Parallel Functional Programming with Skeletons: the OCamlP3L experiment

Marco Danelutto, Roberto Di Cosmo, Xavier Leroy, Susanna Pelagatti

Parallel Functional Programming with Skeletons: the ocamlp3l experiment

Marco Danelutto ° Roberto Di Cosmo • Xavier Leroy ⋆ Susanna Pelagatti °

° Dipartimento di Informatica
Università di Pisa
PISA – ITALY

• LIENS-DMI
Ecole Normale Supérieure
PARIS – FRANCE

⋆ INRIA Rocquencourt
Domaine de Voluceau
ROCQUENCOURT – FRANCE

Abstract

Writing parallel programs is not easy, and debugging them is usually a nightmare. To cope with these difficulties, a structured approach to parallel programs using skeletons and template based compiler techniques has been developed over the past years by several researchers, including the P3L group in Pisa.

This approach is based on the use of a set of primitive forms that are just functionals implemented via templates exploiting the underlying parallelism, so it is natural to ask whether marrying a real functional language like Ocaml with the P3L skeletons can be the basis of a powerful parallel programming environment. We show that this is the case: our prototype, written entirely in Ocaml using a limited form of closure passing, allows a very simple and clean programming style, shows real speed-up over a network of workstations and, as an added fundamental bonus, allows logical debugging of parallel programs in a sequential framework without changing the user code.

Key words: skeletons, functional languages, closures, parallelism.

1 Introduction

Functional programming languages have greatly improved since their appearance in the sixties. The performance achieved in functional program execution is now comparable to the performance achieved using classical imperative programming languages [17, 21]. In addition, modern functional programming languages have advanced features, like strong typing and advanced module systems, that improve both programmer productivity and code quality and maintainability.

But what about *parallel* functional programming? Since the very beginning it has been claimed that functional programming languages are implicitly parallel, mainly due to the possibility of using eager evaluation strategies for all the strict functions appearing in a program. Eager evaluation strategies, in conjunction with the referential transparency property sported by pure functional languages, allow functional languages compilers (or interpreters) to schedule in parallel the evaluation of all the parameters of a strict function call. Both the possibility of automatically exploiting this kind of *implicit* parallelism through parallel graph reduction or other compiler techniques [5, 19, 26, 15] and the possibility of providing the user with ways to *annotate* (somehow) functional programs in order to drive parallelism exploitation [18] have been explored, with different results. After Cole's work on skeletons [8], a new research track has been initiated concerning parallel functional programming with *skeletons*. As an example, Bratvold showed how skeletons can be looked for and exploited within an ML dialect [6], and Darlington's group at the Imperial College in London, started considering plain functional skeletons [12] and came up with the definition of a skeleton functional coordination language [14].

Some of the authors of this paper developed at the University of Pisa a skeleton parallel programming language, `p3l` [2]. `p3l` is actually an imperative programming language, although the skeleton framework used is completely functional. The `p3l` compiler uses an original template-based parallelism exploitation technique achieving very good performance as well as programmability and portability features [3]. The authors started looking at the possibility of exporting the template-based skeleton parallelism exploitation techniques to the functional world [11].

In the meanwhile, the Objective Caml functional programming language has been developed at INRIA Rocquencourt, in France. Objective Caml [21] (`ocaml` in the sequel) is a functional language of the ML family [24]. It supports both functions as first-class values and full imperative features, in particular arrays modifiable in-place. This combination of features makes it well adapted to skeleton-based programming: higher-order functions (functions taking user-provided functions as arguments) can be suitably used to model/implement skeletons, while the imperative features can be exploited to provide a parallel implementation of skeletons. Other useful features of `ocaml` include a powerful module system, allowing several implementations of the skeletons to be substituted for one another without changing the user code, and a built-in marshaler, allowing transmission of arbitrary data structures over byte streams, based on the same structural information used by the `ocaml` garbage collector.

The goal of the authors, at the beginning of the `ocamlp3l` project, was to assess the merits and the feasibility of the integration of the `p3l` language inside `ocaml`. This gave `ocamlp3l`, a programming environment that allows to write parallel programs in `ocaml` according to the skeleton model

[0]The `ocamlp3l` project has been partially funded by the Galileo bilateral France-Italy project n.97029. Further information on the `ocamlp3l`, Ocaml and `p3l` projects can be found at the URLs `http://www.di.unipi.it/~marcod/ocamlp3l/ocamlp3l.html`, `http://pauillac.inria.fr/ocaml/` and `http://www.di.unipi.it/~susanna/p3l.html`

supported by the parallel language p3l. It provides seamless integration of parallel programming and functional programming and advanced features like sequential logical debugging of parallel programs and strong typing, useful both in teaching parallel programming and in building full-scale applications. In addition, we wanted the skeleton implementation to run on widely available computer systems.

During the implementation of the system, it turned out that we could get more than that. In our implementation the user code containing the skeleton expressions can be linked with different modules in order to get either a parallel executable (running on a network of workstation) or a sequential executable (running on a single workstation). Therefore users are enabled to perform logical program debugging using the sequential version of the program and then to move to the parallel version, by just changing a linker option.

The high level of abstraction provided by functional programming, coupled with the ability to send closures over an I/O channel provided the key to an elementary and robust runtime system that consists of a very limited number of lines of code.

2 Skeletons

The skeleton parallel programming model supports structured parallel programming [8, 12, 10]. Using this model, the parallel structure/behavior of an application has to be expressed by using *skeletons* picked up from of a set of predefined ones, possibly in a nested way. Each skeleton models a typical pattern of parallel computation and it is parametric in the computation performed in parallel. Skeletons can be understood as second order functionals that model the parallel computation coming out from the application of a given parallelism exploitation pattern to the parameter functions.

