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

Breaking Sequential Habits of Thought

Guy Steele
Sun Fellow
Sun Microsystems Laboratories
March 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
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

```plaintext
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 `SUM(X)`.

What, not how.

No commitment yet as to strategy. This is good.
SUM = 0
DO I = 1, 1000000
  SUM = SUM + X(I)
END DO
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?
Linear versus Multiway Decomposition

- One problem: linearly organized data structures
  - Length of a list is $1 + \text{(length of rest of list)}$.
  - 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:
  
  $\text{length } [ ] = 0$
  
  $\text{length } [a] = 1$
  
  $\text{length } (a++b) = (\text{length } a) + (\text{length } b)$
  
  This is just a summation problem: adding up a bunch of 1’s!
  
  (More generally: a bunch of 0’s and 1’s.)
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)

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

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

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

```
lian string of words
lian string of words
```

```
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”, “sesquipedialian”, “string”, “of”),
    “words”)
Splitting a String into Words (5)

```scala
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)

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

words(s: String) = do
   g = ⨁_{k←0#length(s)} processChar(substring(s, k, k + 1))
   typecase g of
      Chunk ⇒ maybeWord(g.s)
      Segment ⇒ maybeWord(g.l) || g.A || maybeWord(g.r)
   end
end
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?

Associative combining operators! (append, not cons)
To Summarize: A Big Idea

• Loops and summations and list constructors are alike!

\[
\text{for } i \leftarrow 1:1000000 \text{ do } x_i := x_i^2 \text{ end}
\]

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

\[
\langle x_i^2 \mid i \leftarrow 1:1000000 \rangle
\]

> Generate an abstract collection

> The \textit{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
We Need a New Mindset

- DO loops are so 1950s!
- So are linear linked lists!
- 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.
- 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