The Future Is Parallel: What's a Programmer to Do?

Breaking Sequential Habits of Thought

Guy Steele
Sun Fellow
Sun Microsystems Laboratories
April 2009
Copyright © 2008, 2009 Sun Microsystems, Inc. ("Sun"). All rights are reserved by Sun except as expressly stated as follows. Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted, provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers, or to redistribute to lists, requires prior specific written permission of Sun.
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
This Talk Is 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?
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?

\[ \sum_{i=1}^{1000000} x_i \] or maybe just \[ \sum x \]

Compare Fortran 90 \text{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)
END DO
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))))))
```
Append Lists (1)
Append Lists (2)
Append Lists (3)

append

singleton

null
Append Lists (4)

append

singleton

null
Defining Lists Using car cdr cons

(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 (addrightrt xs a)
  (cond ((null? xs) (list a))
        (else (cons (car xs) (addrightrt (cdr xs) a)))))
Defining Lists Using \texttt{car} \hspace{1mm} \texttt{cdr} \hspace{1mm} \texttt{cons}

\begin{verbatim}
(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)
        ((null? xs) (list a))
        (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))))
\end{verbatim}
map reduce mapreduce

(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

(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 p (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))
reverse

(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 '()))
Linear versus Multiway Decomposition

- Linearly linked lists are inherently sequential.
  - Compare Peano arithmetic: \( 5 = (((((0+1)+1)+1)+1)+1)+1 \)
  - Binary arithmetic is much more efficient than unary!
- We need a multiway decomposition paradigm:
  
  \[
  \begin{align*}
  \text{length} \ [ \ ] & = 0 \\
  \text{length} \ [a] & = 1 \\
  \text{length} \ (a++b) & = (\text{length} \ a) + (\text{length} \ b)
  \end{align*}
  \]

  This is just a summation problem: adding up a bunch of 1’s!
  (More generally: a bunch of 0’s and 1’s.)
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))))
(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 a zs))))))))

(define (addleft a xs) (append (list a) xs))
(define (addright xs a) (append xs (list a)))
Parallel map reduce mapreduce

(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)
                         (mapreduce f g id zs)))))))

(define (map f xs)
  (mapreduce (λ (x) (list (f x))) append ’() xs))

(define (reduce g id xs)
  (mapreduce (λ (x) x) g id xs))
length filter reverse

(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 (yx zs) (append zs ys)) '() xs))
Exercise: Write Mergesort and Quicksort in This Binary-split Style

- Mergesort: structural induction on input
  > Cheaply split input in half
  > Recursively sort the two halves
  > Carefully merge the two sorted sublists (tricky)

- 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
A Modest Example: Filter (1)

```
sequentialFilter[E](a: List[E], p: E → Boolean): List[E] = do
    result: List[E] := ⟨⟩
    for k ← seq(0 # a.size()) do
        if p(a_k) then result := result.addRight(a_k) end
    end
    result
end
```

Example of use:

```
odd(x: Z) = ((x MOD 2) ≠ 0)
sequentialFilter(⟨1, 4, 7, 2, 5⟩, odd)
```

So what language is this? **Fortress.**
A Modest Example: Filter (2)

\[
\text{recursiveFilter}\[E\](a: \text{List}[E], p: E \rightarrow \text{Boolean}): \text{List}[E] =
\]

\[
\text{if } a.\text{isEmpty}() \text{ then } \langle \rangle \\
\text{else}
\]

\[
(first, rest) = a.\text{extractLeft}().\text{get}()
\]

\[
rest' = \text{recursiveFilter}(rest, p)
\]

\[
\text{if } p(first) \text{ then } rest'.\text{addLeft}(first) \text{ else } rest' \text{ end}
\]

end
A Modest Example: Filter (3)

\[
\text{parallelFilter}[E](a: \text{List}[E], \ p: E \rightarrow \text{Boolean}): \text{List}[E] = \\
\quad \text{if} \ |a| = 0 \ \text{then} \ \langle \rangle \\
\quad \text{elif} \ |a| = 1 \ \text{then} \\
\qquad \quad (\text{first}, _) = a.\text{extractLeft()}.\text{get()} \\
\qquad \quad \text{if} \ p(\text{first}) \ \text{then} \ a \ \text{else} \ \langle \rangle \ \text{end} \\
\quad \text{else} \\
\qquad \quad (x, y) = a.\text{split()} \\
\qquad \quad \text{parallelFilter}(x, p) \parallel \text{parallelFilter}(y, p) \\
\quad \text{end}
\]
A Modest Example: Filter (4)

\[
\text{reductionFilter}([E])(a: \text{List}([E]), p: E \rightarrow \text{Boolean}): \text{List}([E]) = \\
\| (\text{if } p(x) \text{ then } [x] \text{ else } [] \text{ end}) \\
\|^{x \leftarrow a}
\]

\[
\langle x \mid x \leftarrow a, p(x) \rangle
\]

Oh, yes: \[
\sum_{i \leftarrow 1:1000000} x_i
\]
Splitting a String into Words (1)

- Given: a string
- Result: 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
Splitting a String into Words (2)

- Tests:
  ```java
  println words("This is a sample")
  println words("Here is another sample")
  println words("JustOneWord")
  println words("")
  println words(""")
  ```

- Expected output:
  ```
  ⟨This, is, a, sample⟩
  ⟨Here, is, another, sample⟩
  ⟨JustOneWord⟩
  ⟨⟩
  ⟨⟩
  ```
Splitting a String into Words (3)

\[ \text{words}(s: \text{String}) = \text{do} \]
\[ \text{result: List[\text{String}]} := \langle \rangle \]
\[ \text{word: String} := "" \]
\[ \text{for } k \leftarrow \text{seq}(0 \# \text{length}(s)) \text{ do} \]
\[ \text{char} = \text{substring}(s, k, k + 1) \]
\[ \text{if } (\text{char} = "") \text{ then} \]
\[ \text{if } (\text{word} \neq "") \text{ then result := result \parallel \langle \text{word} \rangle \text{ end}} \]
\[ \text{word} := "" \]
\[ \text{else} \]
\[ \text{word} := \text{word \parallel char} \]
\[ \text{end} \]
\[ \text{end} \]
\[ \text{if } (\text{word} \neq "") \text{ then result := result \parallel \langle \text{word} \rangle \text{ end} \]
\[ \text{result} \]
\[ \text{end} \]
Splitting a String into Words (4a)

Here is a sesquipedalian string of words

Here is a sesquipedalian string of words

Here is a sesquipedalian string of words
Splitting a String into Words (4b)

Here is a sesquipedalian string of words

Chunk("sesquipedea")
Splitting a String into Words (4c)

Here is a sesquipedalian string of words

Segment(“g”, (“of”), (“words”))
Splitting a String into Words (4d)

Here is a sesquipedalian string of words

Segment("Here", ["is", "a"], ")
Splitting a String into Words (4e)

Here is a sesquipedalian string of words

Segment(“lian”, ⟨⟩, “strin”)
Splitting a String into Words (4f)

Here is a sesquipeda

Here is a sesquipeda

Segment("Here", ["is", "a"], "") Chunk("sesquipeda")

Segment("Here", ["is", "a"], "sesquipeda")
Splitting a String into Words (4g)

Segment("lian", ⟨⟩, "string")
Segment("g", ⟨"of"⟩, "words")
Segment("lian", ⟨"string", "of"⟩, "words")
Splitting a String into Words (4h)

Here is a sesquipedalian string of words

Segment(“Here”, (“is”, “a”, “sesquipedalian”, “string”, “of”), (“words”))
maybeWord(s: String): List[String] =
  if s == "" then ⟨⟩ else ⟨s⟩ end

trait WordState
  extends { Associative[WordState, ⊕] }
  comprises { Chunk, Segment }
  opr ⊕(self, other: WordState): WordState
end
object Chunk(s: String) extends WordState

opr ⊕(self, other: Chunk): WordState =
   Chunk(s || other.s)

opr ⊕(self, other: Segment): WordState =
   Segment(s || other.l, other.A, other.r)

end
Splitting a String into Words (7)