As an example, a common skeleton is the *pipeline* one. Pipeline is a *stream parallel* skeleton. The parallelism exploited comes from performing in parallel computations relative to different input data sets, appearing on the input data stream. In particular the pipeline skeleton models the parallelism coming from the parallel computation of different *stages* of a function onto different input data items appearing on the input data stream. In other words, `pipeline`$(f_1, \ldots, f_n)$ computes the function $f(x) = f_n(f_{n-1}(\ldots f_1(x)\ldots))$ over all the data items $x_1, \ldots, x_k$ appearing onto the input data stream. Parallelism comes from computing in parallel $f_1(x_i)$, $f_2(f_1(x_{i-1}))$, $\ldots$, $f_n(f_{n-1}(\ldots f_1(x_{i-n+1})\ldots))$.

In general, a skeleton programming model provides a set $\mathcal{S}$ of skeletons that the programmer can use to model the parallel behavior of an application. Simpler skeleton models, such as the original models by Cole and Darlington [8, 12] do not allow skeleton composition. Therefore, the programmer must simply pick up the "best" skeleton from the set $\mathcal{S}$ and then provide the function parameters as sequential code. As an example, a program exploiting pipeline parallelism can be expressed by using the pipeline skeleton and by providing sequential code for the f_i appearing in the `pipeline` call. Other skeleton models, such as p3l[2], allow full skeleton composition. In this case, a *sequential* skeleton is included in $\mathcal{S}$ and the function parameters of any other skeleton must also be skeletons. Therefore programmers can either provide the function parameters of a pipeline by using the sequential skeleton to encapsulate sequential code, or by using other skeletons (including the `pipeline` one) to express any of the pipeline stages.

In ocamlp3l, we allow skeletons to be arbitrarily composed, and we provide the user with both stream parallel and data parallel skeletons (data parallelism being the one coming from performing in parallel computations relative to a single input data item).

The skeleton set of ocamlp3l contains most of the skeletons appearing in p3l, although some of the ocamlp3l skeletons are actually simplified versions of those appearing in p3l: (here, we will denote by f the function computed by skeleton F on a single data item of its input data stream):

Farm skeleton The farm skeleton, written `farm`, computes in parallel a function f over different data items appearing in its input stream. From a functional viewpoint, given a stream of data items $x_1, \ldots, x_n$, `farm`(F, k) computes $f(x_1), \ldots, f(x_n)$ (F being the skeleton parameter). Parallelism is exploited by having k independent, parallel processes computing f on different items of the input stream.

Pipeline skeleton The pipeline skeleton, denoted by the infix operator `|||`, performs in parallel the computations relative to different stages of a function over different 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 belonging to the input stream. Parallelism is exploited by having n independent parallel processes. Each process computes a function f_i over the data items produced by process computing f_{i-1} and delivers results to process computing f_{i+1}.

Map skeleton The map skeleton, written `mapvector`, computes in parallel a function over all the data items of a vector, generating a new vector. Therefore, for each vector X appearing in the input data stream, `mapvector`(F, n) computes the function f over all the items of the vector, using n different, parallel processes computing f over distinct vector items.

Reduce skeleton The reduce skeleton, written `reducevector`, folds a binary, associative function over all the data items of a vector. Therefore, `reducevector`(F, n) computes $x_1 f x_2 f \ldots f x_n$ out of the vector $x_1, \ldots, x_n$, for each one of the vectors appearing in the input data stream. The computation is performed using n different, parallel processes computing f.

We also included a sequential skeleton, written `seq`, that is only used to transform a plain function into a skeleton, in such a way that sequential code can be used as a skeleton parameter. It's worthwhile to point out that the choice of the skeletons of ocamlp3l pretends to be neither complete nor definitive. Our goal was to investigate the feasibility of integrating the skeleton programming model within a functional programming model. Therefore new skeletons can be added to ocamlp3l in the future.

2.1 An example

In order to understand how an ocamlp3l application can be written, let suppose that we want to develop a parallel application plotting the Mandelbrot set. Provided that we have defined ocaml functions:

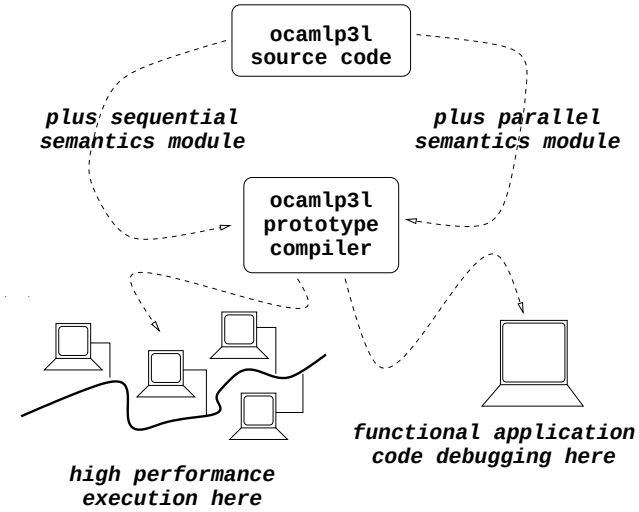

Figure 1: Overall design of the `ocamlp3l` prototype compiler

1. computing the color of a pixel in the set out of its coordinates,
2. displaying a color pixel of given coordinates on the display,
3. opening the graphic display,
4. closing the graphic display after a user acknowledge,
5. generating a pixel coordinate record each time the function is called and raising an `End_of_file` exception when there are no more pixels to generate (this function keeps state in a global, mutable variable),

then we can write the application as follows:

```
(* link skeleton implementation code *)
open Parp3l
open Nodecode
open Template
(* suitable, plain ocaml, sequential code *)
let color_pixel       coords      = ... ;;
let display_pixel    (coords,col) = ... ;;
let open_display     ()           = ... ;;
let close_display    ()           = ... ;;
let generate_a_pixel ()           = ... ;;
let dummy_fun        ()           = ()  ;;
(* set up the parallel program structure *)
let mandelprogram () =
  startstop
    (generate_a_pixel,dummy_fun)
    (display_pixel,open_display,close_display)
    (farm(seq(color_pixel),10)) in

    pardo mandelprogram;;
```

where:

- the `open` directives just tell the `ocaml` compiler to compile and link other `ocaml` modules (see Section 3)
- the `startstop` expression sets up a framework suitable for generating an input stream to the skeleton program (first parameter couple: the first function generates each item of the input stream, the second function initializes the input stream), for processing the output stream produced by the skeleton program (second parameter tuple: the first item is the function processing each output stream item, the second and the third provide to initialize and finalize the output stream handling process), and for evaluating a given skeleton program (the third parameter of `startstop` provides the actual skeleton program)
- the `pardo` expression initiates the program evaluation, according to the semantics defined by the modules included with the `open` directives. In the case of the code shown above, the farm skeleton is evaluated in parallel setting up a network of `10` independent processes computing the color of the pixels, plus a scheduler process and a process collecting and displaying colored pixels on the screen. In case the user included the sequential skeleton implementation module with the directive `open Seqp3l`, instead, the farm skeleton is computed within a single, sequential process.

3 `ocamlp3l` implementation

We implemented a prototype compiler generating executable code out of the `ocamlp3l` source code. The prototype compiler is entirely written in `ocaml` and is made out of a set of modules. Depending on the set of modules linked with the user code, the compiler either produces code that can be run on a single workstation or code that can be run in parallel onto a network of workstation. In both cases the code can either be `ocaml` byte-code or native code (see Figure 1).

In case of sequential execution, the modules linked to the user code contain second order functions for each one of the skeletons used. These functions actually implement the sequential semantics of the skeletons, i.e. sequentially compute the result of applying the skeleton to the parameters (see Figure 2 left).

In case of parallel execution, the modules linked to user code contain second order functions for each one of the skeletons that build an abstract skeleton tree out of the user code. The skeleton tree is traversed, and a (distributed) process network is logically assigned for the execution of each skeleton. Therefore a logical process graph is obtained. Then, this graph is traversed and logical channels (actually Unix sockets) are assigned as the input/output channels of each one of the processes. Finally, a closure is derived relative to each one of the processes in the graph. These closures are computed looking at the functions stored in the template library. Each one of this functions models the behavior of one of the processes implementing a skeleton. E.g. there are template process functions modeling a pipeline stage, a farm worker, a farm emitter process, etc. When the parallel program is eventually to be run onto a workstation network, these closures will be used to specialize each one of the processes executed on a workstation, in such a way that the collection of processes participating in the program execution actually implement the logical process network derived from the skeleton tree (see Figure 2 right).

More details on the `ocamlp3l` compiler can be found in [9]; in the following sections we will discuss the main features of the prototype compiler.

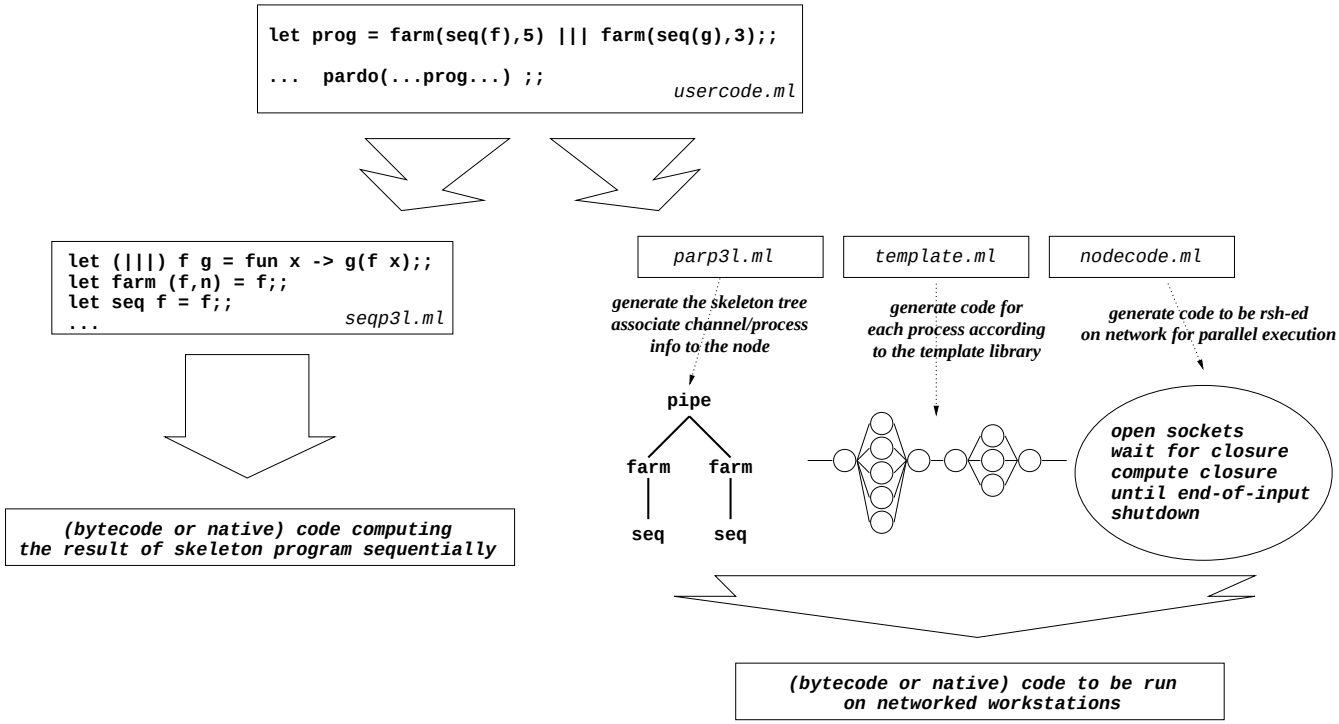

Figure 2: Compiler algorithm sketch: sequential code generation (left) parallel code generation (right)

