Skeletons for parallel image processing: an overview of the SKiPPER project

Jocelyn Serot, Dominique Ginhac

Skeletons for parallel image processing : an overview of the SKiPPER project

Jocelyn Sérot [a,*] Dominique Ginhac [b]

[a] *LASMEA, UMR 6602 CNRS, University Blaise Pascal de Clermont-Ferrand, Campus des Cézeaux, F-63177 Aubière, France. E-mail: Jocelyn.Serot@lasmea.univ-bpclermont.fr*

[b] *LE2I, FRE 2309 CNRS, University of Burgundy, F-21078 Dijon, France. E-mail: dginhac@u-bourgogne.fr*

Abstract

This paper is a general overview of the SKIPPER project, led at Blaise Pascal University between 1997 and 2002. The main goal of the SKIPPER project was to demonstrate the applicability of *skeleton-based* parallel programming techniques to the fast prototyping of reactive vision applications. This project has produced several versions of a full-fledged integrated parallel programming environment (PPE). These PPEs have been used to implement realistic vision applications, such as road following or vehicle tracking for assisted driving, on embedded parallel platforms embarked on semi-autonomous vehicles. All versions of SKIPPER PPEs share a common front-end and repertoire of skeletons – presented in previous papers – but essentially differ in the techniques used for implementing skeletons. This paper focuses on these implementation issues, by making a comparative survey, according to a set of four criteria (efficiency, expressivity, portability, predictability), of these implementation techniques. It also gives an account of the lessons we have learned, both when dealing with these implementation issues and when using the resulting tools for prototyping vision applications.

Key words: Parallelism, skeleton, computer vision, fast prototyping, data-flow

1 Introduction

The general context of the SKIPPER project is the development of realistic vision applications for embedded platforms. These applications may be found

* Correspondance to J. Sérot

for instance in remote inspecting robots or vehicle equipped with assisted-driving systems, as presented in [25], [27] or in Fig. 1 (an assisted-driving application based upon vehicle tracking and road following). Although relying on algorithms and programming paradigms encountered in the mainstream of computer vision, these applications raise two specific issues. First, they implement *reactive systems*, operating "on the fly" on digital *streams* of images. This means that they must be able to absorb input data and output results at a minimum frequency and produce responses with a maximal latency. For assisted-driving applications, for instance, the typical frequencies are in the range of 10-30 frame/s and the maximal latency rarely exceeds 50 ms. Second, they must meet stringent operational constraints in terms of volume or power consumption, which often rules out implementations based upon stock-hardware.

Fig. 1. A reactive vision application for assisted driving

For these applications, the problem of maximizing the performances while coping with the operational constraints can be solved either by relying on dedicated hardware – such as digital signal processors (DSP), Field Programmable Gate Arrays (FPGA) or Application Specific Integrated Circuits (ASIC) – or by using a multi-processor (parallel) architecture. The *Transvision* platforms [19,15], built at LASMEA between 1992 and 1998, are examples of the second approach. These MIMD architectures built upon Transputer and Alpha processors could deliver significant computing power, provided built-in facilities for video i/o and could be embarked in a vehicle (Fig. 2).

