PROGRAMMING FOR EXASCALE

Andrew Pochinsky

avp@mit.edu

DRAMATIS PERSONÆ

avp@mit.edu
ssyritsyn@lbl.gov
osborn@alcf.anl.gov

WHY

- Problem space
- Computers
- Software

WHY

Physics:

Intrinsically parallel

Local interactions

Multiscale (explicit or implicit)

WHY

Computers:

Parallel systems: no single-CPU machine in Top500

Memory and networks are bandwidth limited

Floating point is (almost) free if you are willing to go to exotic designs

Architectures changing fast

WHY

Software:

Languages designed in 1950-70s

Support for parallel architectures via libraries

Prone to conflicts and race conditions

WHY

Efficient, safe, easy to use --- pick one.

GOALS

A. Ease of use.

B. Efficiency.

C. Safety.

D. Quick turnaround.

E. Isolation of physics from h/w details in the code.

F. Integration with external libraries.

G. Portability.

BASE

Lua is a simple yet powerful scripting language with

- Modern design
- Adequate feature set
- Embeddable
- Small footprint
- Good support for extensions

Lua: values

```
-- comments start with two dashes
3.1415926, 2.7182812828  -- numbers
"Hello World\n" -- strings
{ 1, 2, 3, -4, "message", id = 42}  -- arrays, a.k.a. tables
function (x) return x + 2 end -- functions
function (x)
   z = x * 2
   return function (y)
                return z + y
             end
end
```

Lua: names vs. values

```lua
require "stdlib"

x = 2.7182818284950
printf("x is a number: %12.10f\n", x)
x = "Hello World!"
printf("x is now a string: %s\n", x)

-- output:
-- x is a number: 2.7182818285
-- x is now a string: Hello World!
```

Lua: dynamic typing

Types in lua are associated with values, not names.

Convenient for putting pieces together in a script, makes life difficult for compilers. But there is no compiler in lua.

There is a price: some mistakes are not detected until code is executed.

Lua: functions

- Are first class:

  can be bound to names,

  can be passed as arguments,

  can be returned as values

- Are full lexical closures:

  can refer and modify lexically visible bindings

Lua: functions

```
require "stdlib"
function make_counter()
  local x = 0 -- the counter
  return { inc = function() x = x + 1; end, -- count up
           dec = function() x = x - 1; end, -- count down
           value = function () return x; end } -- get the counter
end
c1 = make_counter()
c2 = make_counter()
c1.inc(); c1.inc(); c2.dec()
printf("c1 = %d\n", c1.value())
printf("c2 = %d\n", c2.value())
-- output:
-- c1 = 2
-- c2 = -1
```

Lua: generic functions

Dynamic types and first class functions together make every function generic: as long as computation make sense for given arguments, the code will work: no special effort is required (no templates, no virtual methods, no mess.)

Lua: summary

- Dynamically typed
- First class functions
- Garbage collected
- Lexically scope, indefinite extend

LUA + QCD = QLUA

Lattice QCD with Lua:

Six Easy Pieces

Qlua: parallel objects

Parallel objects live on a *lattice* which defines the number of dimensions and extends in each directions:

```
exaLattice = qcd.lattice {256, 256, 256, 1024}
smallLat = qcd.lattice { 8, 9, 10 }
```

Parallel values are associated with a lattice, e.g.,

```
x = smallLat:Int(42)
y = smallLat:pcoord(2)
z = smallLat:Read(x + y)
z[{1, 2, 3}] = 3.141592
```

Qlua: QCD data types

Integer	`exaLattice:Integer(...)`
Real	`exaLattice:Real(...)`
Complex	`exaLattice:Complex(...)`
Random Number	`exaLattice:RandomState(...)`
Color Vector	`exaLattice:ColorVector(...),` `exaLattice:ColorVectorN(Nc, ...)`
Gauge Transform	`exaLattice:ColorMatrix(...), ...`
Fermion	`exaLattice:DiracFermion(...), ...`
Propagator	`exaLattice:DiracPropagator(...), ...`

Qlua: parallel operations

```
L = qcd.lattice{4,6,8,7}
x = L:Real(...)
u = L:ColorMatrixN(7,...)
f1 = L:DiracFermionN(7,...)
f2 = L:DiracFermionN(7,...)
g = x * f1 + u * f2
ng = qcd.dot(g, g) -- a lattice complex
ngScalar = ng:real():sum() -- scalar
```

Qlua: parallel shifts

Computing the plaquette:

```
function plaq(U, i, j)
   local U12 = U[i+1]*U[j+1]:shift(i, "from_forward")
   local U34 = U[j+1]*U[i+1]:shift(j, "from_forward")
   return U12*U34:adjoin()
end
```

Qlua: subsets

```
L = qcd.lattice{6,6}
x = L:Int(0)
```

0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0

Qlua: subsets

```
L = qcd.lattice{6,6}
x = L:Int(0)
L:Subset("odd"):
   where(function () x:set(L:Int(1)) end)
```

1	0	1	0	1	0
0	1	0	1	0	1
1	0	1	0	1	0
0	1	0	1	0	1
1	0	1	0	1	0
0	1	0	1	0	1

Qlua: subsets

```
L = qcd.lattice{6,6}
x = L:Int(0)
L:Subset("odd"):
   where(function () x:set(L:Int(1)) end)
L:Subset(axis=1, position = 2):
   where(function() x:set(-L:pcoord(0)) end)
```

1	0	1	0	1	0
0	1	0	1	0	1
1	0	1	0	1	0
0	-1	-2	-3	-4	-5
1	0	1	0	1	0
0	1	0	1	0	1

Qlua: reductions

```
r = L:Real(...)
-- compute a global sum across the lattice
gr = c:sum() -- a single complex number
-- compute a set of sums across the lattice
xr = c:sum(L:MultiSet(L[0], L:pcoord(0))
for i = 0, L[0] - 1 do
   printf("%5d: %10.7f\n", i, xr[i])
end
```

STATUS

- Production use
- Algorithm development, prototyping and test of L3 implementations
- Included L3 routines: Möbius, Clover, QDPOP
- Tutorials on the web
- Code size: 25k lines in 78 files
- L3 glue: about 1,000 lines of bookkeeping per library

CONCLUSIONS

- Scripting: good, bad or ugly?

CONCLUSIONS

- Scripting: good, bad and ugly

  not the solution, but a step toward a better world

CONCLUSIONS

- Scripting: good, bad or ugly?

  not the solution, but a step toward a better world

- Supercomputing could be fun

REFERENCES

- https://usqcd.lns.mit.edu/wiki/QLUA
- https://lattice.lns.mit.edu/trac/downloads
- http://usqcd.org/
- http://www.lua.org/