3.1 Closure passing as distributed higher order parameterization

In order to implement parallel skeleton execution, we choose to use an SPMD (Single Program Multiple Data) approach: all the nodes of the network will run the *same* program, namely a "process template" interpreter. Each one of the nodes will be specialized, by information provided by a special "root" node during the initialization process, with the process template code that it must run. The specialization information includes the closure of the function computed by the node as well as the communication channels the process must use. Overall, the specialized node collection implements a process network implementing parallel execution of the skeletons provided in the user code.

Therefore, the root node performs the following tasks:

- executes the skeleton expressions of the user program. As a consequence a data structure describing the process network is built, which is used to compute the configuration information for each node in the process network.
- maps virtual nodes onto the pool of available machines,
- initializes a socket connection with all the participating nodes,
- gets a local port addresses from each of them,
- sends out to each node the addresses of its connected neighbors (these three steps provide an implementation of a centralized deadlock free algorithm interconnecting the other nodes according to the process network specified by the skeleton expression),
- sends out the specialization information (the *function* it must perform) to each node.

This very last task requires a sophisticated operation: sending a *function* (or a closure) over a communication channel. Section 3.2 discusses the implementation of this operation.

On the other side, all the nodes different from the root one simply wait for a connection to come in from the root node, then send out the address of the local socket port they allocate to do further communication, wait for the list of neighbors and for the specialization function, and then simply perform the specialization function until termination.

3.2 Marshaling of function closures

Most garbage-collected languages provide a built-in marshaler (also called serialization) that takes any data structure and encodes it as a sequence of bytes, suitable for transmission over a communication channel; the symmetric unmarshaling primitive then rebuilds the data structure at the receiving end. Marshaling exploits the same run-time type information that guides the garbage collector.

In functional languages, marshaling is often restricted to "pure" data structures that do not contain any function closures. Marshaling function closures is delicate because it is unclear how to transmit the code pointers embedded in the closures. In distributed programming, the usual solution is not to transmit the code of functions, but rebuild a proxy function on the client side. The proxy transparently sends its argument back to the originating site, where the function application is evaluated, and its result sent back to the caller. This is the traditional remote procedure call or network ob-

jects [23, 4]. However, this solution is not appropriate for us, as it prevents any parallelism: all function evaluations are performed sequentially on the originating site.

To allow parallel evaluation in a general distributed setting, actual code mobility is required: the code of the function must be transmitted "on the wire". This puts strong constraints on the code, which must be fully position-independent and relocatable. In particular, all branches must be PC-relative, and references to global variables must be avoided by putting the global variables in the function closure. Some bytecodes have been designed that fulfill these requirements [7, 16]. However, neither the `ocaml` bytecode nor the native code generated by `ocamlopt` are fully relocatable. Workarounds involving sending relocation information along with the code have been proposed [20], but are quite complex. The size of the transmitted code is also a concern in many applications.

For `ocamlp3l`, a much simpler solution exists. We are doing SPMD applications instead of general distributed applications: the same program text is running on all communicating machines. (The application is statically linked, and we disallow dynamic loading of object files.) Hence, we can simply send a code address on the wire (as offsets from the beginning of the code section), without sending the piece of code pointed to by the address, and be certain that this code address will point to the correct piece of code in the receiver. To guard against code mismatches, the sender transmits an MD5 checksum of its own code along with the code address: this allows the receiver to check quickly that it is running the same code as the sender. (This check proved very useful to guard against inconsistencies when accessing the same NFS-mounted file system from different workstations.) We extended the standard `ocaml` marshaler to support this form of closure sharing; the changes are now integrated in the current `ocaml` distribution.

Like all code mobility schemes, our marshaling of closures require that the sender and receiver run the same instruction set. This is no problem if the application is compiled with the `ocaml` bytecode compiler, since the bytecode is platform-independent. If the application is compiled with the native-code compiler, communication is restricted to machines having the same processor architecture. In other terms, we restrict the SPMD paradigm to "single binary executable program, multiple data".

To summarize, the possibility of sending closures in the implementation allowed us to obtain a form of higher order distributed parameterization that keeps the runtime code to a minimum size (the source codes of the full system is less than 20Kbytes).

3.3 Communication and process support

According to the initial goals of the `ocamlp3l` project, we looked for a simple, portable and reliable communication system. We were interested in coming up with a solution that can be actually used onto widely available, low cost systems. Eventually we chose to use plain TCP/IP sockets. First of all, this choice allowed both the Unix world and the Windows world to be addressed. Second, no particular customization of the support is needed to match the `ocamlp3l` features. Finally, the point-to-point, connection oriented, stream model provided by Unix sockets is perfect to model data streams of `ocamlp3l`. On the down side, the adoption of Unix sockets presents an evident disadvantage which is the low performance achieved in communications. This disadvantage, however, simply implies that parallelism can be usefully exploited only when the communication/computation ratio is small. It does not affect the overall features of prototype.

As far as the process model is concerned, all we need is a mechanism allowing an instance of the template interpreter to be run onto different workstations belonging to a local area network. The Unix `rsh` mechanism matches this requirement, and a similar mechanism can be used within the Windows environment. Note that, as processes are generated and run on different machines just at the beginning of the `ocamlp3l` program execution, any considerations about performance in `rsh`-ing processing is irrelevant.

