


ORACLE®



ORACLE®

How Not to Think about Parallel Programming

Guy L. Steele Jr.
Sun Labs, Oracle



Copyright © 2008, 2009, 2010 Oracle Corporation (“Oracle”). All rights are reserved by Oracle 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 Oracle.

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**
- Decades of specialized expertise for science codes and databases and networking
- **But soon NOW general practitioners must go parallel**
- An opportunity to make parallelism easier for everyone

Moore's Law Reinterpreted

- Number of cores per chip doubles every 2 years, while clock speed remains fixed or decreases
- Need to deal with systems with millions of concurrent threads
 - > Future generation will have billions of threads!
- Numbers of threads of execution doubles every 2 years

- Must rethink the design of our software

Quoted from Jack Dongarra, "An Overview of High Performance Computing and Challenges for the Future," HPDC 2009.

JAOO Conference, Autumn 2008

- Anders Hejlsberg: C#
- Guy Steele: Fortress
- Bill Venners: Scala
- Erik Meijer: functional programming

Data Parallelism

- Standard strategies/idioms with parallel implementations:
 - > Broadcast, reduce, permute, parallel prefix
 - > And don't forget “map”!
 - > And more complex operations: sort, join
- When it works, it's great. But SIMD is too confining.
- Often you want “data parallelism of the program counter”
 - > In functional programming terms: want to `map apply`
- With nonlocal side effects, you get race conditions
- The data parallelism mindset is great for *coordinated communication* and making appropriate composable abstractions

Main Points of This Talk

- The best way to write parallel applications is not to have to think about parallelism.
 - > Need for separation of concerns
- The issue is not so much parallelism as *independence*.
- Accumulators are BAD. Divide-and-conquer is GOOD.
 - > An old message, but now we need to take it seriously.
- Certain algebraic properties are very important.
 - > Programmers need help to ensure these properties.
- For debugging, reproduceability is extremely important.
 - > Worth sacrificing performance for (another old message)

What Makes Code Good?

- 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?

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

Compare Fortran 90 SUM(X).

What, not how.

No commitment yet as to strategy. This is good.

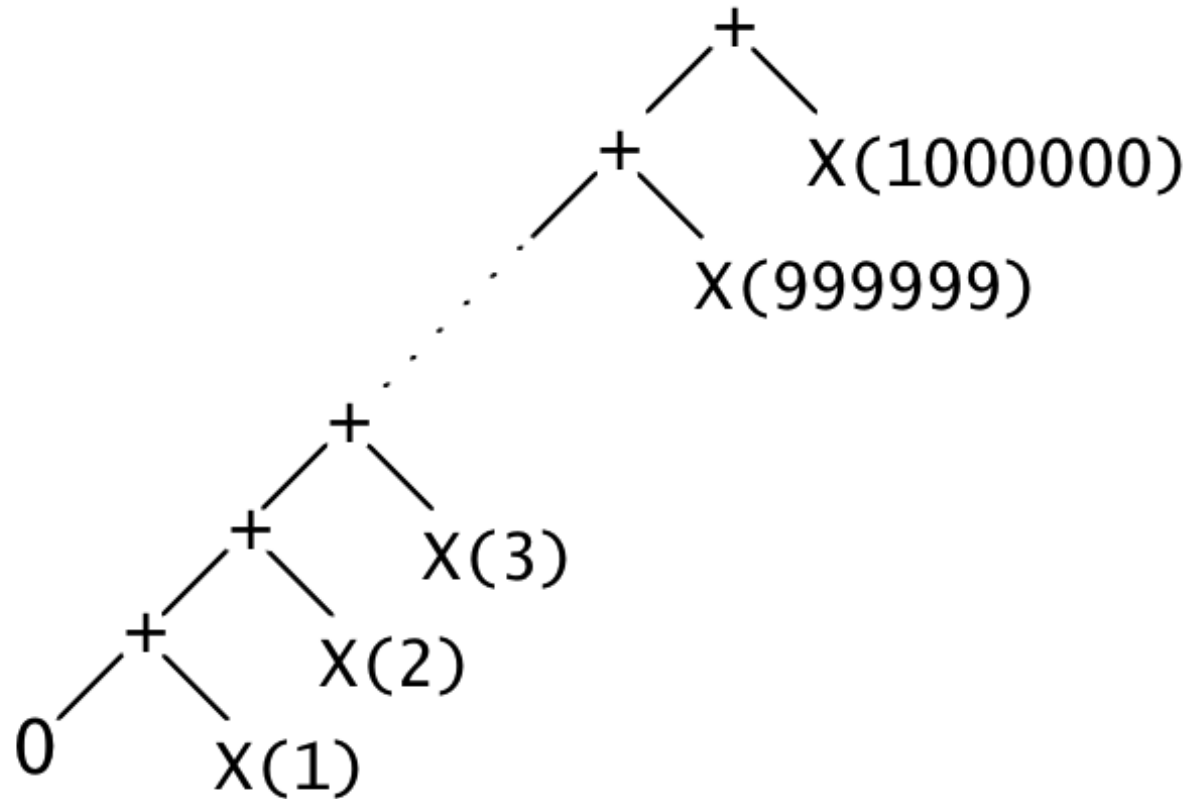
Sequential Computation Tree

```
SUM = 0
```

```
DO I = 1, 1000000
```

```
    SUM = SUM + X(I)
```

```
END DO
```



