Sklml: Functional Parallel Programming

User Manual

Version 1.0

Quentin Carbonneaux François Clément Pierre Weis[1]

January 21, 2014

[1]INRIA Rocquencourt - France

Contents

Abstract

Writing parallel programs is not easy, and debugging them is usually a nightmare. Over the last years, several researchers coped with these difficulties by developing a structured approach to parallel programming via template based compiler techniques. The *skeleton programming* approach uses a set of predefined patterns for parallel computations. The skeletons are higher order functional templates that describe the program underlying parallelism. So, marrying a full-fledged functional language and a carefully crafted skeleton algebra seems to be the way to go to obtain a powerful parallel programming environment.

This document describes the Sklml (*['skəlɛmɛl]*) system that embeds an innovative compositional skeleton algebra into the OCaml language.

Sklml provides an optimizing compiler and a runtime computing network manager. Thanks to its skeleton algebra, Sklml provides two evaluation regimes to programs: a regular sequential evaluation (merely used for prototyping and debugging) and a parallel evaluation obtained via a recompilation of the source program in parallel mode. Sklml is also designed to be a parallel computation driver running worker programs written in heterogeneous external languages (e.g. C, C++, Fortran).

Chapter 1

Skeleton based programming and Sklml

In a skeleton based parallel programming model [2, 6, 4] a set of *skeletons*, i.e. second order functionals modeling common parallelism patterns are provided to the programmer. The programmer uses skeletons to describe the parallel structure of an application and uses a plain sequential programming language to express the sequential portions of the parallel application. There is no way to express parallel activities but skeletons: there are no notions of explicit process creation, scheduling, termination, nor any communication primitives, shared memory concepts, nor any mean to detect execution on parallel architecture.

Sklml is a programming environment that allows to write parallel programs in OCaml[1] according to a new skeleton model derived from CamlP3l[3][2]. It provides a seamless integration of parallel programming in functional programming and advanced features like sequential logical debugging (i.e. functional debugging of a parallel program via execution of the described parallel architecture on a single sequential machine) and strong static typing, useful both in teaching parallel programming and in building full-scale applications.

In this chapter, we will first discuss the goals of our system design, then recall the basic notions of the skeleton model for structured parallel programming and describe the skeleton model provided by Sklml, providing informal sequential and parallel semantics.

1.1 The system design goals

Sklml is a complete rewrite of CamlP3l to improve the usability and code quality of the system. However, original goals of CamlP3l are still leading directions for Sklml. CamlP3l put the ideas of algorithmic skeletons in the OCaml functional language in order to benefit from the strong typing discipline and all the programming facilities offered by OCaml.

We also keep a salient feature of CamlP3l: the dual interpretation of skeletons as a sequential and parallel specification of programs. This is a real advantage since it lets the programmer debug, trace, place breakpoints in its code in the traditional way. When the code is logically correct (i.e. it executes correctly in sequential mode), the programmer is guaranteed to obtain a correct parallel execution. This is definitely not the case of programs written using a sequential language and directly calling communication library primitives such as the Unix socket interface or the MPI or PVM libraries. In effect, these library introduce

[1] See `http://caml.inria.fr/`.

[2] CamlP3l was a major rework of the initial OCamlP3l[5] based on the p3l[9] Pisa Parallel Programming Language.

their own set of problems and pitfalls, while imposing the inextricable interleaving of the program logic with low level management of data exchange and handling of process creation and monitoring.

Furthermore, the compilation and execution of Sklml programs is extremely simplified: we provide the `sklmlc` compiler as an alternative to the regular OCaml compiler; programs compiled in sequential mode are regular system executables, while their parallel version, compiled in parallel mode, must be launched with the `sklmlrun` runtime job manager to specify the available computation resources (physical machines, relative computing power), as described in section 2.2.2.

Finally, in order to keep a strong conviction of the equivalence between parallel and sequential execution of Sklml programs, the whole code of the Sklml framework has been rewritten from scratch, so that the sequential and parallel runtime share a large common infrastructure, up to the point that the parallel runtime uses the stream library of the sequential runtime.

