High-performance Technical Computing with Erlang

Alceste Scalas Giovanni Casu Piero Pili

Center for Advanced Studies, Research and Development in Sardinia

ACM ICFP 2008 — Erlang Workshop
September 27th, 2008
Victoria, BC, Canada

Introduction

High-performance Technical Computing (HPTC): use of parallel machines or clusters of interconnected computers for executing massive scientific and numerical applications (like physical simulations).

Standard tools:

C/C++/Fortran as core languages

- **BLAS** *(Basic Linear Algebra Subprograms)* for number crunching
- **MPI** *(Message Passing Interface)* for process distribution and IPC
- **PVM** *(Parallel Virtual Machine)* as an alternative to MPI

In the context of a CRS4 research project, we had the task of **developing a real-time HPTC framework** with the following requirements:

distributed VM for network-transparent IPC

minimized data copying for handling large numeric arrays without wasting memory bandwidth

process migration between cluster nodes, allowing to balance the workload in run-time

fault tolerance for long-running simulations based on hardware interfaces (sensors, actuators, etc.)

code reuse should be *easy* and *efficient*, in order to recycle existing numerical routines written in Fortran or C/C++

Could Erlang/OTP help us?

Sure it could!

distributed VM — **yes**, distributed Erlang does it

minimized data copying — **yes**, with reference-counted binaries

process migration — **yes**, even if not *out-of-the-box*

fault tolerance — **yes**, see the `supervisor` behaviour

code reuse — **no**: linked-in drivers are *not easy enough*

So, why not give Erlang a chance?

A Foreign Function Interface for Erlang

The ability to **call existing numerical code**, written in C/C++ or Fortran, in an **easy** and **efficient** way, is **crucial**

Linked-in drivers are *not* a solution, because

1. binding libraries with **tens or hundreds of functions** is cumbersome and error-prone (but tools like EDTK or Dryverl give some help)
2. even with these tools, deciding to **rewrite a time-critical routine in C** is far from trivial
3. the **data (de)serialization** required by linked-in drivers may increase latencies

That's why we decided to develop our own
Foreign Function Interface (FFI) for Erlang

Our Erlang FFI is detailed in **EEP 7**, and **requires some changes to the Erlang VM**

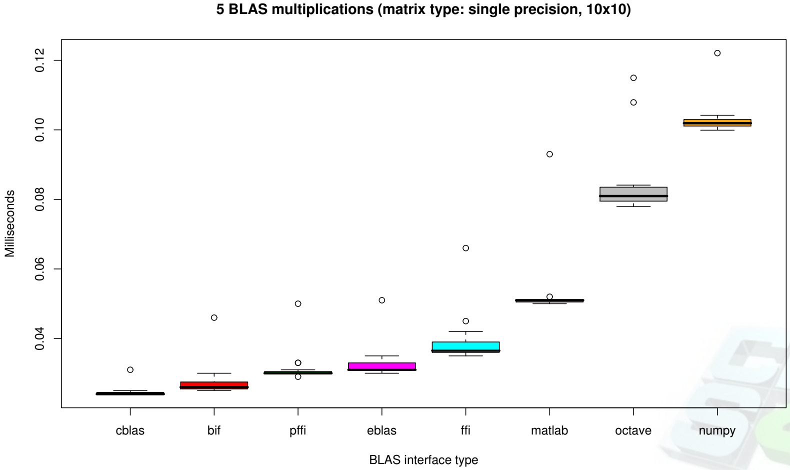
It performs **on-the-fly type conversion** between Erlang and C

```
ok = erl_ddll:load_library("/lib", libc),
Port1 = open_port("libc"),

Pointer1 = ffi:raw_call(Port1,
                        {malloc, 1024},
                        {pointer, size_t}),
ok = ffi:raw_call(Port1,
                  {free, Pointer1},
                  {void, pointer}).
```

It also allows to **preload function symbols** and **precompile function signatures**, thus reducing function call overhead

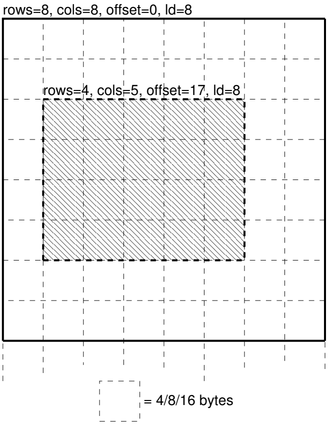
```
ok = erl_ddll:load_library("/lib", libc,
                          [{preload,
                            [{puts, {sint, nonnull}},
                             {putchar, {sint, sint}},
                             {malloc, {nonnull, size_t}},
                             {free, {void, nonnull}}]}]),
Port2 = open_port("libc"),

Pointer2 = ffi:raw_call(Port2, {3, 1024}),
ok = ffi:raw_call(Port2, {4, Pointer2}).
```

What about Erlang FFI performance?