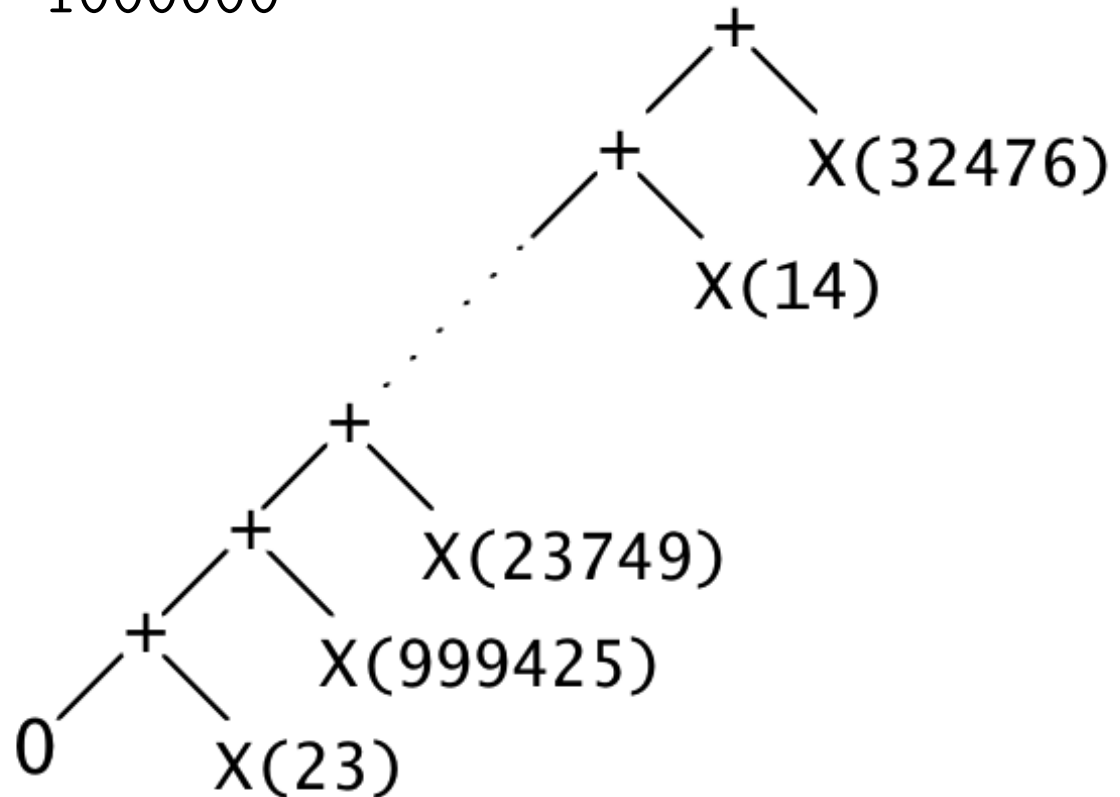
Atomic Update Computation Tree (a)

```
SUM = 0
```

```
PARALLEL DO I = 1, 1000000
```

```
  SUM = SUM + X(I)
```

```
END DO
```