1.2 The skeleton model of Sklml

A skeleton parallel programming model supports so-called 'structured parallel programming' [2, 6, 4]. Using such a model, the parallel structure and behavior of any application has to be expressed by using *skeletons* picked up out of a collection of predefined ones, possibly in a nested way. Each skeleton models a typical *pattern* of parallel computation (or *form* of parallelism) and it is parametric in the computation performed in parallel. As an example, pipeline and farm have been often included in skeleton collections. A *pipeline* models the execution of a number of computations (stages) in cascade over a stream of input data items. Therefore, the pipeline skeleton models all those computations where a function $f_n(f_{n-1}(\ldots(f_2(f_1(x)))\ldots))$ has to be computed (the f_i being the functions computed in cascade). A *farm* models the execution of a given function in parallel over a stream of input data items. Therefore, farm models all those computations where a function $f(x)$ has to be computed independently over n input data items in parallel.

In a skeleton model, a programmer must select the proper skeletons to program his application leaving all the implementation and optimization to the compiler. This means, for instance, that the programmer has no responsibility in deriving code for creating parallel processes, mapping and scheduling processes on target hardware, establishing communication frameworks (channels, shared memory locations, etc) or performing actual inter-process communications. All these activities, needed in order to implement the skeleton application code onto the target hardware are completely in charge of the compile and runtime support of the skeleton programming environment. In some cases, the support also computes some parameters such as the parallelism degree or the communication grain needed to optimize the execution of the skeleton program onto the target hardware [7, 1, 8].

Current Sklml version supplies three kinds of skeletons:

- *Task parallel* skeletons model parallelism exploited between *independent* processing activities relative to different input data. In this set, we have: `pipe` (`|||` in infix form) (cf. 1.3.3) and `farm` (cf. 1.3.2).
- *Data parallel* skeletons model parallelism exploited computing different parts of the same input data. In this set, we provide `farm_vector` (cf. 1.3.7) and `rails` (cf. 1.3.8). The `farm_vector` skeleton models the parallel application of a generic function f to all the items of a vector data structure. So does the `rails` skeletons with fixed size vectors. The skeletons product (`***` in infix form) and sum (`+++` in infix form) are also data parallel, they act on data built of smaller elements and find their roots in basic functional programming (cf. section 1.3.4 and 1.3.5).

- *Service* skeletons or *control* skeletons, which are not parallel *per se*. Service skeletons encapsulate OCaml non-parallel code within other skeletons (`loop` skeleton, cf. 1.3.6) or transform a sequential code into a valid Sklml skeleton (the `skl` syntactic construct, cf. 1.3.1).

1.2.1 Parallel execution model

Structure of Sklml programs

A Sklml program has three sections:

1. a set of plain sequential function definitions written in OCaml;

2. a set of skeleton definitions;

3. a `pardo` invocation.

Set (1) has nothing mysterious: it is just regular OCaml programming. Set (2) is specific to Sklml, the skeletons defined here are values of type (α, β) `skel`. Although (α, β) `skel` is intended to be the type of skeletons from α values to β values, this type is still fully abstract, hence no skeleton of set (2) can be applied to anything. In Sklml, the only way to apply (or use, or run) an (α, β) `skel` value is to turn it into a true function via the provided primitive `parfun`:

```
val parfun : ('a, 'b) skel -> 'a stream -> 'b stream;;
```

This transformation must occur into the `pardo` invocation (the third section of a Sklml program). The `pardo` function is the Sklml primitive that starts the computation of a Sklml program. To launch the execution, the `pardo` primitive should be applied to the main function of the parallel program. Then `pardo` applies the main function to the `parfun` primitive, hence allowing main to turn skeletons of set (2) into regular OCaml functions from streams to streams. Last but not least, the main procedure must define an initial stream of values and apply the freshly obtained functions to this initial stream.

Interpretation of Sklml programs

The processes that execute a parallel program are launched on a computation grid by a dedicated program available on every node. This program starts instances of the compiled code for the current architecture and tells each instance which sequential function it has to run, and connects it to the other nodes. When all processes are up and running, the `parfun` primitive sends the input stream in the computational network and returns the stream of results.

Notice that the execution model assumes an unlimited number of homogeneous processors. In practice, there are much more processes than processors, and processors have heterogeneous capacity. The Sklml library handles this situation in a transparent way. However, the programmer may help the library by providing hints on the computing power of processors and hints on complexity of sequential functions (see the color annotations in section 1.4).

1.2.2 A simple example: farming computation

Let us examine a complete Sklml program. The source of program `S` is in figure 1.1. The program `S` computes the square of each element of an input stream.

