Scripting HALMD with Lua and Luabind

Peter Colberg

University of Toronto

May 4th, 2011

What is HALMD?

A Highly Accelerated Large-scale Molecular-Dynamics package.

- first „monolithic" version written during physics diploma thesis
  - Lennard-Jones fluid simulation (3D and real 2D)
  - 80-fold speed-up on NVIDIA G200 series GPUs
  - numerically stable over 10^8 steps with double-single precision
  - on-the-fly computation of correlation functions
  - HDF5 trajectory and correlation function output
- modular rewrite of HALMD as tool for PhD thesis research
  - multiple concurrent potentials, integrators, ...
  - self-contained C++ modules for GPU or CPU
  - module coupling with a dynamic language
  - run-time selection of modules
  - command-line options for simple simulations (Lennard-Jones fluid)
  - scripting for complex simulations (biochemical systems)

What is Lua?

An extensible embedded language.

- ideal as a *lightweight* scripting engine for C/C++ software
- simple, self-descriptive, fast to learn syntax
- well-written *complete* user guide and reference manual
- small!
  - liblua.a (Lua interpreter and standard libs) is **172K** on `x86_64`
- highly portable, runs on any ANSI C platform
  - e.g. the SciNet TCS cluster (AIX on POWER6)
- one of the fastest among the dynamically typed languages
  - "close to C" performance with LuaJIT just-in-time compiler
- supports object-oriented programming
  - or functional programming, declarative programming, ...

What is Luabind?

A library for language binding between Lua and C++.

- exposes C++ functions and classes to Lua
- Lua bindings written directly in C++
- supports overloading, operators, multiple inheritance
- convert between custom C++ and native Lua types
- automatically casts between base and derived types
- supports smart pointers and smart pointer casts
  - very useful for loose, safe coupling of C++ modules
- extend C++ classes in Lua, including virtual functions
- works with any compiler adhering to C++ standard
  - GCC, Clang, Intel, XL C++ on POWER6/AIX

Lua values

Lua has eight value types:

- nil (`nil`)
- boolean (`true` or `false`)
- number
  - double-precision floating-point, single-precision or integer
- string
- function
  - functions are first-class values
- userdata
  - C structs, C++ classes
- thread
- **table**
  - may be used for arrays, lists, maps, objects, . . .

Lua functions

Our first Lua program.

```lua
function greet(whom)
    print("Hello, " .. whom .. "!")
end

greet("World") -- Hello, World!
```

Lua is indifferent to white space.

```lua
function greet(whom) print("Hello, " .. whom .. "!") end greet("World")
-- Hello, World!
```

Lua functions

How are functions “first class” values?

```
greet = function(whom)
    print("Hello, " .. whom .. "!")
end
greet("World") -- Hello, World!

greet = print
greet("World") -- World!

greet = function() end
greet("World") --

greet = function(...) print(...) end
greet("Hello", "World") -- Hello   World
```

Lua closures

Lua supports closures i.e. lexical scoping.

```
function accumulate()
    local count = 0
    return function()
      count = count + 1
      return count
    end
end

acc = accumulate()
print(acc()) -- 1
print(acc()) -- 2
print(acc()) -- 3
```

Lua closures

Lua supports closures i.e. lexical scoping.

```
function accumulate(step)
    local count = 0
    return function()
      count = count + step
      return count
    end
end

acc = accumulate(14)
print(acc()) -- 14
print(acc()) -- 28
print(acc()) -- 42
```

Lua tables

Arrays

```lua
t = { 1, 2, 7, 6 }
print(t[3] * t[4]) -- 42

for i, v in ipairs(t) do
    print(i, v)
end
-- 1, 1
-- 2, 2
-- 3, 7
-- 4, 6
```

Lua tables

Arrays. . . ?

```lua
t = { [3] = 7, [4] = 6 }
print(t[3] * t[4]) -- 42
print(t[2]) -- nil
t[2] = 2
print(t[2]) -- 2

for i, v in ipairs(t) do
    print(i, v)
end
--

for i, v in pairs(t) do
    print(i, v)
end
-- 2, 2
-- 3, 7
-- 4, 6
```