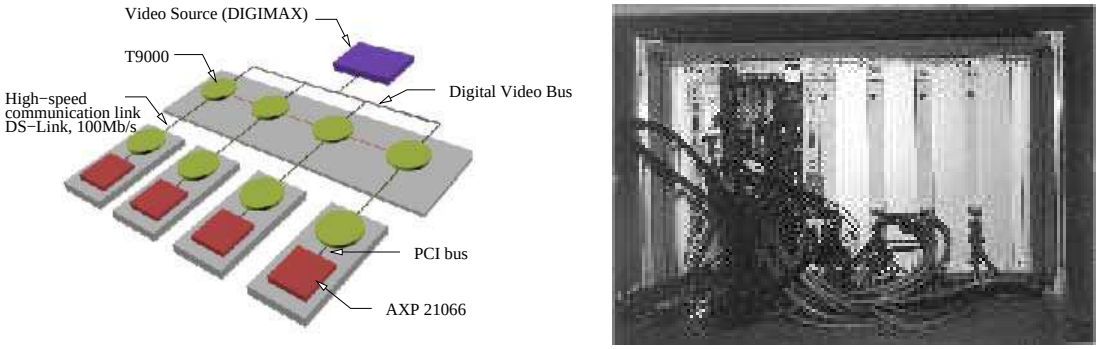


Fig. 2. The Transvision parallel platform

But relying on parallel machines place severe strains on programmers : in the absence of high-level parallel programming models and environments, they have to explicitly take into account every aspects of parallelism such as task partitioning and mapping, data distribution, communication scheduling or load-balancing. Having to deal with these low-level details results in long, tedious and error-prone development cycles – especially when the persons in charge of developing the algorithms are image processing, not parallel programming, specialists –, thus hindering a true experimental approach. For *reactive* applications, the problem is reinforced by the fact that the need to evaluate the *dynamic* properties of the algorithm at realistic frame-rate effectively rules out any prototyping phase solely based upon off-line, sequential simulation on stock hardware. Parallel programming at a low level of abstraction also limits code reusability and portability.

The SKIPPER project was developed in response of the aforementioned problems. Basically, its goal was to "capture" – in a efficient and portable way – the expertise gained by programmers when implementing reactive vision applications using low level parallel constructs, to make it readily available to algorithmicians and image processing specialists.

The SKIPPER programming methodology is based upon the concept of *algorithmic skeletons* [7,8]. Skeletons are high-level program constructs that abstract common patterns of parallel computation in a parametric way. Common examples of skeletons are *process farms*, *pipelines* and *divide-and-conquer trees*. With this approach, the structure of a parallel application is expressed only as combination/nesting of the skeletons provided. The programmer only specifies the *qualitative* aspects of parallelism. All quantitative aspects are dealt with by the compiler and/or the run-time system. The repertoire of skeletons therefore acts as a sort of "parallel toolbox" from which parallel programs can be built with a minimal concern for low-level details. In the case of SKIPPER, this repertoire was built "bottom-up", from a careful analysis of a large corpus of existing low-to-mid level vision applications hand-coded in parallel C. This retrospective abstraction process drew out four skeletons, called SCM (Split-Compute-and-Merge), DF (Data Farming), TF (Task Farming) and ITERMEM (ITERate with MEMory). A detailed description of these skeletons can be found in previous papers such as [27].

Between 1996 and 2001, four skeleton-based PPEs were built: SKiPPER-0, SkiPPER-1, SKiPPER-2 and SKiPPER-D. All these realizations share a common general architecture, sketched in Fig. 3. They use the same repertoire of skeletons [1], rely on a similar front-end and therefore look similar to the application programmer. They differ in the intermediate representation used for implementing the skeletons. The common features, shared by all realizations,

[1] The only noticeable exception is SKIPPER-0, for reasons given in Section 4.

will be recalled in Section 2. The discriminating features will be highlighted in Section 3 and each version of SKIPPER will be presented in turn in Sections 4, 5, 6 and 7.

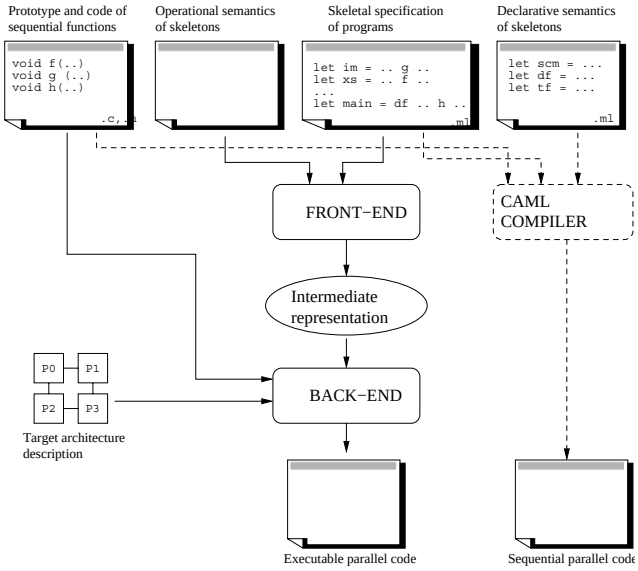


Fig. 3. Generic overview of SKIPPER PPEs

2 Common features in Skipper

Most of these features have been presented in previous papers, such as [27]. The following is therefore only a brief recall of the most salient ones.