The structure of program `S` is as follows:

```
1    open Skl;;

2    let square x = x * x;;

3    let print_result x = print_int x; print_newline ();;

4    let main { Parfun.parfun = parfun; } =
5      let square_worker = skl () -> square in
6      let farm_worker = farm (farm_worker (), 4) in
7      let compute = parfun farm_worker in
8      let s = Sklstream.of_list [1;2;3;4;5;6;7;8] in
9      let result = compute s in
10     Sklstream.iter print_result result
11   ;;

12   Parfun.pardo main;;
```

Figure 1.1: Sklml code using a farm to square a stream of integers.

1. First section contains two simple OCaml functions: `square` and `print_result` (lines 2 and 3).
2. The `main` function gets a `parfun` argument. In lines 5 and 6 `main` defines two skeletons: the `square_worker` base skeleton and the `farm_worker` skeleton that uses four `square_worker` to compute the results in parallel.
3. The `farm_worker` skeleton is turned into the `compute` function using the `parfun` primitive (line 7), because of its type, the `parfun` function can be applied several times inside the `main` function to transform skeletons.
4. The `compute` is applied to the input stream `s` giving a result stream named `result`.
5. The results are displayed using the Sklml function `Sklstream.iter` to iterate the printing functions over every element of the output stream.
6. The last top-level statement in line 12 will run the main function and it will pass to `main` the appropriate `parfun` argument. This `parfun` function, depending on the compilation mode is either a parallel process launcher or a sequential implementation of the skeleton semantics.

1.3 Skeleton syntax, semantics, and types

Each skeleton, once started by a `parfun` application, is a stream processor, transforming an input stream into an output stream and is equipped with two semantics.

Sequential semantics a suitable sequential OCaml function transforming all the elements of the input stream.

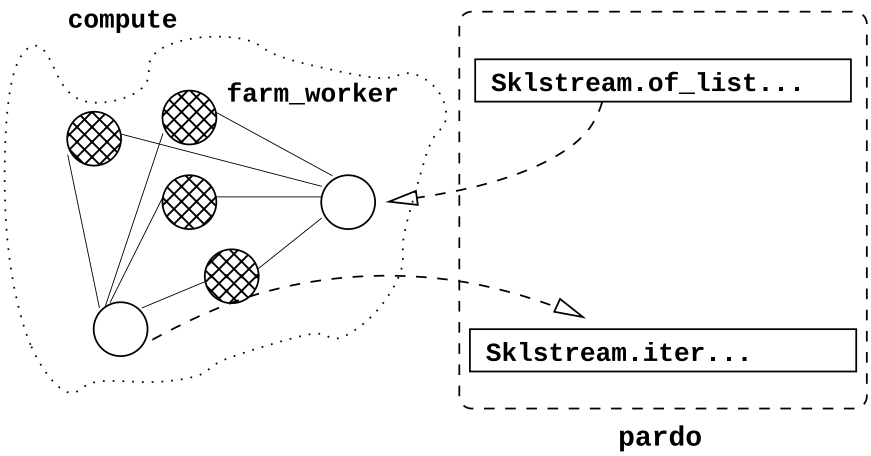

Figure 1.2: Overall process network of the simple farm squaring a stream of floats.

Parallel semantics a process network implementing the stream transformation in parallel.

From an OCaml point of view, a skeleton taking elements of type α and returning elements of type β has the type `(α, β) skel`.

1.3.1 The `skl` construction

```
let (succ_skl : unit -> (int, int) skel) =
  skl () -> succ in ...
let (add_skl : int -> (int, int) skel) =
  skl i -> fun x -> x + i in ...
```

Figure 1.3: Sklml code using the `skl` construct.

```
let (my_skeletons : (int, int) skel list) =
  List.map (skl i -> (+) i) [ 1; 2; 3; 4; 5; 6; 7; 8; ] in ...
```

Figure 1.4: Sklml code creating initialized skeletons.

The `skl` syntactic construction lets the programmer lift a regular OCaml function f to the universe of skeletons. This construct is the Sklml correspondent of the `fun` construct of the OCaml language. It provides a way to define a skeleton generator that associates a skeleton to the value of an *initialization parameter*, that is a value of any non-functional type that is marshalled and sent to the remote worker at initialization time. An example of such an initialization is given in figure 1.4.

Indeed, the initialization parameter is mandatory for at least two reasons:

- it solves the “initialization problem” in a nice way;
- it is needed in order to make one of the programming constraints described in the next paragraph lighter.

Golden rules. Using the `skl` construct, two rules must be respected:

1. No free variables shall appear in the body of a `skl` construct, except global variables.

2. The type of the initialization parameter **must** be a marshallable type. In particular, it can not contain functional values.