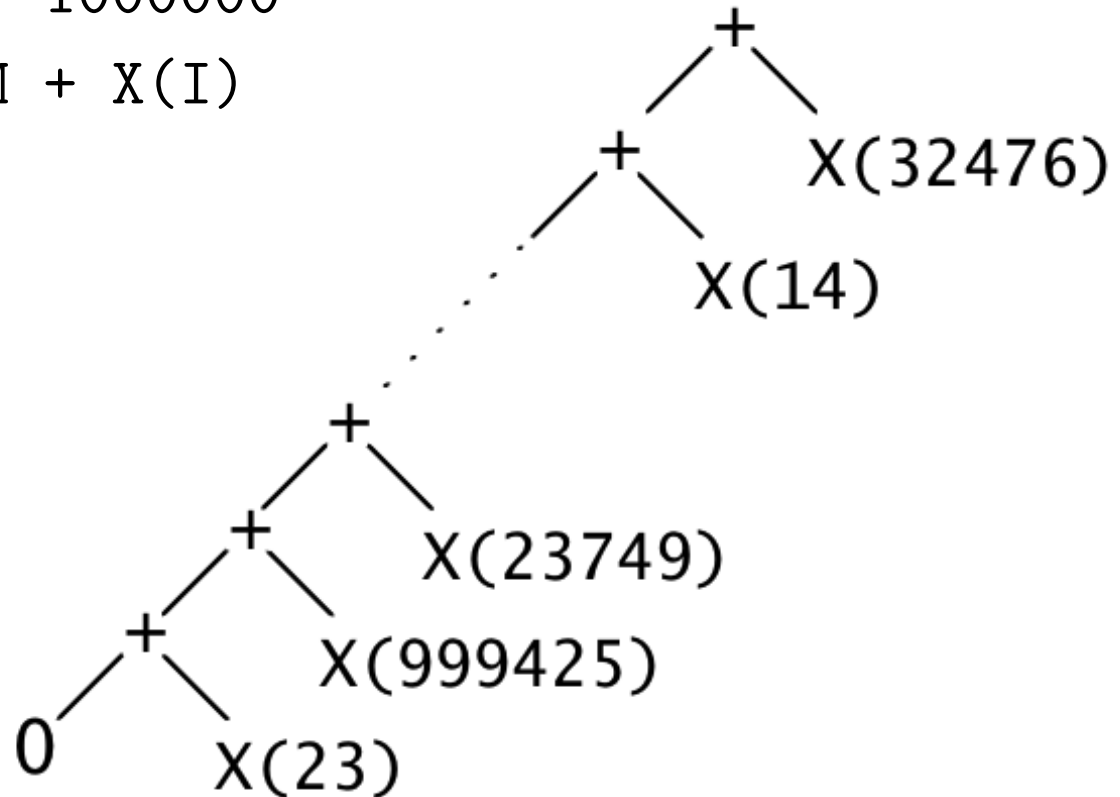
Atomic Update Computation Tree (b)

SUM = 0

PARALLEL DO I = 1, 1000000

 ATOMIC SUM = SUM + X(I)

