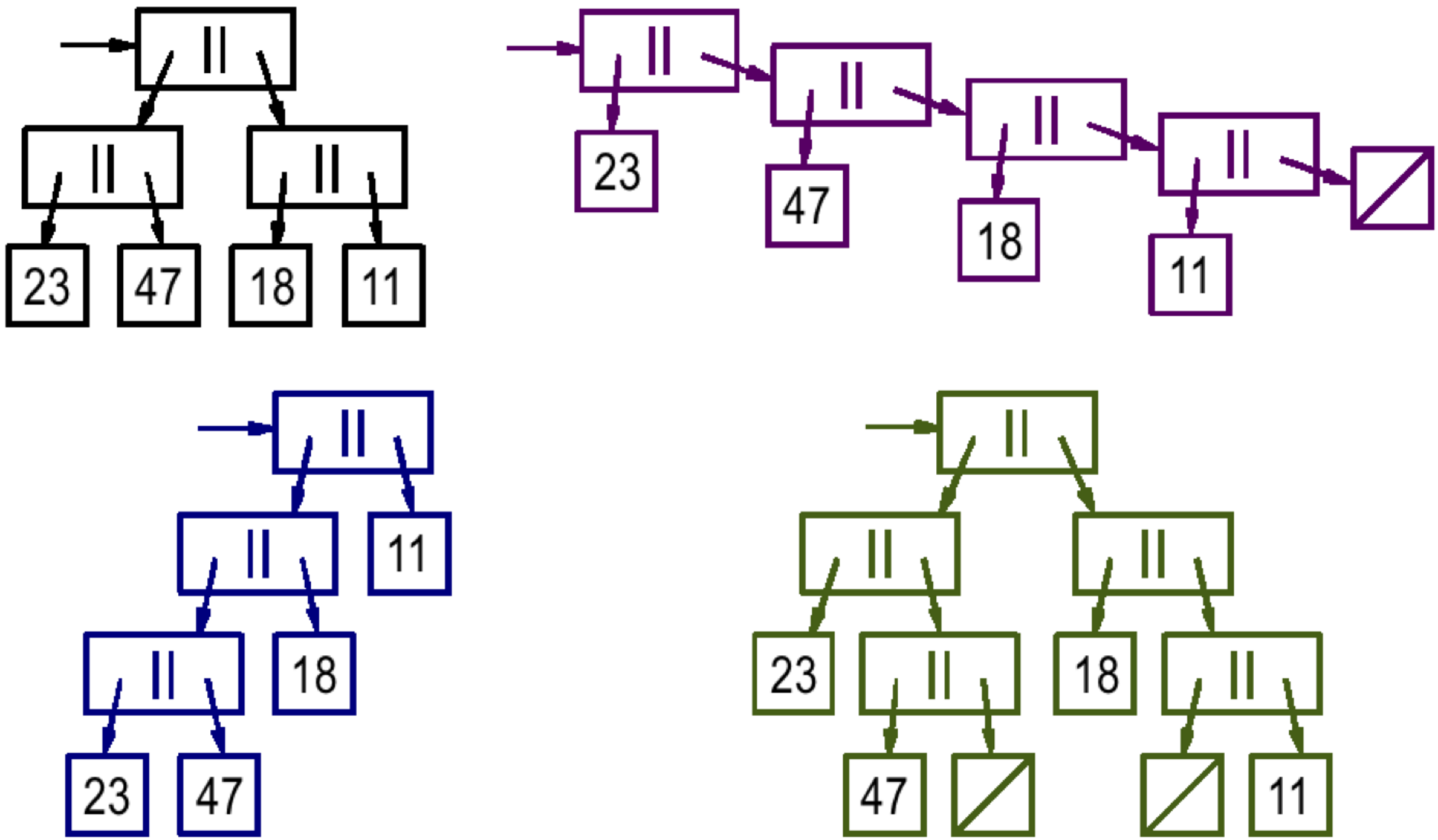


$$\Big(\langle 23\rangle \,\|\, \big(\langle 47\rangle \,\|\, \langle\rangle\big)\Big)$$
$$\|\, \Big(\langle 18\rangle \,\|\, \big(\langle\rangle \,\|\, \langle 11\rangle\big)\Big)$$