If you you do not respect these rules, in the best case, your code will not compile, in the worse case it will behave in the strangest way. If the constraint (2) is violated, a runtime error will occur at initialization time.

Note: these rules are in sharp contrast with the OCamlP3l system for which the programmer had to move all its parfun applications to the top-level to let the library know what skeletons he will be using in its code.

```
let main { parfun = parfun; } =
  let (my_skeletons : (int, int) skel) =
    List.map (fun x -> (skl () -> (+) x) ())
      [ 1; 2; 3; 4; 5; 6; 7; 8; ] in
  ...
```

Figure 1.5: Wrong Sklml code.

The code in figure 1.5 violates constraint (1); the right way to implement this behavior is to properly use the initialization parameter, as shown in figure 1.4.

A good practice to write a Sklml `skl` skeleton is to write it as a standard OCaml function, then abstract all the free variables violating rule (1) as extra arguments in the initialization parameter.

```
let main { parfun = parfun; } =
  let skls =
    List.map (skl s -> Printf.eprintf "%s\n" s; succ)
      [ "hello"; "world"; ] in
  let print_int_list =
    List.iter (fun i -> print_int i; print_newline ()) in
  let run_skl l sk =
    Sklstream.to_list (parfun sk (Sklstream.of_list l)) in
  let ss' = List.map (run_skl [ 1; 2; 3; ]) skls in
  List.iter print_int_list ss'
;;
pardo main;;
```

Figure 1.6: `skl` initializations with side effects.

Finally, note that *the body of the `skl` construct is running on the **remote** slave.* Hence, slave specific initializations must be done after the arrow of the `skl` construct. For example, in the example of figure 1.6, the two strings are printed on the slave's displays; so if the slaves are remote, you might never see the output!

1.3.2 The farm skeleton

The farm skeleton computes in parallel a function f over the data items in its input stream. From a functional point of view, given a stream of data items $x_1, \ldots, x_n$, and a function f, the expression farm(f, k) computes $f(x_1), \ldots, f(x_n)$. Parallelism is gained by having k independent processes that compute f on k different items of the input stream. The farm skeleton has type:

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

The farm function takes a pair of parameters as argument:

- the first one denotes the skeleton expression of the farm worker;
- the second one specifies the degree of parallelism of the farm, i.e. the number of worker processes that have to be set up in the farm.

1.3.3 The ||| skeleton

The pipe, or pipeline, skeleton is denoted by the infix operator |||; it performs in parallel the computations relative to different stages of a function composition over data items of the input stream. Functionally, $f_1 ||| f_2 \ldots ||| f_n$ computes $f_n(\ldots f_2(f_1(x_i))\ldots)$ over all the data items x_i in the input stream. Parallelism is now gained by having n independent parallel processes. Each process computes a function f_i over the data items produced by the process computing f_{i-1} and delivers its results to the process computing f_{i+1}.

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

In terms of (parallel) processes, a sequence of data appearing onto the input stream of a pipe is submitted to the first pipeline stage. This stage computes the function f_1 onto every data item appearing onto the input stream. Each output data item computed by the first stage is submitted to the second stage, computing the function f_2 and so on, until the output of the $n-1$ stage is submitted to the last stage. Eventually, the last stage delivers its own output onto the pipeline output channel.

1.3.4 The *** skeleton

The *** skeleton is denoted by the infix operator ***; it performs two computation in parallel by splitting its input, then computing on the two resulting values and merging the results. Functionally, f_1***f_2 computes $(f_1(x_{i,1}), f_2(x_{i,2}))$ on each element $(x_{i,1}, x_{i,2})$ of the input stream. Parallelism is gained by running the computation of f_1 and f_2 in parallel.

```
val ( *** ) :
  ('a, 'b) skel -> ('c, 'd) skel -> ('a * 'c, 'b * 'd) skel;;
```

A typical usage of the *** skeleton is when one wants to compute the function $f : x \to g(x) + h(x)$ by running the computation of g and h in parallel. The Sklml way to implement this parallel scheme is shown in figure 1.7.

```
let split_skl = skl () -> fun x -> (x, x) in
let merge_skl = skl () -> fun (x, y) -> x + y in
let f_skl = (split_skl ()) ||| (g *** h) ||| (merge_skl ()) in
...
```

Figure 1.7: Using `***` to compute $f : x \to g(x) + h(x)$.