```scala
object Segment(l: String, A: List[String], r: String)
  extends WordState

  opr ⊕(self, other: Chunk): WordState =
  Segment(l, A, r || other.s)

  opr ⊕(self, other: Segment): WordState =
  Segment(l, A || maybeWord(r || other.l) || other.A, other.r)

end
```
Splitting a String into Words (8)

\begin{align*}
\text{processChar}(c: \text{String}): \text{WordState} &= \quad \\
\text{if } (c = " ") \text{ then } \text{Segment}("", \langle \rangle, "") \\
\text{else } \text{Chunk}(c) \\
\text{end}
\end{align*}

\begin{align*}
\text{words}(s: \text{String}) &= \text{do} \\
\quad g &= \bigoplus_{k \leftarrow 0 \# \text{length}(s)} \text{processChar}(\text{substring}(s, k, k + 1)) \\
\quad \text{typecase } g \text{ of} \\
\quad \text{Chunk } \Rightarrow \text{maybeWord}(g.s) \\
\quad \text{Segment } \Rightarrow \text{maybeWord}(g.l) \| g.A \| \text{maybeWord}(g.r) \\
\quad \text{end} \\
\text{end}
\end{align*}
Splitting a String into Words (9)

(* The mechanics of BIG OPLUS *)

\[ \text{opr BIG } \oplus [T](g : (\text{Reduction}[\text{WordState}],
\quad T \rightarrow \text{WordState})
\quad \rightarrow \text{WordState}): \text{WordState} =
\quad g(\text{GlomReduction, identity}[\text{WordState}]) \]

\begin{verbatim}
object GlomReduction extends Reduction[WordState]
  getter toString() = "GlomReduction"
  empty(): WordState = Chunk(""")
  join(a: WordState, b: WordState): WordState = a \oplus b
end
\end{verbatim}
What’s Going On Here?

Instead of linear induction with one base case (empty), we have multiway induction with two base cases (empty and unit).

Why are these two base cases important?
Representation of Abstract Collections

Binary operator ◊
Leaf operator ("unit") □
Optional empty collection ("zero") ε
that is the identity for ◊

Diagram:
- Left diagram:
  - Root: ◊
  - Left child: ε
  - Right child: 4
  - Left child of ε: 2
  - Right child of ε: 3

- Right diagram:
  - Root: ◊
  - Left child: 1
  - Right child: 4
  - Left child of 4: 2
  - Right child of 4: 3
## Algebraic Properties of ♦

<table>
<thead>
<tr>
<th>Associative</th>
<th>Commutative</th>
<th>Idempotent</th>
<th>Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>no</td>
<td>no</td>
<td>no</td>
<td>binary trees</td>
</tr>
<tr>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>weird</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>“mobiles”</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>weird</td>
</tr>
<tr>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>lists (arrays)</td>
</tr>
<tr>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>weird</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>multisets (bags)</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>sets</td>
</tr>
</tbody>
</table>

The “Boom hierarchy”
Associativity

These are all considered to be equivalent.
Catamorphism: Summation

Replace $\triangleleft \square \varepsilon$ with $+$ identity $0$
Computation: Summation

Replace $\diamond \quad \Box \quad \varepsilon$ with $+$ identity $0$

Diagram: Two expressions are shown, one involving operations $\diamond$, $\Box$, and $\varepsilon$, and another involving $+$ operations.

Result: The expression on the right shows the result of replacing $\varepsilon$ with $0$, leading to $10$. 
Catamorphism: Lists

Replace ◊ □ ε with append ⟨-⟩ ⟨⟩

\[
\text{append} \\
\langle 1 \rangle \langle 2 \rangle \langle 3 \rangle \langle 4 \rangle
\]

\[
\langle 1, 2, 3, 4 \rangle
\]
Computation: Lists

Replace $\diamond$ $\Box$ $\varepsilon$ with append $\langle -$ $\rangle$ $\langle -$ $\rangle$

![Diagram showing the computation process](image-url)
Representation: Lists

Replace $\Diamond$ $\Box$ $\varepsilon$ with append $\langle -$ $\rangle$

Diagram:

Left:
- $\Diamond$
- $\Diamond$
- $\Diamond$
- $1$
- $2$
- $3$
- $4$

Right:
- append
- $\langle 1 \rangle$
- $\langle 2 \rangle$
- $\langle 3 \rangle$
- $\langle 4 \rangle$
- $\langle 1, 2, 3, 4 \rangle$
Catamorphism: Loops

Replace $\diamond \square \varepsilon$ with seq identity () or par identity ()
where seq: (,()) → () and par: (,()) → ()

for $i \leftarrow \text{seq}(1:4)$ do print $i$ end

Generators can modify the catamorphism and so control the parallelism.
To Summarize: A Big Idea

- Loops and summations and list constructors are alike!
  
  ```
  for i ← 1:1000000 do  x_i := x_i^2 end
  