Menu-driven approach. Within SKIPPER, like in P3L [3], Anacleto [6] or SkIE [4], skeletons are viewed as explicit indications to the compiler of where and which parallelism will be deployed. This approach can be contrasted, for instance, to the one used by Michaelson et al. [20,23,21], in which skeletons are viewed as *possible* realizations of common higher-order functions.

Program specification. The skeletal structure of parallel programs – ie. which skeletons are used, with which arguments and in what order – is explicited textually using a subset of the CAML language. Skeletons are represented as higher-order, polymorphic functions and programs are just sequences of value definitions expressing the data dependencies of the algorithm. Fig. 4, for example, shows a program making use of two SCM skeletons to binarize an image.

Here `row_block`, `histo`, `merge_histo`, `get_img`,`bimod`, `binar` and `display_img` are the application specific, sequential functions (written in C in our case). `row_block` decomposes an image into horizontal sub-images, `histo` computes the histogram of a (sub)image and `merge_histo` sums the partial histograms

```
let src = get_img 256;;
let h = scm 4 row_block histo merge_histo src;;
let th = bimod h;;
let res = scm row_block (binar th) block_row src;;
let main = display_img res;;
```

Fig. 4. A small program making use of the SCM skeleton

computed on each subimage into the final one. `bimod` and `binar` respectively computes and applies an optimal binarization threshold. The `get_img` and `display_img` functions respectively retrieves the next image from the video input stream and displays the binarized image on the screen.

Declarative semantics of skeletons. This declarative semantics is used to convey the "meaning" of a skeleton in a target-independent manner. It is also given in CAML. For the DF (Data Farming) skeleton, for example, this definition can be written as follows:

```
let df comp acc xs = foldl1 acc (map comp xs)
```

where `xs` is the list of data items to process, `comp` is the function applied to each item and `acc` performs the accumulation of partial results. `map` and `foldl1` are built-in higher-order functions for applying a function and iterating a binary operator over a list of elements, respectively. This definition states the skeleton declarative semantics in a purely applicative manner (as a combination of calls to its functional arguments), without any reference to an underlying execution model.

Application-specific, sequential functions are written in C. The possibility of using sequential functions written in C – although the parallel program specification makes use of a higher-order formalism (CAML) – is essential, since we don't want programmers to recode their algorithms from scratch (and especially in CAML). We share this (very pragmatic) concern with the P3L project for example.

Sequential emulation of parallel programs. This possibility results from the fact that, insofar as the declarative semantics of skeletons is given in CAML, it automatically confers a default sequential semantics to these skeletons. This default sequential semantics can be used to debug parallel programs on sequential stock-hardware before running them on the parallel target, thereby offering a means of separating "algorithmical" debug from 'implementational" debug. The idea is to first debug the sequential code on a sequential platform (with all the smart debugging tools available) so that runs on the parallel target platform are only used to test/assess (not debug) the influence of real environmental parameters on the application behavior. The merits of this approach have already be underlined by Danelutto *et al.* in [11], under the name "logical debugging".

3 Discriminating features

The discriminative features of SKIPPER successive versions lie in the *operational semantics* of the skeletons, ie. in the way these skeletons are implemented on the parallel target. This is obviously related to the *intermediate representation*, as suggested in Fig. 3. Four types of intermediate representation have been used: Synchronous Data Flow Graphs (SDFG), Process Network Templates (PNT), Hierarchical Task Graphs (HTG) and Dynamic Data Flow Graphs (DDFG). In the sequel, each of these representation we will surveyed, according to five criteria:

Run-time vs compile-time support. Some approaches (like those based on SDFG) rely on a sophisticated compile-time support to take most (if not all) decisions regarding the mapping and scheduling of the application-specific sequential functions, whereas for others these decisions are taken at run-time by a specialized piece of software (interpreter, kernel).

Efficiency of the target code. This can be assessed by comparing the run-time performances of the "skeletized" application with the ones obtained with a carefully hand-crafted parallel version (using C+MPI for instance). When the skeleton implementation relies on a specific run-time support, the efficiency depends on the *overhead* introduced by this support.

Portability. Skeleton-based parallel programs have often been claimed to be more portable than their counterparts built upon low-level parallel constructs. This is because the porting effort, when targeting a new parallel architecture, is reduced to redefining the intermediate representation of a small set of skeletons instead of rewriting the whole program. In the context of embedded vision applications, this portability issue must also take into account the possibility to target architectures with *little or no OS-level support* [2], such as machines built from specialized or digital signal processors (DSPs).