Ubuntu™ 7.10, Intel™ Pentium™ 4, 2800 Mhz, 1 GB RAM, ATLAS 3.6.0 optimized for SSE2

A BLAS binding for Erlang



The BLAS API is quite peculiar:

$$\text{sgemm():} \quad C \leftarrow \alpha AB + \beta C$$

$$\text{saxpy():} \quad Y \leftarrow \alpha X + Y$$

Our BLAS binding for Erlang must:

- provide a **functional API**
- implement an **easy-to-use procedural interface**
- offer a **direct mapping to BLAS**
- handle BLAS vectors and matrices as **standard Erlang terms**
- handle different **memory layouts**

We implemented matrices and vectors as Erlang records

```
-record(matrix, {
    type,   % Atom: s, d, c, z
    rows,   % Number of rows
    cols,   % Number of columns
    ld,     % Leading dimension
    trans,  % Transposition indicator
    offset, % Offset from beginning of binary data
    data    % Refcounted binary with matrix data
  }).

-record(vector, {
    type,   % Atom: s, d, c, z
    length, % Number of elements
    inc,    % Distance between elements
    trans,  % Transposition indicator
    offset, % Offset from beginning of binary data
    data    % Refcounted binary with vector data
  }).
```

What does the Erlang BLAS API look like?

```
blas:init(),

%% Create a 3x3 identity matrix
I = blas:eye(s,     % Precision: 's'ingle or 'd'ouble
             3),    % Rows and columns
V = blas:vector(s, 3, [1.0, 2.0, 3.0]),

%% Functional API example: blas:mul/2
V2 = blas:mul(blas:mul(2.0, I), V),
VL = blas:to_list(blas:transpose(V2)),
%% VL is:
VL = [[2.00000,4.00000,6.00000]],

%% Procedural API example: blas:mul/3
VTarget = blas:vector(s, 3), % Random data
ok = blas:mul(blas:mul(2.0, I), V, VTarget),
VL = blas:to_list(blas:transpose(VTarget)).
%% VTarget has been overwritten, thus matching VL
```

A Matlab™-style language

The BLAS binding, taken alone, has some shortcomings:

1. it's quite **verbose**
2. it leaves all the **optimizations in the hands of the developer**
   - example: *given a sequence of operations, is it safe to use the procedural API to recycle existing matrices for intermediate results, thus minimizing allocations?*
3. our project is aimed at **physicists and engineers that are accustomed to languages like Matlab™ or GNU Octave**, and cannot be expected to learn Erlang

We have thus developed **Matlang**, a Matlab^{TM}-style language that compiles into Core Erlang

Since multiple assignments to the same variable cannot be translated directly, the compiler performs two steps:

1. the Matlang parse tree is converted into **SSA (Static Single Assignment) form**
2. the SSA form is compiled into Core Erlang
   - Matlang `if` → Core Erlang `case`
   - Matlang `for` loop → Core Erlang `letrec`
   - Matlang `while` loop → Core Erlang `letrec`

A Matlang code sample

```
%% Create a 3x3 identity matrix
I = eye(3);
%% The following expression is equivalent to:
%%   V = blas:transpose(blas:vector(s, 3, [1.0, 2.0, 3.0]))
V = [1.0, 2.0, 3.0]';

%% This equals to:  V2 = blas:mul(blas:mul(2.0, I), V)
V2 = 2 * I * V;
%% Result:  V2 = [2.00000, 4.00000, 6.00000]'

%% Function definition
function y = fn(x, t, data)
  y = -x * 3;
end;

%% Function integration (4th-order Runge-Kutta)
Y = rk4(fn,                  % Function to integrate
        3.0,                 % Initial value
        [0.0, 0.1, 0.2],     % Integration points
        []);                 % Data (unused here)
%% Integration result (on final point):  Y = 1.64652
```

FLOW, an Erlang framework for HPTC

Going back to Erlang, we choose a model for abstracting a **generic HPTC application**:

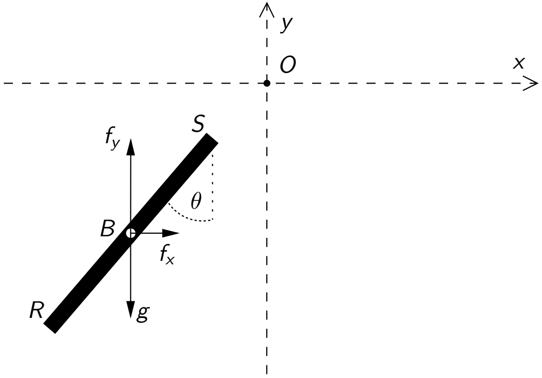
> *A distributed numerical application is a set of looping numerical* **processes** *connected by predefined communication channels (called* **buses***)*

We implemented such model in a framework, called **FLOW**

The developer only needs to define the **bus topology** and the **functions** being looped by each process, while FLOW takes care of:

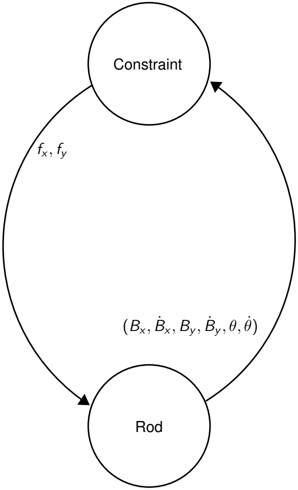
- **distributing** the processes over a cluster of computers
- **dispatching** communications among processes
- **monitoring** the system behaviour

Example: **compound pendulum simulation**



- the rod is free-falling under gravity g. Its position is indicated with barycenter B (coordinates (B_x, B_y)) and angle θ. Horizontal, vertical and angular velocities are indicated with $\dot{B}_x$, $\dot{B}_y$ and $\dot{\theta}$
- the pendulum constraint is simulated by two artificial forces (f_x and f_y) applied to B. They are periodically recomputed so that the rod is moved to make point S coincide with point O. f_x and f_y are thus the pendulum constraint reactions

The FLOWChildSpecs for the pendulum simulation




```
[%% FLOW process specs
 [{type, process},
  {id, 'Constraint'},
  {core, fun core_Constraint/3},
  {node, n1},
  {params, {1.0, 36.0, ...}}, % Coefficients
  {state0, 0},
  {input_ports, [{'State', {vector}}]},
  {output_ports, [{'Forces', {float,float}}]}],
 [{type, process},
  {id, 'Rod'},
  {core, fun core_Rod/3},
  {node, n2},
  {params, {9.8, 1.0, ...}}, % Gravity ...
  {state0, {blas:vector(...)}},
  {output0, [{'State', blas:vector(...)}]},
  {input_ports, [{'Forces', {float, float}}]},
  {output_ports, [{'State', {vector}}]}],
 %% FLOW bus specs
 [{type, bus},
  {id, 'B_state'},
  {node, n2},
  {input_process, {'Rod', 'State'}},
  {output_processes, [{'Constraint','State'}]}],
 [{type, bus},
  {id, 'B_fxfy'},
  {node, n1},
  {input_process, {'Constraint', 'Forces'}},
  {output_processes, [{'Rod', 'Forces'}]}]].
```

ClusterL, an IDE for HPTC applications

We now have all the building blocks for HPTC applications:

the Erlang FFI allows to easily interface existing C code

the BLAS binding adds number-crunching capabilities to Erlang

the Matlang language helps writing concise numerical code

the FLOW framework implements a model for HPTC apps

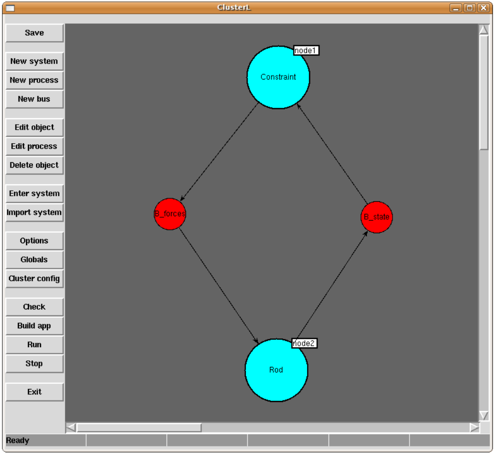
There are, however, two more problems:

- writing `FLOWChildSpecs` may be **tedious and error-prone**
- and **we cannot expect our target users to handle them**

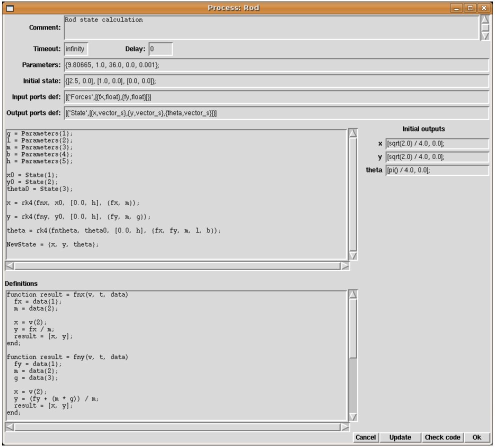
That's why we developed

ClusterL, an IDE for HPTC applications

ClusterL workspace, with pendulum simulation



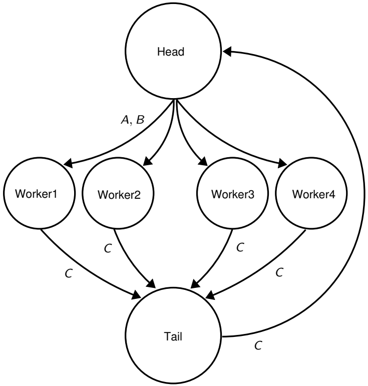
ClusterL process editing window



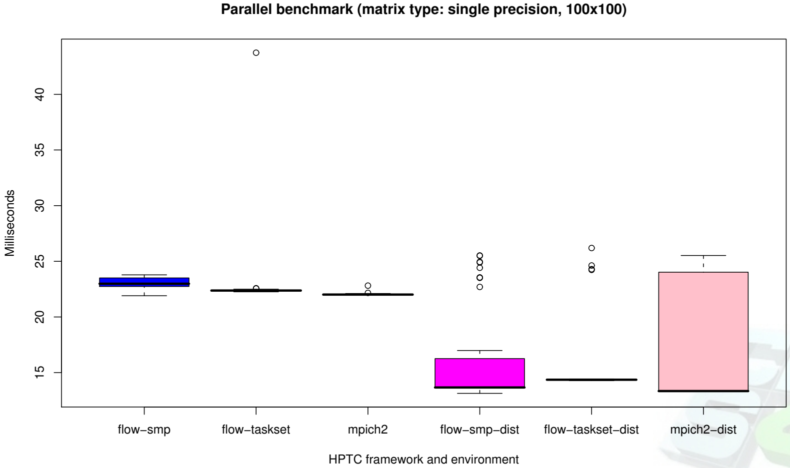
A parallel benchmark

What are the overall performances of our Erlang HPTC suite?

To have an idea, we built the following parallel system using both **ClusterL (on OTP R11B-5)** and **C + MPICH2 1.0.6p1**



Parallel benchmark results



Ubuntu™ 8.04, dual-core AMD™ Athlon™ 64 4200+, 2 GB RAM, ATLAS 3.6.0, MPICH2 1.0.6p1

Conclusions

We have seen how **Erlang could handle HPTC applications**:

the Erlang FFI allows to easily interface existing C code

the BLAS binding adds number-crunching capabilities to Erlang

the Matlang language helps writing concise numerical code

the FLOW framework implements a model for HPTC apps

the ClusterL IDE assists the development of FLOW-based apps

Furthermore, we can say that

- the **benchmarks** show that our Erlang HPTC solution has **good performance**
- we, as **framework developers**, are **much more productive** in Erlang than in C!

Future developments

A lot of work is left to be done:

- ~~FLOW should support **nested processes**, like Simulink™ *systems*~~ — DONE!
- FLOW run-time control functions should be made available through an **user-friendly GUI**
- the ClusterL GUI should be implemented with a modern graphical toolkit, like **wxErlang**
- the **Matlang compiler should be improved**, with more optimizations, **more static typing support** and **less run-time checks**

References

Alceste Scalas
Erlang enhancement proposal 7: Foreign function interface, September 2007
http://erlang.org/eeps/eep-0007.html

Alceste Scalas
Home page of the foreign function interface (FFI) for Erlang/OTP, 2008
http://muvara.org/crs4/erlang/ffi/

Thank you!