Conc Lists for $\langle 23, 47, 18, 11 \rangle$ (4)



KAIST Computer Science

Primitives on Lists (1)

	constructors	predicates	accessors
cons lists	`'()`	`null?`	
	`(cons a ys)`		`car`, `cdr`
	`(cons (car xs) (cdr xs)) = xs`		
conc lists	`'()`	`null?`	
	`(list a)`	`singleton?`	`item`
	`(conc ys zs)`		`left`, `right`
	`(list (item s)) = s`		
	`(conc (left xs) (right xs)) = xs`		

Primitives on Lists (2)

	constructors	predicates	accessors
cons lists	`'()`	`null?`	
	`(cons a ys)`		`car`, `cdr`
	`(cons (car xs) (cdr xs)) = xs`		
conc lists	`'()`	`null?`	
	`(list a)`	`singleton?`	`item`
	`(conc ys zs)`		`split`
	`(list (item s)) = s`		
	`(split xs (λ (ys zs) (conc ys zs))) = xs`		

Primitives on Lists (3)

	constructors	predicates	accessors
cons lists	`'()`	`null?`	
	`(cons a ys)`		`car`, `cdr`
	`(cons (car xs) (cdr xs)) = xs`		
conc lists	`'()`	`null?`	
	`(list a)`	`singleton?`	`item`
	`(conc ys zs)`		`split`
	`(list (item s)) = s`		
	`(split xs (λ (ys zs) (conc ys zs))) = xs`		
	`(split xs conc) = xs`		

Defining Lists Using `car cdr cons` (1)

```
(define (first x)
  (cond ((null?  x) '())
        (else (car x))))

(define (rest x)
  (cond ((null?  x) '())
        (else (cdr x))))

(define (append xs ys)
  (cond ((null?  xs) ys)
        (else (cons (car xs) (append (cdr xs) ys)))))

(define (addleft a xs) (cons a xs))

(define (addright xs a)
  (cond ((null?  xs) (list a))
        (else (cons (car xs) (addright (cdr xs) a)))))
```

Defining Lists Using car cdr cons (2)

```
(define (first x)                        ;Constant time
  (cond ((null?  x) '())
        (else (car x))))

(define (rest x)                         ;Constant time
  (cond ((null?  x) '())
        (else (cdr x))))

(define (append xs ys)                   ;Linear in (length xs)
  (cond ((null?  xs) ys)
        (else (cons (car xs) (append (cdr xs) ys)))))

(define (addleft a xs) (cons a xs))   ;Constant time

(define (addright xs a)                  ;Linear in (length xs)
  (cond ((null?  xs) (list a))
        (else (cons (car xs) (addright (cdr xs) a)))))
```

Defining Lists Using `item list split conc` (1)

```
(define (first xs)                          ;Depth of left path
  (cond ((null?  xs) '())
        ((singleton?  xs) (item xs))
        (else (split xs (λ (ys zs) (first ys))))))

(define (rest xs)                           ;Depth of left path
  (cond ((null?  xs) '())
        ((singleton?  xs) '())
        (else (split xs (λ (ys zs) (append (rest ys) zs))))))

(define (append xs ys)                      ;Constant time
  (cond ((null?  xs) ys)
        ((null?  ys) xs)
        (else (conc xs ys))))
```

Defining Lists Using `item list split conc` (2)

```
(define (first xs)                             ;Depth of left path
  (cond ((null?  xs) '())
        ((singleton?  xs) (item xs))
        (else (split xs (λ (ys zs) (first ys))))))

(define (rest xs)                              ;Depth of left path
  (cond ((null?  xs) '())
        ((singleton?  xs) '())
        (else (split xs (λ (ys zs) (append (rest ys) zs))))))

(define (append xs ys)                         ;???
  (cond ((null?  xs) ys)
        ((null?  ys) xs)
        (else (REBALANCE (conc xs ys)))))
```

Defining Lists Using `item list split conc` (3)

```
(define (addleft a xs)
  (cond ((null?  xs) (list a))
        ((singleton?  xs) (append (list a) xs))
        (else (split xs (λ (ys zs) (append (addleft a ys) zs))))))

(define (addright xs a)
  (cond ((null?  xs) (list a))
        ((singleton?  xs) (append xs (list a)))
        (else (split xs (λ (ys zs) (append ys (addright zs a))))))
```