Predictability of performances. For most of existing skeleton-based PPEs [3,6,4,23,21] this takes the form of analytical *cost models*, from which the average timing behavior can be predicted on the basis of application-specific parameters (such as the estimated duration of the sequential functions) and architecture-specific parameters (such as communication latency). In the context of *reactive* applications, one may wish to replace this *statistical* approach by a *deterministic* one, in which *strict* temporal bounds can be computed at compile-time.

[2] By OS-level support, we mean the facilities typically provided by multi-tasking, Unix-like, operating systems: multi-processing, inter-process communication and synchronization, virtual memory, etc.

Expressivity. By expressivity we mean the ability to implement an application expressed as an arbitrary combination of skeletons. In practice, experience has shown that the critical point here is whether the intermediate representation supports *nesting* or not, ie. the ability for a skeleton to take another skeleton as argument. Although it is still unclear whether realistic applications really need nesting (see [8]), its support has always been perceived has a challenge by skeletons' implementors.

In practice, the above issues are closely related and often in tension one with each other. Relying on run-time level mechanisms for implementing skeletons, for instance, increases expressivity – it makes the implementation of nesting a much more tractable problem in particular – but complicates performance prediction (who must rely, ultimately, on stochastic models). The amount and the complexity of the run-time support may also be a concern when porting to architectures with small memory and/or limited OS-level support. By contrast, implementation models relying on compile-time mechanisms for solving the mapping and scheduling problems generally require smaller (if not no) run-time support, make performance prediction easier, but may lack expressivity. In this view, the development of the successive versions of Skipper may be viewed as a search – constrained by the operational context sketched in Section 1 – for an acceptable trade-off between efficiency, predictability, portability and expressivity.

4 Static data-flow. Skipper-0

The first version of Skipper used an intermediate representation of skeletal programs as synchronous data-flow graphs (SDFG) [5]. This representation was obtained from the textual description of the application in Caml thanks to a front-end tool called Camlflow. An in-depth description of this tool (which is based upon *abstract interpretation*) can be found in [26]. With this approach, skeletons were viewed as means of denoting recurrent data-flow *graph patterns* and were encoded directly in Caml as higher-order functions. The mapping of the SDFG onto the target architecture was handled by a third party software called Syndex [17]. This involved finding a (static) distribution of the sequential functions associated with nodes on processors and a (also static) scheduling of communications on inter-processor channels. The SDFG/Syndex approach is illustrated in Fig. 5. The left window shows the data-flow graph of a simple application making use of the scm (split, compute then merge) skeleton along with the target architecture (four ring-interconnected processors) on which it must be implemented. The right window illustrates the mapping of operations onto processors computed by Syndex, ie. the distribution of operations onto processors (one per column) and the scheduling of operations (oval boxes) and communications (diagonal

lines) on each processor. From this mapping, SYNDEX could finally generates parallel C code for the target architecture. This code took the form of a set of processor-independent programs (*m4* macro-code, one per processor), in which a `main` function contained direct calls to the sequential functions attached to the scheduled operations, interleaved with the communication instructions for exchanging data between processors. The macro-code was built from a small kernel of processor-independent primitives, which were finally inlined in C (or assembler) to get the final code.