As an alternative to TCP/IP sockets, we are also considering implementing `ocamlp3l` on top of the MPI library for message passing in distributed-memory multiprocessors [22]. On networks of workstations, MPI is implemented on top of TCP/IP sockets and therefore has no advantages over our hand-written implementation. However, vendors of distributed-memory supercomputers (such as the Cray T3E and SGI's and Digital's clusters of multiprocessors) provide implementations of MPI that take advantage of the fast, custom interconnection networks of those machines. `ocamlp3l` over MPI would thus benefit from these custom communication hardware without sacrificing portability. Another interesting feature of MPI is the group communication primitives (broadcast, scatter, gather, reduce) that can be implemented very efficiently on certain communication topologies.

3.4 Template implementation

Within `ocamlp3l`, the skeleton instances of a program are implemented by using *implementation templates*, i.e. process networks implementing the parallel semantics of the skeleton, following the approach adopted within the `p3l` compiler [2]. Figure 3 shows the implementation templates used. Each "white" circle represents a whole process network computing a skeleton. In case the skeleton is `seq`(F) the network has a single process computing the sequential code of the function f over all the data items appearing onto the input stream. Each "black" circle represents a process generated by the `ocamlp3l` implementation, aimed at either coordinating parallel computation or at merging/splitting the data streams according to the skeleton semantics. Finally, arrows represent communication channels (hosted by Unix sockets) implementing data streams.

Within `ocamlp3l` these implementation templates can be nested to any depth, reflecting the full composability of the skeletons provided to the programmer. Each template appearing in `ocamlp3l`:

- is parametric in the parallelism degree exploited. As an example the farm template may accommodate any numbers of worker processes.
- is parametric in the skeleton computed as the body of the skeleton. As an example, the `farm` template is specialized by the skeleton representing the `farm` worker.
- provides a set of *process templates* i.e. parametric process specifications that can be instantiated to get the real process codes building out the each one of the processes participating in the parallel evaluation of a `ocamlp3l` program. Such process templates are provided as functions in one of the library files

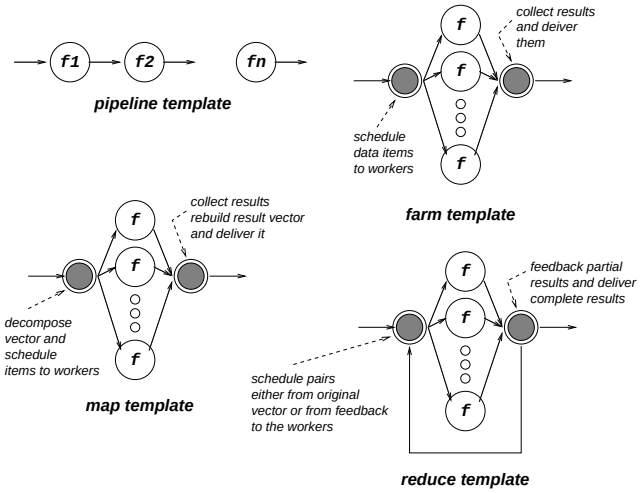

Figure 3: Templates used to implement the `ocamlp3l` skeletons

(`template.ml`) of `ocamlp3l`. Parameters to the process template include the input and output stream (streams) specification as well as the "user" function to be computed on the items appearing onto the input stream in order to get the data items that have to be delivered onto the output stream.

3.5 Template based compilation

The whole compilation process transforming an `ocamlp3l` skeleton program into the parallel process network implementing the program can be summarized in three steps (see Figure 2 right):

- first, the skeleton code is transformed into a skeleton tree data structure, recording all the significant details of the skeleton nesting supplied by the user code
- then, the skeleton tree is traversed and processes are assigned to each skeleton according to the implementation templates. During this phase, processes are denoted by their input/output channels, identified via a unique number
- finally, once the number and the kind of parallel processes building out the skeleton code implementation is known, code is generated that either delivers the proper closures, derived by using the process templates, to the "template interpreter" instances running on distinct workstations ("root" node), or waits for a closure and repeatedly computes this closure on the proper input and output channels until an `EndOfFile` mark is received (non-"root" nodes). Closures are sent to the various template interpreter nodes in a round-robin way. This policy will be changed in the next versions of `ocamlp3l`, in order to be able to achieve a better load balancing.

4 Program development with `ocamlp3l`

In order to develop a new parallel application using `ocamlp3l` the user is expected to perform the following steps:

- develop skeleton code modeling the application at hand. This just requires a full understanding of the skeleton semantics and usually allows the user to reuse consistent portions of existing applications written in plain `ocaml`.
- test the functionality of the new application by supplying relevant input data items and looking at the results computed using the sequential skeleton semantics. In case of problems, the user may run the sequential debugging tools to overcome the problem.
- link the parallel skeleton semantics module and run the application onto the workstation network. Provided that the application was sequentially correct, no new errors will be found at this step, assuming the runtime support is correct.
- look at the performance results achieved by running the application on the number of processing nodes available and possibly modify the program either by adjusting the significant performance parameters (such as the parallelism degree of the `farm`, `mapvector` and `reducevector`), or by changing the skeleton nesting used to exploit parallelism (insert `farm`s, split `pipeline` stages, etc.).

Therefore, the `ocamlp3l` user developing a parallel application is not involved in any one of the error prone, boring and heavy activities usually related to parallel programming (process scheduling, explicit communication handling, buffer management, etc.). These "details" are completely hidden within the compiler libraries/code.