Defining Lists Using `item list split conc` (4)

```
(define (addleft a xs) (append (list a) xs))
```

```
(define (addright xs a) (append xs (list a)))
```

map reduce mapreduce Using car cdr cons

```
(map (λ (x) (* x x)) '(1 2 3)) ⇒ (1 4 9)

(reduce + 0 '(1 4 9)) ⇒ 14

(mapreduce (λ (x) (* x x)) + 0 '(1 2 3)) ⇒ 14

(define (map f xs)                 ;Linear in (length xs)
  (cond ((null?  xs) '())
        (else (cons (f (car xs)) (map f (cdr xs))))))

(define (reduce g id xs)           ;Linear in (length xs)
  (cond ((null?  xs) id)
        (else (g (car xs) (reduce g id (cdr xs))))))

(define (mapreduce f g id xs)      ;Linear in (length xs)
  (cond ((null?  xs) id)
        (else (g (f (car xs)) (mapreduce f g id (cdr xs))))))
```

length filter **Using** car cdr cons

```
(define (length xs)                          ;Linear in (length xs)
  (mapreduce (λ (q) 1) + 0 xs))

(define (filter p xs)                        ;Linear in (length xs)
  (cond ((null?  xs) '())
        ((p (car xs)) (cons (car xs) (filter p (cdr xs))))
        (else (filter p (cdr x)))))

(define (filter p xs)                        ;Linear in (length xs)??
  (apply append
        (map (λ (x) (if (p x) (list x) '())) xs)))

(define (filter p xs)                        ;Linear in (length xs)!!
  (mapreduce (λ (x) (if (p x) (list x) '()))
             append '() xs))
```

The latter analysis depends on a crucial fact: in this situation, each call to append will require constant, not linear, time!

reverse Using car cdr cons

```
(define (reverse xs)                          ;QUADRATIC in (length xs)
  (cond ((null? xs) '())
        (else (addright (reverse (cdr xs)) (car xs)))))

(define (revappend xs ys)                     ;Linear in (length xs)
  (cond ((null? xs) ys)
        (else (revappend (cdr xs) (cons (car xs) ys)))))

(define (reverse xs)                          ;Linear in (length xs)
  (revappend xs '()))
```

Structural recursion on cons lists produces poor performance for reverse. An accumulation trick gets it down to linear time.

Parallel map reduce mapreduce Using item list split conc

```
(define (mapreduce f g id xs)         ;Logarithmic in (length xs)??
  (cond ((null?  xs) id)
        ((singleton?  xs) (f (item xs)))
        (else (split xs (λ (ys zs)
                (g (mapreduce f g id ys)             ;Opportunity for
                   (mapreduce f g id zs)))))))       ; parallelism

(define (map f xs)
  (mapreduce (λ (x) (list (f x))) append '() xs))   ;or conc

(define (reduce g id xs)
  (mapreduce (λ (x) x) g id xs))
```

Parallel length filter reverse Using item list split conc

```
(define (length xs)                          ;Logarithmic in (length xs)??
  (mapreduce (λ (q) 1) + 0 xs))

(define (filter p xs)                        ;Logarithmic in (length xs)??
  (mapreduce (λ (x) (if (p x) (list x) '()))
             append '() xs))

(define (reverse xs)                         ;Logarithmic in (length xs)??
  (mapreduce list (λ (ys zs) (append zs ys)) '() xs))
```

Exercise: Write Mergesort and Quicksort in This Binary-split Style

- Quicksort: structural induction on output
  - > Carefully split input into lower and upper halves (tricky)
  - > Recursively sort the two halves
  - > Cheaply append the two sorted sublists
- Mergesort: structural induction on input
  - > Cheaply split input in half
  - > Recursively sort the two halves
  - > Carefully merge the two sorted sublists (tricky)

Filters in Fortress (1)