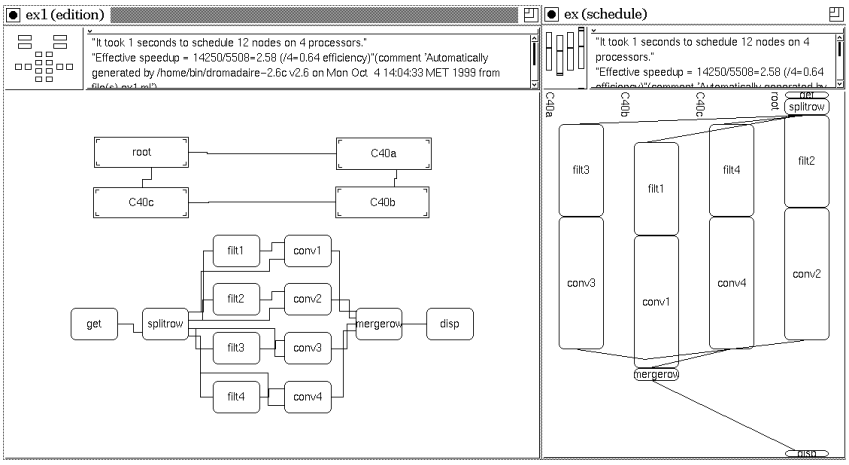


Fig. 5. SDFG representation of a skeletal program as handled by the SYNDEX tool

4.1 Assessment

With SKIPPER-0, no run-time support was necessary and the generated executives were very efficient (with an almost zero overhead compared to hand-crafted implementation). Accurate performance predictions were obtained by using a two passes process: in a first pass, rough estimates of the durations of the sequential functions were given to SYNDEX, which generated a first, sub-optimal, parallel program but with automatic profiling instructions inserted in it. This program could then be run on typical data to extract the real durations. These durations were used in turn to get the final program by means of a mapping and scheduling heuristic based upon minimization of the total latency. One could also use upper bounds for function durations in order to predict worst case behavior, in order to satisfy hard real time constraints for instance [3]. Portability was also good: because the output macro-code was built on a small set of kernel primitives, re-targeting an application on an architecture built from a new processor type only required (re)writing this set of kernel primitives. This proved to be a straightforward task for the platform

[3] To our knowledge, SKIPPER-0 is the only realization of a skeleton-based PPE capable of handling such hard real-time timing constraints.

we had to deal with [4].

The main problem with the SDFG/SYNDEX approach was expressivity. *Dynamic* skeletons, based on data *farming* in particular, could not directly be expressed. Data farming is useful for applying a function to a list of data items when the size of the list is unknown and/or the time to process one item can vary significantly [5]. In this case, a static allocation of items to processors is not always possible and would result, anyway, to a uneven work-load between processors (which in turn results in a poor efficiency). Data farming solves this problem by having a *master* process dynamically doling out items to a pool of *worker* processes and collecting results back, on a "first done, first served" basis. This model, however, makes it impossible to schedule the communications between the master and the workers at compile-time. It therefore cannot be described as a synchronous data-flow graph construct and could not be implemented within the SKIPPER-0 framework.

5 Template-based implementation. Skipper-1

In SKIPPER-1, the limitations of SKIPPER-0 are overcome by relying on *process networks* for the intermediate representation of skeletal programs and on *implementation templates* for skeletons. This approach is the most widely used for existing skeleton-based PPEs ([3,6,23,21]). Implementation templates are " *known parametric parallel process networks that efficiently implements a skeleton on a particular parallel target architecture at hand*" [13]. They generally take of the form of process graphs that can be parameterized in the parallelism degree (the number of the *worker* nodes for instance) and the sequential function(s) associated with each node. The intermediate representation of the application as a process network is then obtained by *instantiating* the skeleton templates [6]. The most often claimed advantage of template-based approaches lies in the fact that, being written once and for all for a given architecture, they can be carefully hand-crafted to make them both reliable and highly efficient.

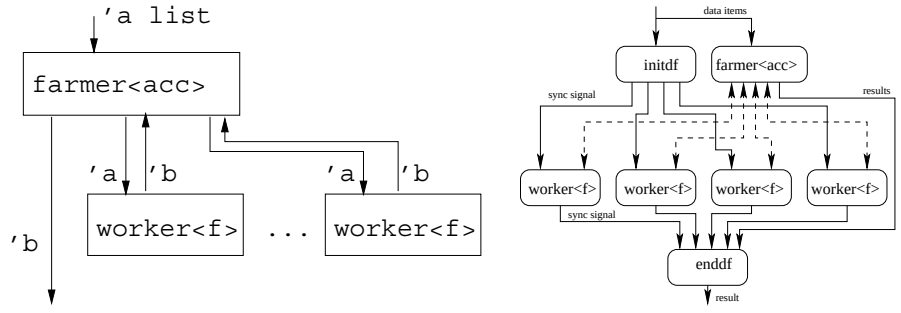
The CAMLFLOW front-end of SKIPPER was therefore modified to produce pro-

[4] The kernel definition for the Transputer processor was less than 300 lines of m4 code. Kernels have been written for several well-know DSPs and also for clusters of Unix machines running TCP/IP communication layers.