Performance debugging (or *tuning*) is the activity a user is supposed to perform in order to get the last MIPS out of his code running on the parallel machine at hand. In order to perform performance debugging the programmer must look out for bottlenecks in the computation of his program and take actions aimed at removing such bottlenecks. An `ocamlp3l` programmer may look at the times spent in the computation of the different stages of his skeleton program and try to understand if there is some stage which behaves as a bottleneck. Once individuated the stage he can perform different actions:

- if the stage is already parallel (e.g. it is a `farm`) the user can augment the parallelism degree of the stage (e.g. put a bigger `int` as the second parameter of the `farm` call)
- if the stage is not parallel, the user can figure out strategies to parallelize it by using the proper skeletons.

In both cases, the user is only asked to modify the parallelization strategies of the program, either in the quality or in the quantity of parallelism exploited. He is not asked to modify process or communication code, which is the thing normally happening when using other parallel programming approaches.

5 Preliminary results

The preliminary results we obtained concern both programmability and performance.

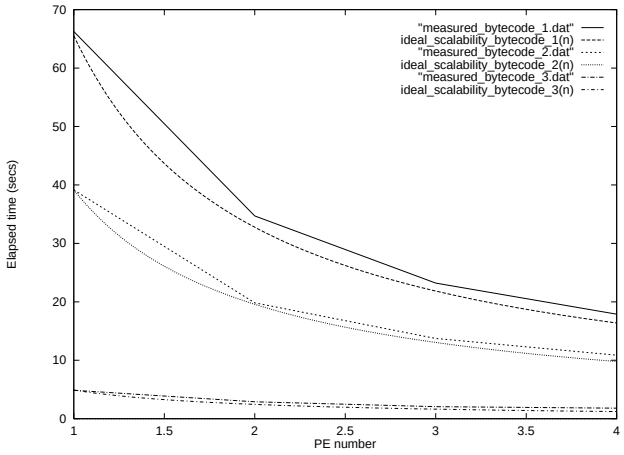

Figure 4: bytecode application scalability (farm/map template)

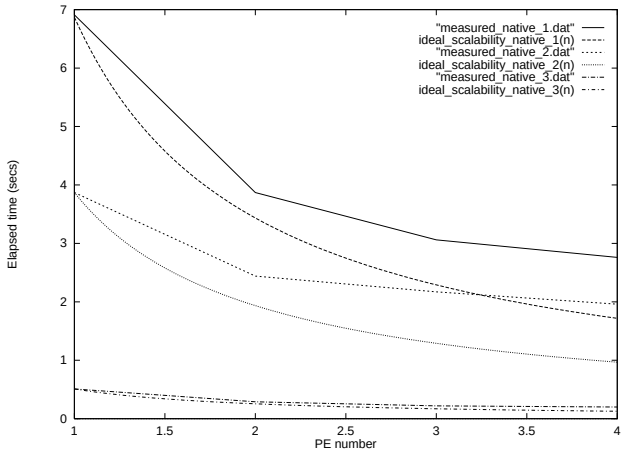

Figure 5: native code application scalability (farm/map template)

Programmability Parallel applications can be easily developed in `ocamlp3l`. Once a programmer has a clear idea of the skeletons available, it takes a very short time to transform a sequential code into a skeleton program. Times involved in designing a parallel application when the programmer has explicitly to deal with communications, synchronization and process handling are orders of magnitude higher. The major activity a user has to perform when developing `ocamlp3l` applications is performance debugging/tuning. However, changing either the parallel structure or the parallelism degree of an `ocamlp3l` application is a very simple and fast task. Therefore, the user is both encouraged to experiment different parallel structures of his application, and supported in the performance tuning activity. Overall, we experimented times of minutes to transform sequential `ocaml` code into running parallel `ocamlp3l` applications and times of minutes/hours to perform performance tuning of an `ocamlp3l` application.

We have developed several test applications, in order to validate the `ocamlp3l` prototype and to evaluate its performance figures. These applications include well-known applications, such as Mandelbrot set computation and matrix multiplication as well as more complex applications, in particular a protein folding application currently developed by a research team at the University of La Plata in Argentina. This application computes the three-dimensional folding shape of protein chains.

Performance Preliminary results concerning performance are encouraging. We measured actual speedups when running the parallel code on small workstation networks. These speedups turned out to be almost linear when the communication/computation ratio of each process belonging to the process network implementing the `ocamlp3l` program was suitably small. Figure 4 and 5 show the scalability of a very simple application, just exploiting a farm skeleton. Both Figures plot application completion time (in seconds) relative to the same application operating on differently sized datasets (the upper curves referring to bigger datasets). Figure 4 plots completion time of application code compiled to bytecode, whereas Figure 5 is relative to the same applications compiled to native code. Scalability is close to ideal one (i.e. the speedup is almost linear) although in the bytecode runs the measured times are closer to ideal ones than in native code runs. This is due to the fact that in the second case, local computations are performed faster than in the first case, while interprocess communication always take the same time (therefore communication/computation ratio is higher). However, if we decrease the communication/computation ratio by augmenting the data set size, we get closer ideal/measured curves even in case of native code compiled applications (see Figure 6, relative to the same application of Figures 4 and 5 run onto a larger dataset). All these Figures plot completion times relative to application run on a network of up to four homogeneous (same processor, a 233 Mhz Pentium II, same memory, cache and disk size and speed, etc.) machines.

