Parallel programming with Sklml

Quentin Carbonneaux François Clément Pierre Weis

INRIA

MaGiX@LiX - September 22nd, 2011

Industry standards

OpenMP

- It is used to parallelize purely sequential code;
- it is designed for shared memory architectures;
- it is low level and intrusive.

MPI

- It is a kind of assembly toolbox for parallelism;
- let you fine tune the parallelism for the application;
- the code is a mixture of sequential instructions and parallel primitives;
- the parallelization process is difficult and lengthy.

- Both approaches give very efficient parallel programs.

Design goals for Sklml

The traditional approaches to parallelism exhibit major drawbacks

- too low level notations and concepts;
- hence, extremely error prone;
- hence, very demanding in programming/debugging effort.

The Sklml answers

- separation: the parallelization code does not interfere with the core of the computational code;
- high-level: skeleton programming is an abstract description of parallelism;
- reliable: functional and statically type checked;
- well-founded: the sequential and parallel versions of a program always give the same results (adequacy theorem).

What Sklml is

As a result, Sklml

- is high level: based on a compositional combinator algebra;
- clearly isolates the description of the parallelism in the skeletons of the algebra;
- is a powerful tool to describe parallelism (parallelization code is typically a few tens of lines);
- is type safe by construction due to the skeleton algebra;
- is a true Domain Specific Language embeded in OCaml;
- frees the programmer from all the ugly low level details (message passing, process management);
- is not restricted to shared memory systems (works on clusters);
- is a complete toolkit (compiler + library + runtime system).

What Sklml is not

On the other hand,

- Sklml does not give access to processes, shared memory, . . . ;
- hence, Sklml does not permit to encode every parallel scheme;
- hence, Sklml may not be the fastest parallel toolkit.

Sklml skeletons

What is a skeleton

A skeleton is an OCaml value with type `('a, 'b) skel` (its input is of type `'a` and its output is of type `'b`).
A skeleton is a function acting on streams (a potentially infinite sequence of data).

The Sklml library provides skeletal combinators which might either

- encode some kind of parallelism (data parallelism, program parallelism);
- encode some kind of control structure (`if-then-else`, `do-while`,...).

Sklml skeletons

The farm skeleton combinator

The farm skeleton combinator applies one treatment in parallel to a flow of data.

```
val farm : ('a, 'b) skel * int → ('a, 'b) skel;;
```

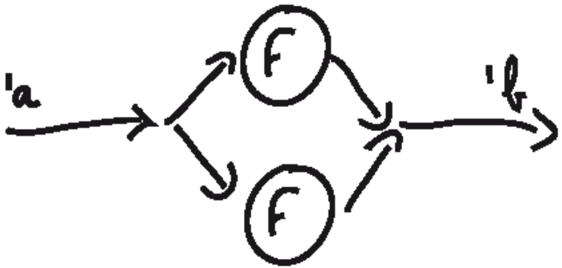

Figure: farm $(F, 2)$ skeleton graph

Sklml skeletons

The pipeline skeleton combinator

The pipeline skeleton combinator modelizes the parallel composition of functions.

```
val ( ||| ) :
  ('a, 'b) skel → ('b, 'c) skel → ('a, 'c) skel;;
```

Figure: *G* ||| *F* skeleton graph

Sklml skeletons
The loop skeleton combinator

The loop skeleton combinator is a control combinator: it iteratively applies a skeleton on a data until the resulting value negates a given predicate.

```
val loop :
  ('a, bool) skel * ('a, 'a) skel → ('a, 'a) skel;;
```

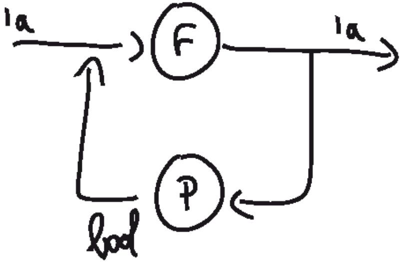

Figure: `loop` (P, F) skeleton graph

Sklml skeletons
Other skeleton combinators

The `&&&` skeleton combinator modelizes the parallel application of two functions.

```
val ( &&& ) :
  ('a, 'b) skel → ('c, 'd) skel →
  ('a * 'c, 'b * 'd) skel;;
```

The `+++` skeleton combinator modelizes the parallel application of two functions on the elements of the direct sum of two sets.

```
val ( +++ ) :
  ('a, 'c) skel → ('b, 'c) skel →
  (('a, 'b) sum, 'c) skel;;
```

where `sum` is the classical direct sum of sets defined as

```
type ('a, 'b) sum = Inl of 'a | Inr of 'b;;
```

Sklml skeletons

Other skeleton combinators

The `farm_vector` skeleton combinator modelizes the parallel application of a function to the items of a vector.

```
val farm_vector :
  ('a, 'b) skel * int → ('a array, 'b array) skel;;
```

The `rails` skeleton combinator modelizes the parallel application of a vector of *n* functions to the *n* items of an input vector.

```
val rails :
  (('a, 'b) skel) array → ('a array, 'b array) skel;;
```

A simple example
Introducing the example

Problem

Find the first element which does not satisify a given property *P*.
We suppose that *P* is expensive and must be computed in parallel.
We also have two functions:

- `next_elm` which gives the "successor" of its input;
- `test_elm` a predicate function which test if an element satisfies the property *P*.

This problem is borrowed from the program `PrimeGen` that generates primes satisfying strong cryptographic properties.