[5] This situation is frequent is reactive vision, where a varying number of *region of interest*, of varying size, often have to be processed in each frame.

[6] This instantiation is done on the basis of the provided application-specific sequential functions. It can also take into account some architectural parameters, to adjust the declared parallelism degree of the skeleton to the one actually offered by the architecture for instance.

cess networks out of CAML skeletal descriptions instead of data-flow graphs. For this, each skeleton was described (in CAML, again) as a *parametric process network*[7]. Fig. 6a gives a parametric process network (PPN) for the DF (Data Farming) skeleton[8]. This graph is parametric in the number of `worker` nodes, in the type of data items exchanged between nodes (denoted with type variables `'a` ... `'b`) and in the sequential functions run on the nodes `farmer` and `worker` (this "parameterization" being denoted with brackets).



(a) Simplified form (b) Actual form

Fig. 6. The parametric process network of the DF skeleton

The behavior of the `farmer` and `worker` processes was stored separately as a *parametric process template* (PPT). A PPT is a piece of sequential code whose behavior can be specialized by providing numeric parameters, data types and/or functional parameters[9].

The compilation path in SKIPPER-1 could then be decomposed into four steps: parametric process network generation, parametric process template instantiation, mapping/scheduling and code generation. It is illustrated on Fig. 7. The two last steps (mapping/scheduling and code generation) were still handled by the SYNDEX software. This may seem contradictory since, as stated in Section 4, SYNDEX can only handle synchronous data flow graphs and not process graphs. The solution adopted in SKIPPER-1 was in fact an hybrid one: process graphs were "viewed" by SYNDEX as data-flow graphs and mapped/scheduled as data-flow graphs. In particular, SYNDEX only scheduled (at compile-time) "static" communications (the ones that mark the start and the end of a farming skeleton for instance). The "dynamic" communications (the ones occurring between the master and the workers during the activity of a farming skeleton) were handled by ad-hoc processes "hidden" in the data-flow nodes. This technique – which amounts to tolerating some "critical sections" of dynamically scheduled code within a globally statically scheduled application – is detailed

[7] To facilitate cross-referencing, we use here the terms introduced in [27]. Conceptually, *parametric process networks* are *implementation templates*.

[8] This graph is a simplified one. The PPN actually used in SKIPPER-1 appears in Fig. 6b (see later).

[9] Specialization is carried out using macro substitution.

in [15]. It is illustrated in Fig. 6b, where "static" communications are denoted with plain lines and "dynamic" ones with dashed lines. Synchronization barriers were used to ensure that the dynamic communications did not interfere, at run time, with the static ones.