  \[ \sum_{i=1}^{1000000} x_i^2 \]
  
  \[ \langle x_i^2 | i \leftarrow 1:1000000 \rangle \]
  ```

- Generate an abstract collection
- The *body* computes a function of each item
- Combine the results (or just synchronize)

- 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 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
  
  $$s := s_0; \text{for } i \leftarrow \text{seq}(1:1000000) \text{ do } s := f_i(s) \text{ end},$$
  
  compute
  
  $$s := (\circ_{i\leftarrow1:1000000} f_i) s_0$$
- Because function composition is associative, the latter has a parallel strategy
- In the “words in a string” problem, each character can be regarded as defining a state transformation function
Splitting a String into Words (3, again)

\[\text{words}(s: \text{String}) = \begin{aligned} &\text{do} \\
&\quad \text{result: List[\text{String}]} := \langle \rangle \\
&\quad \text{word: String} := \text{""} \\
&\quad \text{for } k \leftarrow \text{seq}(0 \# \text{length}(s)) \text{ do} \\
&\quad \quad \text{char} = \text{substring}(s, k, k + 1) \\
&\quad \quad \text{if } (\text{char} = \text{""}) \text{ then} \\
&\quad \quad \quad \text{if } (\text{word} \neq \text{""}) \text{ then result := result || } \langle \text{word} \rangle \text{ end} \\
&\quad \quad \quad \text{word := \""} \\
&\quad \quad \text{else} \\
&\quad \quad \quad \text{word := word || char} \\
&\quad \text{end} \\
&\text{end} \\
&\text{if } (\text{word} \neq \text{""}) \text{ then result := result || } \langle \text{word} \rangle \text{ end} \\
&\text{result} \\
&\text{end} \end{aligned}\]
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.
We Need a New Mindset

- 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!
- As soon as you say “first, SUM = 0” you are hosed. Accumulators are BAD.
- 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 new strategies for problem decomposition.
  - Data structure design/object relationships
  - Algorithmic organization
  - Don’t split a problem into “the first” and “the rest.”
  - Do split a problem into roughly equal pieces.
    Then figure out how to combine general subsolutions.
  - Often this makes combining the results a bit harder.

- We need programming languages and runtime implementations that support parallel strategies
  and hybrid sequential/parallel strategies.

- We must learn to manage new space-time tradeoffs.
Conclusion

- A program organized according to linear problem decomposition principles can be really hard to parallelize.
- A program organized according to parallel problem decomposition principles is easily run either in parallel or sequentially, according to available resources.
- The new 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.
guy.steele@sun.com
http://research.sun.com/projects/plrg
http://projectfortress.sun.com