A simple example
The actual Sklml code

In sequential C, this actually boils down to a simple while loop:

```
do {
    elm = next_elm(elm);
} while (test_elm(elm) == True);
```

A simple example

The actual Sklml code

In sequential C, this actually boils down to a simple while loop:

```
do {
    elm = next_elm(elm);
} while (test_elm(elm) == True);
```

In Sklml, the program uses the `loop` skeleton, with a predicate described as a parallel pipeline:

```
let find_skl nw =
  loop ( farm_vector (test_elm, nw) ||| fold_or,
         next_elms ) in
  ...
```

The Sklml compiler can compile this program for both sequential and parallel executions.

Domain Decomposition problems using Sklml (1)

Sklml was developed to cope with scientific computing problems and in particular domain decomposition problems.

Domain decomposition algorithm

A computation needs to be performed on a grid (*domain*) splitted in different small *subdomains*.

Domain decomposition algorithms perform a sequence of rounds built of two steps:

1. each processor run a step of a numerical scheme on its subdomain;
2. border information is exchanged between processors.

Domain Decomposition problems using Sklml (2)

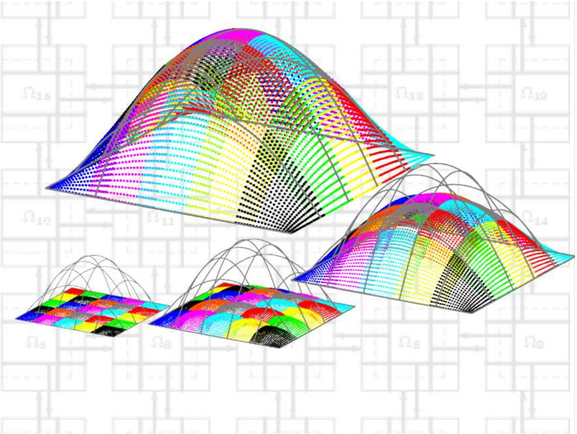

Figure: Computation using a domain decomposition algorithm

Domain Decomposition problems using Sklml (3)

Sklml provides a library of derived operators written in terms of composition of the basic skeletons.

The `make_domain` skeleton is specific to decomposition domain algorithms.

Given a vector of skeleton workers, the connectivity of the subdomains, and a stopping criterion, the `make_domain` skeleton combinator creates a skeleton implementing the appropriate domain decomposition algorithm.

```
type ('a, 'b) worker_spec =
  ('a border list, 'a * 'b) skel * int list

val make_domain :
  (('a, 'b) worker_spec) array ->
  ('b array, bool) skel ->
  ('a array, ('a * 'b) array) skel
```

The Sklml distribution

Sklml is a set of 4 components written both in OCaml and Sklml:

- a compiler (`sklmlc`);
- a core library of basic skeletons;
- an extra library of derived skeletons;
- a parallel process manager (`sklmlrun`).

Sklml is free software available at `http://sklml.inria.fr/`.

Sklml's key feature (1)

Fact

Skeletal combinators have simple sequential semantics.

As a consequence, two compilation modes are proposed, a sequential interpretation of skeletal combinators and a parallel one.

Sklml's key feature (1)

Fact

Skeletal combinators have simple sequential semantics.

As a consequence, two compilation modes are proposed, a sequential interpretation of skeletal combinators and a parallel one.

The two semantics in practice

Compile either in parallel mode:

```
sklmlc -mode par code.ml
```

Or in sequential mode:

```
sklmlc -mode seq code.ml
```

Sklml's key feature (2)

The Sklml system guaranties that:

- the parallel and sequential programs give the same results;
- hence, if the code runs properly in sequential mode, it is guaranteed to be correct in parallel mode.

Hence, the methodoly:

1. develop and debug using the sequential semantics;
2. start the heavy parallel computation after changing a flag in the makefile!

Sklml and OCaml 3.12

Due to its high abstraction level, Sklml needs advanced features of the OCaml language:

- first class modules to emulate GADTs (3.12);
- lazy evaluation to represent possibly infinite computations;
- second rank polymorphism to provide a polymorphic API;
- polymorphic recursion to uniformly implement the skeletons (3.12).

Sklml and the other languages

Sequential parts of Sklml programs can be written:

- in pure OCaml;
- in C, with the standard OCaml Foreign Language Interface;
- in many languages, with the external data communication layer associated to Sklml (Pio, the Polyglot I/O library).

Already written code can be parallelized with Sklml!
(In particular, closed or complex codes from third party).

State of the art

SkIml is robust and usable but can be improved:

- improve the load balancing system;
- handle and recover from network or machine failures;
- improve error messages;
- enrich the library of derived skeletons;
- evangelism: tell people they must use it!

That's all folks!

- Any questions?
- Want to see some code?

Implementing simple helper skeletons

```
let projl = skl () -> fun (x, _) -> x;;
let projr = skl () -> fun (_, x) -> x;;

let injl = skl () -> fun x -> Inl x;;
let injr = skl () -> fun x -> Inr x;;
```

Implementing a `if_then_else` skeleton

```
let dup = skl () -> fun x -> (x, x);;
let to_sum = skl () ->
  fun (x, b) -> if b then Inl x else Inr x
;;

let if_then_else (cond_skl, then_skl, else_skl) =
  dup () ||| (id () *** cond_skl) |||
  to_sum () ||| (then_skl +++ else_skl)
;;
```

Factorial in pure Sklml

```
let is_gt = skl i -> ( < ) i;;
let con = skl x -> fun _ -> x;;
let minus = skl i -> fun x -> x - i;;
let mult = skl () -> fun (a, b) -> a * b;;

let fact =
  dup () ||| (id () *** con 1) |||
  loop
    (projl () ||| is_gt 1,
     dup () ||| ((projl () ||| minus 1) *** mult ()))
  projr ()
;;
```