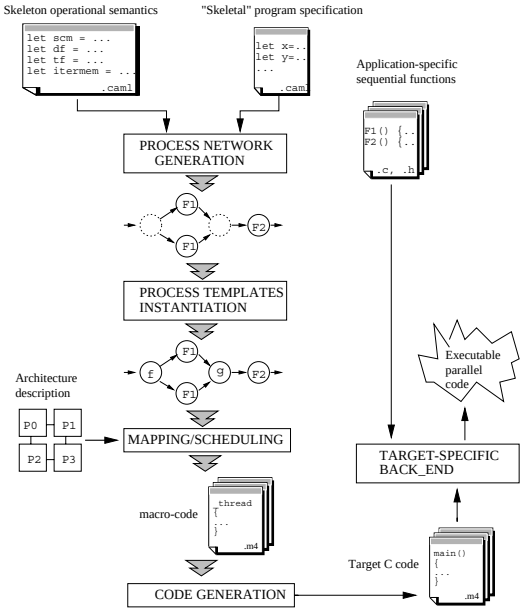


Fig. 7. Compilation path in the SKIPPER-1 parallel programming environment

5.1 Assessment

The SKIPPER-1 version was the first to be used for implementing realistic reactive vision applications, most noticeably those described in [16] (segmentation by connected component labeling), [25] (vehicle tracking) and [27] (road tracking). Thanks to the SYNDEX back-end, efficiency remained high (with an overhead never exceeding 25 % for the applications implemented). For applications making use only of "static" skeletons (such as SCM), this overhead was almost zero, as for SKIPPER-0. Predictability of performances relied on a set of analytical cost models [15] that provided an accuracy in the range of 10-20 %. But, unlike SKIPPER-0, strict timing bounds could not always be exhibited: this is clearly the price to pay for accepting dynamically scheduled skeletons such as DF. The main problem with SKIPPER-1 lied in the hybrid nature of the intermediate representation. Because dynamic communications were transparent to SYNDEX, the routing of these communications between distant processors had to be handled explicitly by some auxiliary processes (whereas it is done automatically by SYNDEX for static communications). It turned out that including the description of these auxiliary processes to the SYNDEX kernel, in the form of efficient and architecture-independent parametric process templates was a difficult task. To make the problem tractable, the SKIPPER-1 compilation process therefore made assumptions on the topology of

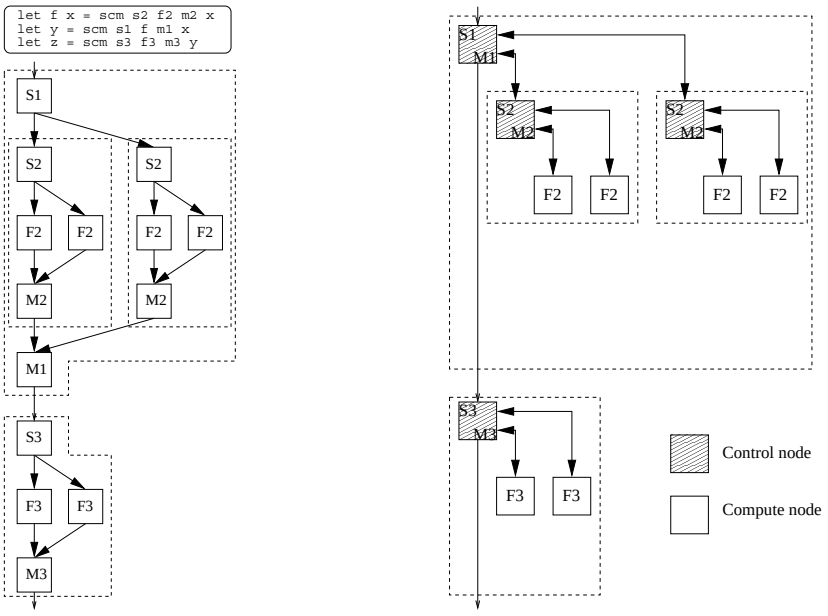
the target architecture (it had to be ring-interconnected). These assumptions, along with the increased size and complexity of the SYNDEX kernel, lowered the portability of the applications developed with SKIPPER-1 (compared to SKIPPER-0). Finally, the hybrid intermediate representation of SKIPPER-1 implicitly relied on a "flat" execution model and was definitely not suited for implementing nested skeletons.

6 Hierarchical task graphs. Skipper-2

The SKIPPER-2 version is based upon an homogeneous intermediate representation of programs as *hierarchical task graphs*. This design choice was made in order to overcome the difficulties raised by hybrid representations (such as the one used in SKIPPER-1) and to solve the problem of skeleton nesting in a systematic way. For this, and at the implementation level, all skeletons of the SKIPPER repertoire are viewed as specialized *instances* of a generic skeleton, called `tf-ii` [10] . The operational semantics of the `tf-ii` skeleton is basically the one of a task farming skeleton: a *master* process doles out tasks to a pool of *worker* (slave) processes, but here a task can be either a sequential function to be computed or another skeleton to be run. The intermediate representation takes the form of a tree of `tf-ii` skeletons. It is computed by a modified version of the CAMLFLOW front-end, which uses alternate definitions of the SCM, DF and TF skeletons as specialized calls to the `tf-ii` higher-order function. This step is illustrated in Fig. 8 where a program making using of three SCM skeletons (two of them nested) is turned into a tree of `tf-ii` descriptors. In this tree, nodes correspond to skeleton control processes and leafs to sequential functions (a detailed presentation of the SKIPPER-2 system can be found in [9] or in the forthcoming [10]).

Interpretation of the intermediate representation within SKIPPER-2 is done at run-time by a specialized program (the "kernel") running in SPMD mode on all processors (see Fig. 9). This kernel – written in C – provides dynamic support for three kind of services: concurrent execution of *master* and *worker* processes, inter-process communication (using a subset of MPI-conformant routines) and handling of shared resources such as the worker pool. Whenever a skeleton needs to be run, either as a "top-level" node (on the spine of the `tf-ii` tree) or as a nested instance, a new copy of the kernel is launched on the local processor. This copy acts as the *master* of the skeleton. It uses the free resources (idle processors) to allocates new workers. When all resources are busy, the execution of *worker* processes is sequentialized on the processor running the *master* process.