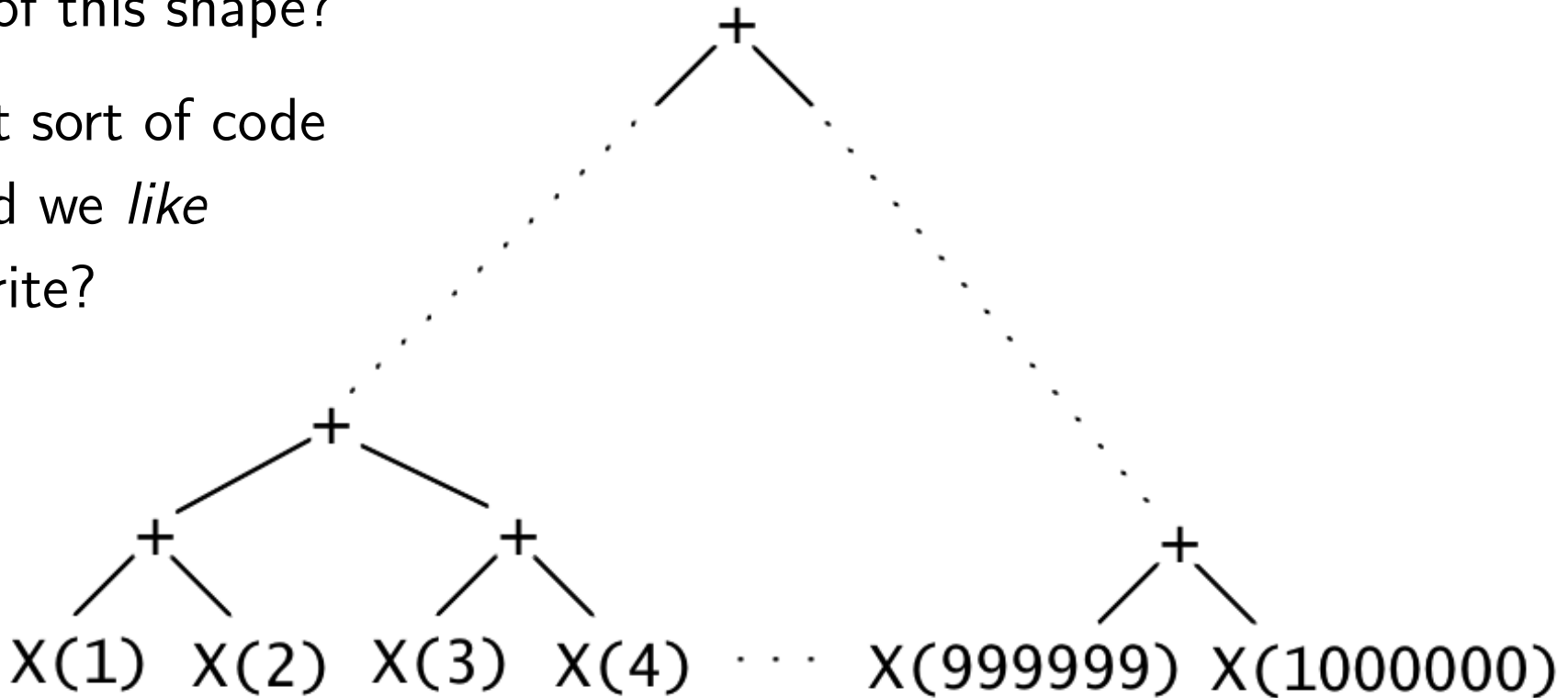
END DO



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?



Communication and Synchronization

- These are important *low-level* concepts
 - > Great for library programmers building the plumbing
 - > We should discourage their use by application programmers
- Why transactional memory is better than locks:
 - > Avoids deadlock? Well, that's nice . . .
 - > Eager (speculative) execution is more efficient? Maybe . . .
 - > The big win: *composable abstractions*
- Side effects are typically used to support accumulation, and accumulation is often the enemy of independence and reproduceability

Accumulation . . .

- Start with an empty solution
- Use each input to incrementally update the solution
 - > The incremental update operator is typically *asymmetric*
- Great if you have committed to using one processor
- Linear time, but really saves space
- Avoids constructing data structures (just use variables)

... vs. Divide-and-Conquer

- From each input construct a *singleton* solution
- Merge solutions (typically pairwise)
- Takes more space, but can be log time
- Intermediate solutions may need to be heap-allocated
- Merge is usually more complicated than incremental update
- *But* merge is typically associative!
 - > Often it is also commutative, but not always
- Identifying this associative combining operator usually lends deeper insight into the problem

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:

```
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)