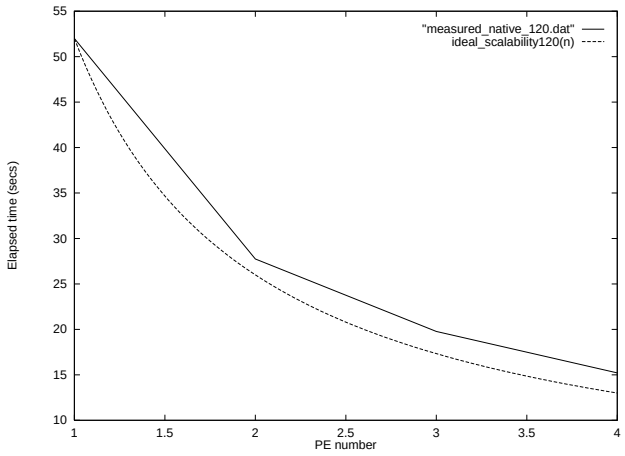

Figure 6: Scalability: farm/map template scalability (native code) low communication/computation ratio

We also measured the overhead introduced by `ocamlp3l` modules. First of all, we run a simple farm application com-

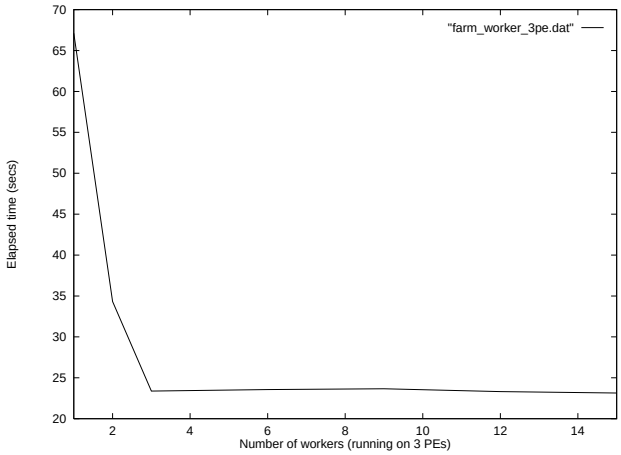

Figure 7: Number of workers in a farm/map template vs. completion time (native code)

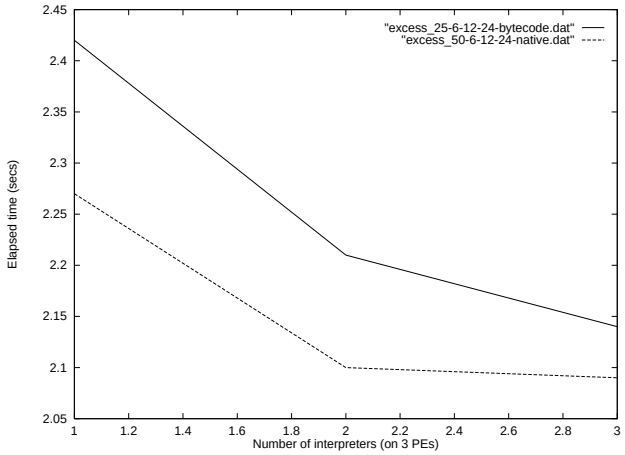

Figure 8: Instances of the template interpreter per node vs. completion time (native and bytecode)

piled by linking the parallel semantics module and with a parallelism degree 1 specified in the farm code. Then we run the same application compiled by linking the sequential semantics module. Finally we run a plain `ocaml` application computing the same algorithm and using a list to emulate the task stream. All the three applications were run onto the same machine. We observed that in bytecode runs the parallel semantics module introduced a rough 1% overhead, while in the native code runs it introduced a 1.5% overhead with respect to the runs using the sequential module. No sensible overhead was introduced by sequential semantics module with respect to the hand-written ML application code.

Finally, we evaluated the effect of "excess parallelism" on completion time. We considered a simple application just exploiting farm parallelism and we measured the completion time in two cases:

- varying the parallelism degree of the farm template, keeping the number of PEs used to execute the application constant (Figure 7)
- varying the number of instances of the template interpreter process per PE, keeping the number of PEs used to execute the application constant (Figure 8)

In the former case, the excess parallelism does not increase the application completion time and slightly smaller completion times have been measured (the gain is below 1%, however). In the latter case, we observed a better behavior: the maximum decrease in completion time was around 15%.

The performance results discussed here are relative to a very small workstation network and to simple application code. Indeed, we actually used in this applications the template which implements farm, map and reduce skeletons, i.e. the most critical one, and we fully experimented the whole runtime system based on closure passing.

As expected, the results showed that scalability actually depends on the amount of computation performed after receiving a single task (i.e. on the already mentioned communication/computation ratio). Therefore, when moving to larger networks, we expect to get good performances provided the communication/computation grain of the application matches the network features.

We also run `ocamlp3l` applications on larger Linux PC networks. However, the differences in CPUs among the PC used make impossible to use these runs to derive suitable performance data.

6 Related work

Many researchers are currently working on skeletons and most of them are building some kind of parallel implementation.

In particular, Darlington's group at Imperial College in London is actively working on skeletons. They have explored the problems relative to implementing a skeleton programming system, but the approach taken uses an imperative language as the implementation language, at least for the code implementing the processes running on the parallel machine nodes [13, 1]. Bratvold [6] takes into account plain ML programs and looks for skeletons within them, compiling these skeletons by using process networks that look like implementation templates. However, both the final target language and the implementation language are imperative. Finally, Serot [25], presents an embedding of skeletons within `ocaml` that apparently looks like to be close to our work. The message passing is performed by interfacing the MPI [22] library with `ocaml`, rather than using sockets, and the skeletons taken into account are slightly different from ours. However, the important difference is that these skeletons cannot be nested. On the one hand, this allows to implement the skeletons by a simple library directly calling MPI. On the other hand, the expressive power of the language is much lower than the expressive power of `ocamlp3l`.

