Will Software Save Moore's Law?

Guy Steele
Sun Microsystems Laboratories
September 21, 2005

Moore's Law Is Doomed

- **Any** exponential trend is doomed
- Do the math:
  - Assume semiconductor chip ≈ 1 gram
  - Assume "only" 6×10^6 transistors per chip today
  - Assume transistor mass ≥ proton mass
  - Therefore at most 6×10^{23} transistors per chip
  - Therefore at most 10^{17} improvement available
  - If we double every 2 years, that's about a century
- My grandchildren will surely see Moore's Law fail (but maybe they will see a new miracle)

We Do Have Lots of Computing Power

- Software can't overcome Moore's Law
- Wirth's Law is merely a comment about how we have chosen to spend our short-term bonanza
- We can spend computing cycles on:
  - Getting results
  - Safety, security, authentication, privacy
  - Programmer productivity
  - Repeatedly looking up 10,000 plug-ins
  - Eye candy

Cycles Are Like Money

- Cycles today have more value than cycles tomorrow
- In any given situation, there is a discount rate
- Better productivity lets you deploy solutions sooner

- But Moore's Law can overwhelm the discount rate (more on this later)

Fortress Project: Big Idea #1

Make scientific programmers more productive by using a more mathematical notation

Yes, it's been tried many times before, but:

- Now we have Unicode
- Now we have better parsers
- Now we have parametric, polymorphic types
- Now we better understand how to abstract

Simple Example: NAS CG Kernel (ASCII)

```
conjGrad(A: Matrix[/Float/], x: Vector[/Float/]):
        (Vector[/Float/], Float)
  cgit_max = 25
  z: Vector[/Float/] = 0
  r: Vector[/Float/] = x
  p: Vector[/Float/] = r
  rho: Float = r^T r
  for j <- seq(1:cgit_max) do
    q = A p
    alpha = rho / p^T q
    z := z + alpha p
    r := r - alpha q
    rho0 = rho
    rho := r^T r
    beta = rho / rho0
    p := r + beta p
  end
  (z, ||x - A z||)
```

Matrix[/T/] and Vector[/T/] are parameterized interfaces, where T is the type of the elements.

The form x:T=e declares a variable x of type T with initial value e, and that variable may be updated using the assignment operator :=.

Simple Example: NAS CG Kernel (ASCII)

```
conjGrad[/Elt extends Number, nat N,
         Mat extends Matrix[/Elt,N BY N/],
         Vec extends Vector[/Elt,N/]
        /](A: Mat, x: Vec): (Vec, Elt)
  cgitmax = 25
  z: Vec = 0
  r: Vec = x
  p: Vec = r
  rho: Elt = r^T r
  for j <- seq(1:cgit_max) do
    q = A p
    alpha = rho / p^T q
    z := z + alpha p
    r := r - alpha q
    rho0 = rho
    rho := r^T r
    beta = rho / rho0
    p := r + beta p
  end
  (z, ||x - A z||)
```

Here we make conjGrad a generic procedure. The runtime compiler may produce multiple instantiations of the code for various types E.

The form x=e as a statement declares variable x to have an unchanging value. The type of x is exactly the type of the expression e.

Simple Example: NAS CG Kernel (Unicode)

```
conjGrad[[Elt extends Number, nat N,
          Mat extends Matrix[[Elt,N×N]],
          Vec extends Vector[[Elt,N]]
         ]](A: Mat, x: Vec): (Vec, Elt)
  cgit_max = 25
  z: Vec = 0
  r: Vec = x
  p: Vec = r
  ρ: E = r^T r
  do j ← seq(1:cgit_max) do
     q = A p
     α = ρ / p^T q
     z := z + α p
     r := r - α q
     ρ₀ = ρ
     ρ := r^T r
     β = ρ / ρ₀
     p := r + β p
  end do
  return (z, ‖x - A z‖)
```

This would be considered entirely equivalent to the previous version. You might think of this as an abbreviated form of the ASCII version, or you might think of the ASCII version as a way to conveniently enter this version on a standard keyboard.

Simple Example: NAS CG Kernel

$$
\begin{aligned}
&conjGrad\llbracket \text{Elt } \textbf{extends } \text{Number}, \textbf{ nat } \text{N},\\
&\qquad \text{Mat } \textbf{extends } \text{Matrix}\llbracket \text{Elt}, N\times N\rrbracket,\\
&\qquad \text{Vec } \textbf{extends } \text{Vector}\llbracket \text{Elt}, \text{N}\rrbracket\\
&\quad \rrbracket(A:\text{Mat},\ x:\text{Vec}):(\text{Vec},\ \text{Elt})\\
&\quad cgit_{\max} = 25\\
&\quad z:\text{Vec} = 0\\
&\quad r:\text{Vec} = x\\
&\quad p:\text{Vec} = r\\
&\quad \rho:\text{Elt} = r^{\mathrm{T}} r\\
&\quad \textbf{for } j \leftarrow \textbf{seq}(1:cgit_{\max}) \textbf{ do}\\
&\qquad q = A\,p\\
&\qquad \alpha = \frac{\rho}{p^{\mathrm{T}} q}\\
&\qquad z := z+\alpha\,p\\
&\qquad r := r-\alpha\,q\\
&\qquad \rho_0 = \rho\\
&\qquad \rho := r^{\mathrm{T}} r\\
&\qquad \beta = \frac{\rho}{\rho_0}\\
&\qquad p := r+\beta\,p\\
&\quad \textbf{end}\\
&\quad (z\,,\ \|x-A\,z\|)
\end{aligned}
$$

It's not new or surprising that code written in a programming language might be displayed in a conventional math-like format. The point of this example is how similar the code is to the math notation: the gap between the two syntaxes is relatively small. We want to see what will happen if a principal goal of a new language design is to minimize this gap.