```
words(s: String) = do
  result: List[[String]] := ⟨ ⟩
  word: String := ""
  for k ← seq(0 # length(s)) do
    char = substring(s, k, k + 1)
    if (char = " ") then
      if (word ≠ "") then result := result || ⟨ word ⟩ end
      word := ""
    else
      word := word || char
    end
  end
  if (word ≠ "") then result := result || ⟨ word ⟩ end
  result
end
```

Splitting a String into Words (4a)

Here is a sesquipedalian string of words

Here is a sesquipedal|ian string of words

Here is a |sesquipedal|ian string| of words

Splitting a String into Words (4b)

Here is a sesquipedalian string of words

```
Chunk("sesquippeda")
```


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", ⟨ ⟩, "string")

Splitting a String into Words (4f)

Here is a sesquippeda

Here is a | sesquippeda

Segment("Here", <"is", "a">, "") \oplus Chunk("sesquippeda")

Segment("Here", <"is", "a">, "sesquippeda")

Splitting a String into Words (4g)

lian string of words

lian string of words

Segment(lian, $\langle \rangle$, strin) \oplus

Segment("g", \langle "of" \rangle , "words")

Segment(lian, \langle "string", "of" \rangle , "words")

Splitting a String into Words (4h)

Here is a sesquipedalian string of words

Segment(
 “Here”,
 ⟨“is”, “a”, “sesquipedalian”, “string”, “of”⟩,
 “words”)

Splitting a String into Words (5)

```
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
```

Splitting a String into Words (6)

```
object Chunk(s: String) extends WordState
  opr  $\oplus$ (self, other: Chunk): WordState =
    Chunk(s || other.s)
  opr  $\oplus$ (self, other: Segment): WordState =
    Segment(s || other.l, other.A, other.r)
end
```


Splitting a String into Words (7)

```
object Segment(l: String, A: List[[String]], r: String)
  extends WordState
  opr  $\oplus$ (self, other: Chunk): WordState =
    Segment(l, A, r || other.s)
  opr  $\oplus$ (self, other: Segment): WordState =
    Segment(l, A || maybeWord(r || other.l) || other.A, other.r)
end
```

Splitting a String into Words (8)

```
processChar(c: String): WordState =  
  if (c = “ ”) then Segment(“”, ⟨ ⟩, “”) else Chunk(c) end
```

```
words(s: String) = do
```

```
   $g = \bigoplus_{k \leftarrow 0 \# \text{length}(s)}$  processChar(substring(s, k, k + 1))
```

```
  typecase g of
```

```
    Chunk  $\Rightarrow$  maybeWord(g.s)
```

```
    Segment  $\Rightarrow$  maybeWord(g.l) || g.A || maybeWord(g.r)
```

```
  end
```

```
end
```

Splitting a String into Words (9)

(* The mechanics of BIG OPLUS *)

```
opr BIG  $\oplus$ [[T]] (g: (Reduction[[WordState], T  $\rightarrow$  WordState)
     $\rightarrow$  WordState): WordState =
    g(GlomReduction, identity[[WordState]])
```

```
object GlomReduction extends Reduction[[WordState]]
    getter toString() = "GlomReduction"
    empty(): WordState = Chunk("")
    join(a: WordState, b: WordState): WordState = a  $\oplus$  b
end
```

Algebraic Properties Are Important!

- Associative
- Commutative
- Idempotent
- Identity
- Zero

Algebraic Properties Are Important!

- Associative: grouping doesn't matter!
- Commutative: order doesn't matter!
- Idempotent: duplicates don't matter!
- Identity: this value doesn't matter!
- Zero: other values don't matter!

Invariants give the implementation *wiggle room*, that is, the freedom to exploit alternate representations and implementations.

In particular, *associativity* gives implementations the necessary wiggle room to use parallelism—*or not*—as resources dictate.

The Big Idea

- Loops and summations and list/set comprehensions are alike!

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

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

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

- > Generate an abstract collection
- > The *body* computes a function of each item (map)
- > Combine the results (or just synchronize) (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 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 $s := (\circ[i \leftarrow 1 : 1000000] f_i) s_0$
- Because function composition is associative (though not commutative), 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)