1.3.5 The +++ skeleton

The `+++` skeleton is denoted by the infix operator `+++`, it is used when data can be of two kinds, i.e. the type of data is a sum of two other types. When treatments on data depend on its kind, parallelism can be gained by computing the two function applications of successive elements at the same time. Using composition it becomes possible to create values of more than two types. Then the `+++` skeleton can be seen as a way to express a simple pattern matching on incoming data.

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

The type sum is defined as follow:

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

> *Note:* The `+++` skeleton can be used to express a skeleton often provided by skeleton libraries, the `if-then-else` skeleton. This implementation is distributed in the standard Sklml distribution in the extra library, see section 3.2.

1.3.6 The `loop` skeleton

The `loop` skeleton computes a function f over all the elements of its input stream until a boolean condition g is verified.

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

This skeleton was designed to find a fixpoint of a function with respect to some criterion. Figure 1.8 is the OCaml description of the action of the `loop` skeleton on an input value x_i.

```
let rec loop_aux xᵢ =
  if not g(xᵢ)
  then xᵢ
  else loop_aux f(xᵢ) in
loop_aux f(xᵢ)
```

Figure 1.8: `loop` semantics in OCaml.

1.3.7 The farm_vector skeleton

The farm_vector skeleton computes in parallel a function over all the data items of a vector, generating the (new) vector of results. Therefore, for each vector X in the input stream, farm_vector(f, n) computes the function f over all items of $X = [x_1, \ldots, x_n]$, using n distinct parallel processes that compute f over distinct vector items $\big[f(x_1), \ldots, f(x_n)\big]$.

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

In terms of parallel processes, a vector appearing onto the input stream of a farm_vector is split in n elements and each element is processed by one of the n workers. Workers simply apply f to the elements they receive. Then all results are merged in an output vector by a dedicated process.

1.3.8 The rails skeleton

The rails skeleton looks like the farm_vector skeleton, except that it uses an array of skeletons.

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

This skeleton will take as input arrays having the same size as the vector of skeletons given as argument. Because input vectors have the same size as the vector of skeletons, the Sklml runtime can provide the guarantee that the worker f_i will always process the ith element of the input vector. This can be very useful when the treatment of elements of an input vector is not homogeneous.

Note: with the rails skeleton, the treatment might not be homogeneous, if the data itself is also not homogeneous the *** skeleton must be used, see section 1.3.4.

Note: the skeleton farm_vector can not be substituted to rails, even if the number of workers and the number of elements in input vectors are the same, since using farm_vector the elements having the same index in the input vectors will not always be processed by the same worker.

1.4 Coloring skeletons

Because not all tasks have the same computational needs and not all processors have the same computational power, the Sklml system allows to attribute a *color* to each task and each processor. The color annotations inform the Sklml system of the computational strength of the nodes of the network and the computational requirements of the tasks.

Colors on skeletons. In Sklml, the programmer only annotates the color of *atomic* skeletons, (the skeletons created by the skl construct as described in section 1.3.1). The color of other skeletons is automatically computed by summing the colors of the atomic skeletons they enclose.

The colorize function takes a skeleton sk and an integer col and change the color of sk to be col, if and only if the skeleton was created using the skl construct.

```
val colorize : ('a, 'b) skel -> int -> ('a, 'b) skel;;
```

Colors on nodes. Only assigning *colors* to tasks does not make sense, one has to tell the Sklml system how much resources are available on each processor. This information is given at launch time, see details in chapter 2.

Chapter 2

Compiling and running a Sklml application

The Sklml distribution provides convenient tools to compile and run programs. These tools can be very effective, however an overview of the internals is still required to get out of tricky situations. This chapter gives an overview of the power tools bundled within Sklml and an overview of what is going on behind the hood.

Throughout this chapter we use the simple example given in figure 2.1. We stress the point that you are highly encouraged to type this program yourself in your favorite text editor because this example, while quite short, is in fact very subtle. As an exercise try to guess all possible outputs (if any) of this code.

In the following sections we assume the example of figure 2.1 to have been written in the file `hello.ml`.

2.1 Compiling

2.1.1 Effective compilation using `sklmlc`

Compiling a Sklml program is achieved by the `sklmlc` compiler. In the current implementation `sklmlc` is a simple wrapper over the OCaml compiler. The OCaml compiler can not compile Sklml source code directly, because of the special `skl` syntax. This peculiar Sklml syntax is handled in a pre-processing phase triggered by `sklmlc`.