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

$$\quad \mathit{result}\colon \mathrm{List}[\![E]\!] := \langle\,\rangle$$

$$\quad \texttt{for}\ a \leftarrow \mathit{seq}(xs)\ \texttt{do}$$

$$\quad\quad \texttt{if}\ p(a)\ \texttt{then}\ \mathit{result} := \mathit{result}.\mathit{addRight}(a)\ \texttt{end}$$

$$\quad \texttt{end}$$

$$\quad \mathit{result}$$

$$\texttt{end}$$

Example of use:

$$\mathit{odd}\ (x\colon \mathbb{Z}) = \big((x\ \texttt{MOD}\ 2) \neq 0\big)$$

$$\mathit{sequentialFilter}\big(\mathit{odd}, \langle\, 1, 4, 7, 2, 5, 3\,\rangle\big)\ \text{ produces }\ \langle 1, 7, 5, 3\rangle$$

Filters in Fortress (2)

$recursiveFilter[\![E]\!](p\colon E \rightarrow \text{Boolean}, xs\colon \text{List}[\![E]\!])\colon \text{List}[\![E]\!] =$

$\quad$ `if` $xs.isEmpty$ `then` $\langle\,\rangle$

$\quad$ `else`

$\qquad (first, rest) = xs.extractLeft.get$

$\qquad rest' = recursiveFilter(rest, p)$

$\qquad$ `if` $p(first)$ `then` $rest'.addLeft(first)$ `else` $rest'$ `end`

`end`

Still linear-time delay.

Filters in Fortress (3a)

$\mathit{parallelFilter}[\![E]\!](p\colon E \to \mathrm{Boolean}, xs\colon \mathrm{List}[\![E]\!])\colon \mathrm{List}[\![E]\!] =$

$\quad$ `if` $|xs| = 0$ `then` $\langle\,\rangle$

$\quad$ `elif` $|xs| = 1$ `then`

$\quad\quad (x, \_) = xs.\mathit{extractLeft}.\mathit{get}$

$\quad\quad$ `if` $p(x)$ `then` $\langle x \rangle$ `else` $\langle\,\rangle$ `end`

$\quad$ `else`

$\quad\quad (x, y) = xs.\mathit{split}()$

$\quad\quad \mathit{parallelFilter}(x, p) \parallel \mathit{parallelFilter}(y, p)$

`end`

Filters in Fortress (3b)

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

$\quad\texttt{if}\ |xs| = 0\ \texttt{then}\ \langle\,\rangle$

$\quad\texttt{elif}\ |xs| = 1\ \texttt{then}$

$\qquad(x, \_) = xs.extractLeft.get$

$\qquad\texttt{if}\ p(x)\ \texttt{then}\ \langle x\rangle\ \texttt{else}\ \langle\,\rangle\ \texttt{end}$

$\quad\texttt{else}$

$\qquad(x, y) = xs.split()$

$\qquad parallelFilter(x, p) \parallel parallelFilter(y, p)$

$\texttt{end}$

Filters in Fortress (3c)

$$parallelFilter[\![E]\!](p\colon E \to \text{Boolean}, xs\colon \text{List}[\![E]\!])\colon \text{List}[\![E]\!] =$$

$$\quad \texttt{if}\ |xs| = 0\ \texttt{then}\ \langle\,\rangle$$

$$\quad \texttt{elif}\ |xs| = 1\ \texttt{then}$$

$$\qquad (x, \_) = xs.extractLeft.get$$

$$\qquad \texttt{if}\ p(x)\ \texttt{then}\ \langle x \rangle\ \texttt{else}\ \langle\,\rangle\ \texttt{end}$$

$$\quad \texttt{else}$$

$$\qquad (x, y) = xs.split()$$

$$\qquad parallelFilter(x, p) \mathbin{\|} parallelFilter(y, p)$$

$$\quad \texttt{end}$$

$$reductionFilter[\![E]\!](p\colon E \to \text{Boolean}, xs\colon \text{List}[\![E]\!])\colon \text{List}[\![E]\!] =$$

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

Filters in Fortress (4)

Actually, filters are so useful that they are built into the Fortress comprehension notation in the usual way:

$$\mathit{comprehensionFilter}[\![E]\!]\big(p\colon E \to \mathrm{Boolean}, xs\colon \mathrm{List}[\![E]\!]\big)\colon \mathrm{List}[\![E]\!] = \langle x \mid x \leftarrow xs, p(x)\rangle$$