```
words(s: String) = do
  result: List[[String]] := ⟨ ⟩
  word: String := ""
  for k ← seq(0 # length(s)) do
    char = substring(s, k, k + 1)
    if (char = " ") then
      if (word ≠ "") then result := result || ⟨ word ⟩ end
      word := ""
    else
      word := word || char
    end
  end
  if (word ≠ "") then result := result || ⟨ word ⟩ end
  result
end
```


Automatic Divide-and-Conquer Code

If you can construct *two* sequential versions of a function that is a homomorphism on lists, one that operates left-to-right and one right-to-left, then there is a technique for constructing a divide-and-conquer version automatically.

Morita, K., Morihata, A., Matsuzaki, K., Hu, Z., and Takeichi, M.

“Automatic inversion generates divide-and-conquer parallel programs.”

Proc. 2007 ACM SIGPLAN PLDI, 146-155.

Just derive a weak right inverse function and then apply the Third Homomorphism Theorem. See—it’s easy!

There is an analogous result for tree homomorphisms. Morihata, A., Matsuzaki, K., Hu, Z., and Takeichi, M. “The third homomorphism theorem on trees: Downward and upward lead to divide-and-conquer.” Proc. 2009 ACM SIGPLAN-SIGACT POPL, 177-185.

Full disclosure: the authors of these papers were members of a research group at the University of Tokyo that has had a collaborative research agreement with the Programming Language Research group at Sun Microsystems Laboratories.

MapReduce Is a Big Deal!

- Associative combining operators are a VERY BIG DEAL!
 - > Google MapReduce requires that combining operators also be commutative.
 - > There are workarounds (attach explicit tags, then sort).
 - > But the system really should maintain order for you.
 - > Parallel prefix is an important related concept
- Inventing **new combining operators** is a very, very big deal.
 - > Don't settle for just SUM, PRODUCT, AND, OR, XOR, MIN, MAX
 - > User-defined monoids! 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! (Constant-time indexing is an illusion.)
- As soon as you say “first, SUM = 0” you are hosed.
- Accumulators are BAD. They encourage sequential dependence and tempt you to use nonassociative updates.
- If you say, “process subproblems in order,” you lose.
- The great tricks of the sequential past **WON'T WORK**.
- The programming idioms that have become second nature to us as everyday tools for the last 50 years **WON'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.
 - > But usually it results in deeper understanding.
- 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.

Floating-Point Summation

- Floating-point addition is *not* associative.
(This is the perennial fly in the ointment.)
- How should we handle large floating-point sums?
- Three principles: accuracy, independence, reproducibility
- Ideally, produce a result with just one rounding error overall.
 - > Hardware and software solutions have been proposed.
- If we are not willing to pay that cost, then we should pay the cost at least for reproducibility, at least when debugging!
- Define a *standard binary tree*: add nonoverlapping (starting from left) adjacent pairs, repeat until only one left.
- Much better standard for the future than “sequential order”

Parallelism Is Like Memory Management

- Resource management problems throughout history:
 - > Registers: register allocators
 - > Main memory: overlays, virtual memory, GC heaps
 - > Cache: cache-oblivious algorithms, self-tuning algorithms
- The key is to maintain an invariant that gives the implementation some wiggle room!
- A good programming language or environment aids or enforces those invariants.
- We need to do for processor allocation what garbage collection has done for memory allocation.

Language Design: Medium Term

- Programming languages and related tools that allows experimentation with the necessary abstractions.
- In particular, we need better abstractions of machine organization that will allow efficient software to port across parallel architectures.
- We have made much progress since HPF.

Language Design: Long Term

- Programming languages and related tools that just run programs efficiently when written in an appropriate and supported style.
- I think that style *must* be the divide-and-conquer style.
- We need tools to help programmers maintain the necessary invariants, including associativity.

Conclusion

- A program organized according to linear problem decomposition principles can be really hard to parallelize.
- A program organized according to independence and divide-and-conquer 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 “independent thinking” that allows parallel computers to run programs effectively.

ORACLE®