The `sklmlc` compiler can deal with all the options of the OCaml compiler. A new `-mode` option tells `sklmlc` whether a program needs to be linked using the parallel or the sequential runtime. Thus, to compile the example file `hello.ml` in sequential mode, two commands are needed :

```
sklmlc -mode seq -o hello.cmo -c hello.ml
sklmlc -mode seq -o hello.byt hello.cmo
```

If you want the parallel version, just type:

```
sklmlc -mode par -o hello.cmo -c hello.ml
sklmlc -mode par -o hello.byt hello.cmo
```

As expected the two commands can be reduced to one:

```
open Skl;;
open Parfun;;

let stream_of_string s =
  let rec explode i =
    if i = String.length s then [] else
    s.[i] :: explode (succ i)
  in
  Sklstream.of_list (explode 0)
;;

let id x = x;;

let put_char c =
  Printf.printf "%c%!" c; c
;;

let main { Parfun.parfun = pf; } =
  let skl_put_char = skl b -> if b then put_char else id in
  let skl_farm b nw = farm (skl_put_char b, nw) in
  let stream = stream_of_string "Hello world!\n" in
  Sklstream.iter ignore
    (pf (skl_farm true 10) stream);
  Sklstream.iter (fun x -> ignore (put_char x))
    (pf (skl_farm false 10) stream)
;;

pardo main;;
```

Figure 2.1: One simple "Hello world" application for Sklml.

```
sklmlc -mode seq -o hello.byt hello.ml # for the sequential
sklmlc -mode par -o hello.byt hello.ml # for the parallel
```

The reader familiar with the OCaml system might ask where is the native Sklml compiler. In fact, the `sklmlc` compiler automatically selects the relevant OCaml compiler: if called with some filenames ending with `cmx*` it uses the OCaml native compiler, otherwise it uses the bytecode compiler.

The Sklml compiler can compile any valid OCaml code (except for bad usage of the new `skl` reserved keyword).

2.1.2 Handling dependencies with `sklmldep`

When bigger projects are created in OCaml it is usual to automate the compilation process because of the number of source files. This is usually done using `make`. However, because of relations between modules, compilation needs to be done in a specific order. This order is provided to `make` by the `ocamldep` tool. Because of the `skl` syntactic construct, `ocamldep` will fail during the analysis of Sklml code. Thus, a specific tool wrapping `ocamldep` is provided within the Sklml system : `sklmldep`. This dependency generator behaves exactly as `ocamldep`, and also parses standard OCaml code as long as this one does not use erroneously the `skl` keyword.

2.1.3 Behind the scene

The two tools presented above have one common option which might be informative in some critical situations and for the self education of the reader, the `-debug` option. This option will make the Sklml tools output the commands they will trigger.

2.1.4 About `hello.ml`

The example file compiled in this section was created to show the subtle design invariants of the Sklml system, more precisely, it shows the contrast between data order and computations order. It is distributed as an example in the current Sklml version. It is in the Sklml archive in the directory `example/Hello`.

2.2 Running

2.2.1 Sequential execution

When compiled against the sequential execution runtime, a Sklml application is a classical executable program, and you run it as any regular OCaml program.

```
./hello.byt
```

2.2.2 Parallel execution

Parallel execution model reviewed

The parallel execution model is briefly described in section 1.2.1. The `pardo` call contacts a daemon program provided in the Sklml distribution named `sklmlpard`. This program can run in two modes, a master mode and a slave mode. The combination of one master and several slaves provides a way to use computational resources to the Sklml runtime. Indeed, when a Sklml application needs to launch one task remotely it contacts the master `sklmlpard` and gives him information necessary to start the task (i.e. an identifier representing the task itself, its color and its initialization parameter). Then, the master `sklmlpard`, which keeps track of the load of each of its slaves, contacts one of them and gives it the information just received. The slave finally spawns the task on the remote machine.

This launching mechanism implies that at least one `sklmlpard` program must be running on each machine used by the computation. If the computation ends correctly, all `sklmlpard` programs are killed.

Launching parallel computations with `sklmlrun`

The `sklmlrun` program has been created because the task of launching the `sklmlpard` program on all the machines used by the computation is tedious.

This convenient tool is configured using a file storing human readable information. Many examples of such a file are provided in the standard Sklml distribution, they are all named `sklmlrun.conf`. Figure 2.2 is an example of such a file.

```
(* Master node *) {
  name = "Master";
  host = "127.0.0.1";
  shell = "sh -c '%c'";
  daemon_path = "sklmlpard";
}

(* Local slave *) {
  name = "Local slave";
  shell = "sh -c '%c 2>local-log~'";
  color = 2;
  host = "127.0.0.1";
  program_path = "./hello";
  daemon_path = "sklmlpard";
}
```

Figure 2.2: Sample `sklmlrun.conf` file.