Oh, yes: $\displaystyle\sum_{i\leftarrow 1:1000000} x_i$ and $\displaystyle\mathop{\mathtt{MAX}}_{i\leftarrow 1:1000000} x_i$

or maybe: $\displaystyle\sum_{a\leftarrow x} a$ and $\displaystyle\mathop{\mathtt{MAX}}_{a\leftarrow x} a$

or maybe just: $\sum x$ and $\mathtt{MAX}\, x$

Point of Order

- For `filter`, unlike summation, we rely on maintaining 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 (which is a spatial order) with the order in which they are computed (which is a temporal order).
- Sequential programming often ties the one to the other. Good parallel programming decouples this unnecessary dependency.
- This strategy for parallelism relies only on associativity, *not* commutativity.

To Summarize: A Big Idea

- Summations and list constructors and loops are alike!

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

$$\langle\, x_i^2 \mid i \leftarrow 1:1000000 \,\rangle$$

$$\texttt{for}\ i \leftarrow 1:1000000\ \texttt{do}\ x_i := x_i^2\ \texttt{end}$$

  > Generate an abstract collection

  > The *body* computes a function of each item

  > Combine the results (or just synchronize)

  > In other words: generate-and-reduce

- 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

Another Big Idea

- Formulate a sequential loop (or finite-state machine) as successive applications of state transformation functions f_i
- Find an *efficient* way to compute and represent compositions of such functions (this step requires ingenuity)
- Instead of computing
  $s := s_0; \mathtt{for}\ i \leftarrow seq(1:1000000)\ \mathtt{do}\ s := f_i(s)\ \mathtt{end}$,
  compute $s := (\bigcirc_{i \leftarrow 1:1000000} f_i)\ s_0$
- Because function composition is associative, the latter has a parallel strategy
- If you need intermediate results, use parallel prefix function composition; then map down the result, applying each to s_0

We Need a New Mindset for Multicores

- DO loops are so 1950s! (Literally: Fortran is now 50 years old.)
- So are linear linked lists! (Literally: Lisp is now 50 years old.)
- Java™-style iterators are **so** last millennium!
- Even arrays are suspect! Ultimately, it's all trees.
- As soon as you say "first, SUM = 0" you are hosed. Accumulators are BAD for parallelism. Note that `foldl` and `foldr`, though functional, are fundamentally accumulative.
- If you say, "process subproblems in order," you lose.
- The great tricks of the sequential past DON'T WORK.
- The programming idioms that have become second nature to us as everyday tools DON'T WORK.

The Parallel Future

- We need parallel strategies for problem decomposition, data structure design, and algorithmic 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 subsolutions.
  - The bottom-up view:
    Don't create a null solution, then successively update it;
    Instead, map inputs independently to singleton solutions, then merge the subsolutions treewise.
  - Combining subsolutions is usually trickier than incremental update of a single solution.

MapReduce Is a Big Deal!

- Associative combining operators are a VERY BIG DEAL!
  - Google MapReduce requires that combining operators also be commutative.
  - There are ways around that.
- Inventing new combining operators is a very, very big deal.
  - Creative catamorphisms!
  - We need programming languages that encourage this.
  - We need assistance in proving them associative.

The Fully Engineered Story

In practice, there are many optimizations:

- Optimized representations of singleton lists.
- Use tree branching factors larger than 2. (Example: Rich Hickey's Clojure is a JVM™-based Lisp that represents lists as 64-ary trees.)
- Use self-balancing trees (2-3, red-black, finger trees, ...).
- Use sequential techniques near the leaves.
- Have arrays at the leaves. Decide dynamically whether to process them sequentially or by parallel recursive subdivision.
- When iterating over an integer range, decide dynamically whether to process it sequentially or by parallel recursive subdivision.

Conclusion

- Programs and data structures organized according to linear problem decomposition principles can be hard to parallelize.
- Programs and data structures organized according to parallel problem decomposition principles are easily processed either in parallel or sequentially, according to available resources.
- This parallel strategy has costs and overheads. They will be reduced over time but will not disappear.
- In a world of parallel computers of wildly varying sizes, this is our only hope for program portability in the future.
- Better language design can encourage better parallel programming.

Sukyoung Ryu

sryu@cs.kaist.ac.kr

http://plrg.kaist.ac.kr