7 Conclusions

In this paper we showed how a skeleton parallel programming model such as the one provided by `p3l` can be successfully merged within the functional programming environments such as the one provided by `ocaml`. In particular, we discussed how skeletons can be embedded within `ocaml`

as second order functions and how modules implementing both the sequential and the parallel skeleton semantics can be supplied that allow users to write, functionally debug and run in parallel skeleton applications using `ocaml` to express sequential computations *and* data types. The whole process preserved the strong typing properties of `ocaml`.

At the moment, the prototype `ocamlp3l` implementation runs under Linux, uses sockets to implement communication, and preliminary results show that the embedding of skeletons within the `ocaml` programming environment is feasible and effective. In the near future we want to adopt a more efficient communication layer, possibly by using MPI [22] instead of the Unix socket library. At the same time, we are porting the system on the ubiquitous Windows systems, for didactic purposes. We also wish to extend the set of skeletons supported, in particular data-parallel skeletons operating on matrices. Finally, we are currently trying to restructure the code of `ocamlp3l` in such a way that new skeletons may be easily added (at the moment the insertion of a new skeleton requires both to modify some fundamental data structures and to add the proper code to the process templates library and to the parallel and sequential semantics modules).

References

[1] P. Au, J. Darlington, M. Ghanem, Y. Guo, H. To, and J. Yang. Co-ordinating heterogeneous parallel computation. In L. Bouge, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Europar '96*, pages 601–614. Springer-Verlag, 1996.

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

[3] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. Summarising an experiment in parallel programming language design. In B. Hertzberger and G. Serazzi, editors, *High-Performance Computing and Networking*, volume 919 of *Lecture Notes in Computer Science*, pages 8–13. Springer-Verlag, 1995.

[4] A. Birrell, G. Nelson, S. Owicki, and E. Wobbler. Network objects. Technical Report 115, DEC Systems Research Center, 1994.

[5] G. E. Blelloch and J. Greiner. Parallelism in sequential functional languages. In *Functional Programming & Computer Architecture*, pages 226–237, June 1995.

[6] T. Bratvold. *Skeleton-Based Parallelisation of Functional Programs.* PhD thesis, Heriot-Watt University, 1994.

[7] L. Cardelli. Amber. In *Combinators and Functional Programming Languages*, volume 242 of *Lecture Notes in Computer Science*. Springer-Verlag, 1986.

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

[9] M. Danelutto, R. D. Cosmo, X. Leroy, and S. Pelagatti. OcamlP3l a functional parallel programming system. Technical Report LIENS-98-1, E.N.S. Paris, 1998. Available on WEB at `http://www.dmi.ens.fr/EDITION/preprints/Index.liens.98.html`.

[10] M. Danelutto, R. D. 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.

[11] M. Danelutto and S. Pelagatti. Parallel Implementation of FP using a Template-Based Approach. In *Proceedings of the 5th International Workshop on Implementation of Functional Languages*, pages 7–21, September 1993. Nijmegen – The Neaderlands.

[12] J. Darlington, A. J. Field, P. Harrison, P. H. J. Kelly, D. W. N. Sharp, Q. Wu, and R. L. While. Parallel Programming Using Skeleton Functions. In M. R. A. Bode and G. Wolf, editors, *PARLE'93 Parallel Architectures and Langauges Europe.* Springer Verlag, June 1993. LNCS No. 694.

[13] J. Darlington, Y. Guo, H. W. To, Q. Wu, J. Yang, and M. Kohler. Fortran-S: A Uniform Functional Interface to Parallel Imperative Languages. In *Third Parallel Computing Workshop (PCW'94).* Fujitsu Laboratories Ltd., November 1994.

[14] J. Darlington, Y. Guo, H. W. To, and J. Yang. Parallel Skeletons for Structured Composition. In *Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* ACM Press, July 1995.

[15] M. Feeley and J. S. Miller. A parallel virtual machine for efficient Scheme compilation. In *Proceedings of the ACM Conference on LISP and Functional Programming*, pages 119–130, June 1990. Nice – France.

[16] C. Fournet, G. Gonthier, and L. Maranget. The join calculus language. Software, documentation and related papers available on the Web, `http://join.inria.fr/`, 1997.

[17] P. Hartel and K. Langendoen. Benchmarking implementations of lazy functional languages. In *Functional Programming & Computer Architecture*, pages 341–349, Copenhagen, June 93.

[18] P. Hudak. Para-Functional programming in Haskell. In B. K. Szymansky, editor, *Parallel Functional Language and Compilers*, chapter 5, pages 159–195. Addison-Wesley, 1989.

[19] S. L. P. Jones. Parallel Implementations of Functional Programming Languages. *The Computer Journal*, 32(2):175–186, 1989.

[20] F. C. Knabe. *Language Support for Mobile Agents.* PhD thesis, School of Computer Science, Carnegie Mellon University, December 1995.

[21] X. Leroy, J. Vouillon, and D. Doligez. The Objective Caml system. Software and documentation available on the Web, `http://pauillac.inria.fr/ocaml/`, 1996.

[22] M.P.I.Forum. Document for a standard message-passing interface. Technical Report CS-93-214, University of Tennessee, November 1993.

[23] A. Ohori and K. Kato. Semantics for communication primitives in a polymorphic language. In *20th symp. Principles of Programming Languages*, pages 99–112. ACM Press, 1993.

[24] L. C. Paulson. *ML for the working programmer.* Cambridge University Press, 1991.

[25] J. Serot. Embodying Parallel Functional Skeletons: An Experimental Implementation on Top of MPI. In G. Lengauer, Griebl, editor, *Proceedings of the Euro-Par'97, Parallel Processing*, pages 629–633. Springer Verlag, 1997. LNCS, no. 1300.

[26] C. Walinsky and D. Banerjee. A data-parallel FP compiler. *Journal of Parallel and Distributed Computing*, 22(1):138–153, 1994.