Once the file `sklmlrun.conf` corresponding to your application has been created, `sklmlrun` will do the job of starting `sklmlpard` and launching your application with the appropriate options:

```
sklmlrun -conf ./sklmlrun.conf ./hello
```

Use the `-conf` option to specify the file in which `sklmlrun` must read its configuration then give the name of the Sklml application. If your application needs to be launched with options you will have to give the command line between double quotes:

```
sklmlrun -conf ./sklmlrun.conf "./hello -some -options 42"
```

> *Note:* a good way to write a `sklmlrun.conf` file is to take an existing one, for example, the one given in figure 2.2, and to change options relative to your application (mainly the `skl_path` one).

Basic description of `sklmlrun`'s configuration file. The configuration file needs at least two sections, *sections* are enclosed within braces. The first section is always about the master `sklmlpard` node, it must have the following fields filled : `name`, `host`, `shell`, `sklmlpard`. One field is affected using a construct of the type

$$\langle field\rangle = \langle value\rangle\langle optional\ semicolon\rangle$$

It is good style to always put a semicolon in an assignment. A $\langle value\rangle$ can be a string (enclosed in double quotes) or an integer (in its decimal representation). Strings assigned to the field `shell` are subject to expansion, a `%c` will be replaced by the command that has to be launched by the shell, a `%h` will be replaced by the `host` field of the current section. This expansion mechanism can be used to have `ssh` or `rsh` as a shell command, thus allowing to reach non local machines.

A more accurate description of the configuration file format can be found in the manual page related to `sklmlrun`. This manual page comes with the Sklml distribution and is installed in an appropriate directory by the installation process.

The `sklmlpard` application

Most of the time the `sklmlpard` program will be launched by the `sklmlrun` helper. However, the knowledge of `sklmlpard` might enlighten when strange behaviors are exhibited.

The `sklmlpard` application, as explained before, can run in two modes, a slave mode and a master mode. A slave `sklmlpard` is always connected to a master `sklmlpard`. Thus, to start a computation network, the master must be launched first. To launch a master `sklmlpard`, use the `-master` option. This option takes an integer as parameter which denotes the number of slaves that will be handled by it.

```
# on master terminal #
$ sklmlpard -master 1
sklmlpard: listenning for slaves on 0.0.0.0:52714
```

When launched this way the `sklmlpard` program displays on its standard output the address it listens on. This address must be used to connect slaves. Slaves are started using the `-slave` option. This option takes the IPv4 address of the master `sklmlpard` as parameter.

```
# on slave terminal #
$ sklmlpard -slave 127.0.0.1:52714

# on master terminal #
$ sklmlpard -master 1
sklmlpard: listenning for slaves on 0.0.0.0:52714
sklmlpard: slave accepted (127.0.0.1)
sklmlpard: listenning for sklml program on 0.0.0.0:43044
```

When the slave `sklmlpard` is successfully connected to the master, this one displays an informative message. And, when all slaves are connected to the master (just after the first one in the example above), the master `sklmlpard` is ready to handle requests of a Sklml application on the address dumped.

The `sklmlpard` program has more than two options, it can be tweaked in many ways, details can be found in its manual page available in the standard Sklml distribution.

Chapter 3

Real programming with Sklml

3.1 Stream handling

When reified as standard OCaml functions, the Sklml skeletons become functions acting on stream. As explained before, this reification is achieved using the `parfun` function. Sklmlstreams are values of an abstract type of the `Sklstream` module. They intend to represent the stream of data transmitted on the network and are heavily used inside the Sklml runtime for this purpose.

There are several ways to create a Sklml stream, figure 3.1 summarizes the different ways to generate and consume a stream.

Function	Type	Description
`singleton`	`'a -> 'a stream`	Generate a one element stream.
`of_list`	`'a list -> 'a stream`	Transform a list into a stream.
`of_fun`	`(unit -> 'a option) -> 'a stream`	Make a stream of a function, if `None` is returned, ends the stream.
`of_vect`	`'a array -> 'a stream`	Transform an array into a stream.
`to_list`	`'a stream -> 'a list`	Transform a list into a stream.
`to_vect`	`'a stream -> 'a array`	Transform an array into a stream.