Lua tables

```lua
t = {
    alpha = 3.3,
    beta = 4,
    gamma = 42.,
}
print(t["alpha"]) -- 3.3
print(("beta = %.3f"):format(t["beta"])) -- beta = 4.000
print(t.gamma) -- 42

for k, v in pairs(t) do
    print(k, v)
end
-- beta    4
-- alpha   3.3
-- gamma   42
```

Lua tables

```lua
i = {}
j = {}
t = { [4] = 5, four = 6, ["cuatro"] = 7, [i] = 8, [j] = 9, [{}] = 10}
print(t[4]) -- 5
print(t.four) -- 6
print(t.cuatro) -- 7
print(t[i]) -- 8
print(t[j]) -- 9
print(t[{}]) -- nil

for k, v in pairs(t) do
    print(k, v)
end
-- cuatro  7
-- table: 0x818b6f0        9
-- four    6
-- 4       5
-- table: 0x818bd78        10
-- table: 0x818b620        8
```

Lua scoping

Global versus local variables

```lua
function global_variable()
    i = 42
    return i
end
function local_variable()
    local j = 42
    return j
end

print(i) -- nil
print(j) -- nil

global_variable()
print(i) -- 42
local_variable()
print(j) -- nil
```

Object-oriented programming with Lua

```lua
Accumulator = {}

function Accumulator.new()
    local self = {}

    self._value = 0

    function self.accumulate(value)
        self._value = self._value + value
        return self._value
    end

    function self.value()
        return self._value
    end

    return self
end
```

Object-oriented programming with Lua

```lua
acc = Accumulator.new()
print(acc.accumulate(7)) -- 7
print(acc.accumulate(3)) -- 10
print(acc.accumulate(32)) -- 42
print(acc.value()) -- 42

acc = Accumulator.new()
print(acc.accumulate(1)) -- 1
print(acc.accumulate(1)) -- 2
print(acc.value()) -- 2
```

Lua metatables

```lua
Accumulator = {}

function Accumulator.new()
    local self = setmetatable({}, { __index = Accumulator })
    self._value = 0
    return self
end

function Accumulator.accumulate(self, value)
    self._value = self._value + value
    return self._value
end

function Accumulator.value(self)
    return self._value
end

setmetatable(Accumulator, { __call = Accumulator.new })
```

Lua metatables

```lua
acc = Accumulator()
print(acc:accumulate(7)) -- 7
print(acc:accumulate(3)) -- 10
print(acc:accumulate(32)) -- 42
print(acc:value()) -- 42

acc = Accumulator()
print(acc:accumulate(1)) -- 1
print(acc:accumulate(1)) -- 2
print(acc:value()) -- 2
```

HALMD Luabind example

```
/** halmd::mdsim::particle */
template <int dimension>
class particle
{
public:
    static void luaopen(lua_State* L);

    particle(std::vector<unsigned int> const& particles);
    virtual ~particle() {}

    /** number of particles in simulation box */
    unsigned int const nbox;
    /** number of particle types */
    unsigned int const ntype;
    /** number of particles per type */
    std::vector<unsigned int> const ntypes;
};
```

HALMD Luabind example

```
template <int dimension>
void particle<dimension>::luaopen(lua_State* L)
{
    using namespace luabind;
    static string class_name("particle_"
                             + lexical_cast<string>(dimension) + "_");
    module(L, "libhalmd")
    [
        namespace_("mdsim")
        [
            class_<particle, shared_ptr<particle> >(class_name.c_str())
                .def_readonly("nbox", &particle::nbox)
                .def_readonly("ntype", &particle::ntype)
                .def_readonly("ntypes", &particle::ntypes)
        ]
    ];
}
```

HALMD Luabind example

```
/** halmd::mdsim::gpu::particle */
template <int dimension>
class particle
  : public mdsim::particle<dimension>
{
public:
    typedef mdsim::particle<dimension> _Base;
    typedef utility::gpu::device device_type;

    static void luaopen(lua_State* L);

    particle(
        boost::shared_ptr<device_type> device
      , std::vector<unsigned int> const& particles
    );

    /** positions, types */
    cuda::vector<float4> g_r;
    /** ... */
};
```