Comparison: NAS NPB 1 Specification

$z = 0$
$r = x$
$\rho = r^T r$
$p = r$
DO $i = 1,25$
$q = A\,p$
$\alpha = \rho / (p^T q)$
$z = z + \alpha\,p$
$\rho_0 = \rho$
$r = r - \alpha\,q$
$\rho = r^T r$
$\beta = \rho / \rho_0$
$p = r + \beta\,p$
ENDDO
compute residual norm explicitly: $\|r\| = \|x - A\,z\|$

$z : \mathrm{Vec} = 0$
$r : \mathrm{Vec} = x$
$p : \mathrm{Vec} = r$
$\rho : \mathrm{Elt} = r^{\mathrm{T}} r$
for $j \leftarrow \mathbf{seq}(1 : cgit_{\max})$ **do**
$q = A\,p$
$\alpha = \dfrac{\rho}{p^{\mathrm{T}} q}$
$z := z + \alpha\,p$
$r := r - \alpha\,q$
$\rho_0 = \rho$
$\rho := r^{\mathrm{T}} r$
$\beta = \dfrac{\rho}{\rho_0}$
$p := r + \beta\,p$
end
$(z\,,\ \|x - A\,z\|)$

Comparison: NAS NPB 2.3 Serial Code

```
do j=1,naa+1
   q(j) = 0.0d0
   z(j) = 0.0d0
   r(j) = x(j)
   p(j) = r(j)
   w(j) = 0.0d0
enddo
sum = 0.0d0
do j=1,lastcol-firstcol+1
   sum = sum + r(j)*r(j)
enddo
rho = sum
do cgit = 1,cgitmax
   do j=1,lastrow-firstrow+1
      sum = 0.d0
      do k=rowstr(j),rowstr(j+1)-1
         sum = sum + a(k)*p(colidx(k))
      enddo
      w(j) = sum
   enddo
   do j=1,lastcol-firstcol+1
      q(j) = w(j)
   enddo
   do j=1,lastcol-firstcol+1
      w(j) = 0.0d0
   enddo
   sum = 0.0d0
   do j=1,lastcol-firstcol+1
      sum = sum + p(j)*q(j)
   enddo
   d = sum
   alpha = rho / d
   rho0 = rho
   do j=1,lastcol-firstcol+1
      z(j) = z(j) + alpha*p(j)
      r(j) = r(j) - alpha*q(j)
   enddo
   sum = 0.0d0
   do j=1,lastcol-firstcol+1
      sum = sum + r(j)*r(j)
   enddo
   rho = sum
   beta = rho / rho0
   do j=1,lastcol-firstcol+1
      p(j) = r(j) + beta*p(j)
   enddo
enddo
do j=1,lastrow-firstrow+1
   sum = 0.d0
   do k=rowstr(j),rowstr(j+1)-1
      sum = sum + a(k)*z(colidx(k))
   enddo
   w(j) = sum
enddo
do j=1,lastcol-firstcol+1
   r(j) = w(j)
enddo
sum = 0.0d0
do j=1,lastcol-firstcol+1
   d   = x(j) - r(j)
   sum = sum + d*d
enddo
d = sum
rnorm = sqrt( d )
```

Fortress Project: Big Idea #2

Make application programmers more productive with extensive libraries of good abstractions for scientific programming

- Matrices and vectors, not just arrays
- Dense and sparse
- Intervals
- Need to support notation, not just data structures

Fortress Project: Big Idea #3

The language needed by library coders is different from the language for application programmers

- Defining abstractions is more complex than using them
- Control over implementation details
- Control over parallelism
- Control over floating-point behavior
- Control over notation
- Generality, with control over special cases
- Code factoring and re-use are much more important

Fortress Project: Big Idea #4

A small team can't do the whole job

- Define a compiler language that mostly consists of general facilities for writing libraries
- Push most of the language definition into libraries
- Give libraries most of the control over syntax
- Include a component system for library mix-and-match
- **Use static type analysis and dynamic compilation to reduce the overhead of abstraction**

It's Actually Two Languages

- A framework for building mathematical languages for large, possibly parallel computers
- A specific set of libraries that define a specific set of notations and facilities
- We hope that others will provide additional libraries and notations for specific application areas

Old Joke: How to Solve an NP-Hard Problem of Size N

- Write a program that solves the problem in time $k \cdot 2^N$ on your current computer (we know how to do this)
- Wait 2N years
- Buy a new computer
- By Moore's Law (?!), it runs the same program in time k
- Voilà: you've solved the problem in time 2N+k = O(N)

- This ignores "discount rate" issues
- But: is deploying sooner *always* better?

When the Crunch Comes . . .

- We will have some hard(er) decisions to make
- They will be economic in nature
- Productivity will become much more important
- As for using portable languages on oddball devices:
  - > Portability, like productivity, has benefits and costs
  - > It can be done; sometimes it's worth it
  - > Fortress is not currently aimed at cell phones

guy.steele@sun.com