Figure 3.1: Generating and consuming streams.

3.2 Using the extra library

The Sklml distribution embeds a simple library intended to provide some useful tools which are not part of the core system.

3.2.1 The extra skeletons

Additional skeletons are in the `Skl_extra` module. Currently, this module contains some simple helpers.

1. The `rails_same` skeleton. It should be used when a `rails` computation needs to be used with the same skeleton.

2. The `if_then_else` skeleton. Given a skeleton to compute a predicate and two “branching” skeletons it executes either one or the other depending on the result of the predicate.

3.2.2 The domain decomposition toolkit

The SklmI system has been developed with a team of computational scientists working on parallelism using domain decomposition. A dedicated toolkit for such computations has been developed using the SklmI core and is provided with the standard distribution.

A domain decomposition algorithm performs a computation on a data set split into several parts. The global computation is achieved in several steps. At each step on each part of the data, one worker performs a computation and can send some data, which will be called *borders*, to some other workers. This sequence of steps is stopped when a global convergence criterion is reached.

Using the general description of domain decomposition above, the following type for the domain skeleton creator can be derived.

```
type 'a borders = ('a border) list
and 'a border = (int * 'a);;
type ('a, 'b) worker_spec = ('a borders, 'a * 'b) Skl.t * int list;;

val make_domain : (('a, 'b) worker_spec) array ->
  ('b array, bool) Skl.t -> ('a array, ('a * 'b) array ) Skl.t
;;
```

Note: the type `Skl.t` is the same type as the `skel` type in other code samples above.

The type definition defines a border as a pair of a user data and an integer. This additional integer identifies the worker which sent this border. A worker is specified with a value of type `worker_spec`, this value will be a pair storing the computational behavior of the worker as a standard skeleton and an integer list called the *connectivity table*. This table gives the list of workers whose borders are needed to do one computation step. The domain decomposition toolkit gives the guarantee that, at each step, one worker will get the borders of the workers given in its connectivity table. The skeleton representing the computational part of one worker takes as input a list of borders and returns its border data and one additional data of type β. This additional data is used by the convergence criterion which is also given as a skeleton taking an array of data of type β and returning a boolean. The looping will continue while the boolean returned by the convergence criterion is `true` and will stop when it becomes `false`. To use the domain decomposition toolkit, open the module `Make_domain`.

Note: the domain skeleton created by `make_domain` presents the same looping behavior as the `loop` skeleton, thus, in any cases, one computation step is done before testing using the convergence criterion.

Note: the `make_domain` helper is built using SklmI core skeletons, this is why it is an external library, the source of this helper is in the standard distribution in the file `src/extra/make_domain.ml`.

Bibliography

[1] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P^3L: A Structured High level programming language and its structured support. *Concurrency Practice and Experience*, 7(3):225–255, May 1995.

[2] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computations*. Research Monographs in Parallel and Distributed Computing. Pitman, 1989.

[3] R. Di Cosmo, Z. Li, M. Danelutto, S. Pelagatti, X. Leroy, P. Weis, and F. Clément. The CamlP3l programming language. Software and documentation available on the Web, `http://camlp3l.inria.fr/eng.htm`, 1999.

[4] M. Danelutto, R. Di Meglio, S. Orlando, S. Pelagatti, and M. Vanneschi. A methodology for the development and support of massively parallel programs. *Future Generation Computer Systems*, 8(1–3):205–220, July 1992.

[5] Marco Danelutto, Roberto Di Cosmo, Xavier Leroy, and Susanna Pelagatti. `OcamlP3l`: a functional parallel programming system. Technical Report 98-01, LIENS - DMI, Ecole Normale Supérieure, 1998.

[6] J. Darlington, A. J. Field, P. G. Harrison, P. H. J. Kelly, D. W. N. Sharp, and Q. Wu. Parallel Programming Using Skeleton Functions. In *PARLE'93*, pages 146–160. Springer, 1993. LNCS No. 694.

[7] S. Pelagatti. A methodology for the development and the support of massively parallel programs. Technical Report TD-11/93, Dept. of Computer Science – Pisa, 1993. PhD Thesis.

[8] S. Pelagatti. *Structured development of parallel programs*. Taylor&Francis, London, 1998.

[9] S. Pelagatti. Task and data parallelism in P3L. In Fethi A. Rabhi and Sergei Gorlatch, editors, *Patterns and Skeletons for Parallel and Distributed Computing*, chapter 6, pages 155–186. Springer-Verlag, London, 2002.

Index