HALMD Luabind example

```
template <int dimension>
void particle<dimension>::luaopen(lua_State* L)
{
    using namespace luabind;
    static string class_name("particle_"
                             + lexical_cast<string>(dimension) + "_");
    module(L, "libhalmd")
    [
        namespace_("mdsim")
        [
            namespace_("gpu")
            [
                class_<particle, shared_ptr<_Base>, _Base>(class_name.c_str())
                    .def(constructor<
                        shared_ptr<device_type>
                      , vector<unsigned int> const&
                    >())
            ]
        ]
    ];
}
```

HALMD Lua example

```
require("halmd.modules")
require("halmd.device")
require("halmd.mdsim.core")
-- grab modules
local device = halmd.device
local mdsim = halmd.mdsim
-- grab C++ wrappers
local particle_wrapper = {
    host = {
        [2] = libhalmd.mdsim.host.particle_2_
      , [3] = libhalmd.mdsim.host.particle_3_
    }
  , [2] = libhalmd.mdsim.particle_2_
  , [3] = libhalmd.mdsim.particle_3_
}
if libhalmd.mdsim.gpu then
    particle_wrapper.gpu = {
        [2] = libhalmd.mdsim.gpu.particle_2_
      , [3] = libhalmd.mdsim.gpu.particle_3_
    }
end
```

HALMD Lua example

```lua
module("halmd.mdsim.particle", halmd.modules.register)

--
-- construct particle module
--
function new(args)
    local core = mdsim.core() -- singleton
    local dimension = assert(core.dimension)
    local npart = args.particles or { 1000 } -- default value

    if not device() then
        return particle_wrapper.host[dimension](npart)
    end
    return particle_wrapper.gpu[dimension](device(), npart)
end
```

HALMD Lua example

```lua
-- assemble module options
function options(desc)
    desc:add("particles", po.uint_array(), "number of particles")
end

-- read module parameters from HDF5 group
function read_parameters(args, group)
    args.particles = group:read_attribute("particles", h5.uint_array())
end

-- write module parameters to HDF5 group
function write_parameters(particle, group)
    group:write_attribute("particles", h5.uint_array(), particle.ntypes)
end
```

HALMD Lua example

```
require("halmd.modules")

-- command-line options override H5MD file parameters
require("halmd.option")
require("halmd.parameter")

require("halmd.mdsim.core")
require("halmd.mdsim.particle")

-- grab modules
local mdsim = halmd.mdsim

module("halmd", halmd.modules.register)

--
-- Construct simulation
--
function new(args)
    local core = mdsim.core() -- singleton
    core.particle = mdsim.particle()
```

Caveats

Why would I *not* want to use Lua? You...

- are writing a scientific program from scratch.
- seek an "all batteries included" dynamic language.
- feel uncomfortable not swimming along the main stream.
- wish to use numerical routines at the scripting level.
- need *predefined* object-oriented programming constructs.

Lua: a beautiful language

Why would I want to use Lua? You...

- wish to extend an existing C/C++ program.
- seek an embeddable, *lightweight* scripting engine.
- value comprehensive, well-written documentation *in one place*.
- appreciate a well-engineered programming language.
- need a scripting interface for non-programmer users.
- aim to develop a domain-specific language.

Acknowledgements

- Professor Raymond Kapral (PhD thesis supervisor)
  - University of Toronto, Ontario, Canada

- Dr. Felix Höfling (HALMD collaborator)
  - Max Planck Institute for Metals Research, Stuttgart, Germany

References

- Lua, `http://www.lua.org/`
- Programming in Lua, `http://www.lua.org/pil/`
- Lua 5.1 Reference Manual, `http://www.lua.org/manual/5.1/`
- Luabind, `http://www.rasterbar.com/products/luabind.html`
- LuaJIT, `http://luajit.org/`
- HALMD, `http://halmd.org/`
- Comp. Phys. Comm. **182**, 1120 (2011)