[10] For Task Farming, version II.



(a) Original program (b) Intermediate representation as a tree of `tf-ii`

Fig. 8. Intermediate representation of skeletal programs within Skipper-2

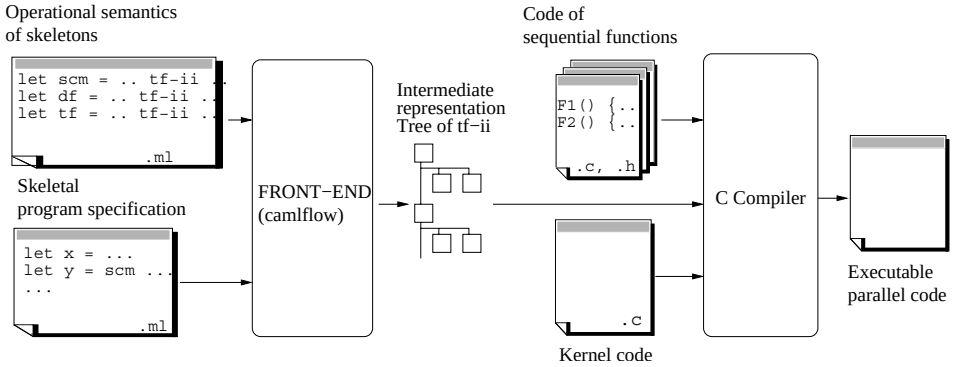


Fig. 9. Compilation path in the Skipper-2 parallel programming environment

6.1 Assessment

Skipper-2 is the first version to use a fully *dynamic* implementation mechanism for skeleton-based programs. This has several advantages. First, in terms of expressivity, since arbitrary nesting of skeletons is naturally supported. The introduction of new skeletons is also facilitated, since it only requires giving their translation in terms of `tf-ii`. Portability remains acceptable since porting applications to new architectures only requires the porting of the run-time kernel. This, in practice, turned to be a relatively straightforward task. The approach used in Skipper-2 also provides automatic load-balancing, since all mapping and scheduling decisions are taken at run-time, depending on the available physical resources. In the same vein, sequential emulation is obtained "for free" by just running the program on a single processor. As regards ef-

ficiency, preliminary experiments [9] on synthetic applications suggest that for applications exhibiting a sufficiently high compute/communication ratio the overhead of the kernel-based implementation (compared to hand-crafted C+MPI code) can be less 10 %, although this overhead can grow up to 50 % when the communications costs dominates [11] . But several problems have been identified in the SKIPPER-2 implementation which in practice have limited its utility in our context. First, predictability of performances is very low. It is very difficult, in particular, to exhibit even approximative cost models for a model in which processors can switch from a *master* to *worker* behavior depending only on actual input data (there's no "fixed" mapping for dynamic skeletons as in SKIPPER-1). Even the interpretation of execution profiles, generated by an instrumented version of the kernel, turned out to be far from trivial. This point raises a pragmatical problem within a programming methodology based upon experimental validation of solutions: here, one not only needs to obtain quickly a running prototype, but also to be able to understand why a given prototype exhibit poor run-time performances. By contrast, the profiling facilities offered by SKIPPER-1 (and detailed in [27]) were much easier to exploit. Second, shared resources are handled in a centralized manner in SKIPPER-2 (each worker allocation requires a couple of communication to a particular processor, in particular). This centralization can become a bottleneck when the size of the intermediate representation increases. Finally, it turned out that the resource allocation strategy used in SKIPPER-2 only performed well on architectures made of processors supporting *multi-processing*. If not, some processors may end up running only one *master* process, with a small load factor, leading to a poor global efficiency [12] . This problem practically limits the applicability of the SKIPPER-2 system in our context and, in a rather unexpected way, makes it more suitable for massively parallel, Beowulf-like clusters than for embedded target architectures.

7 Dynamic Data-flow. Skipper-D

The implementation of SKIPPER-D started in 2000 and was inspired by results obtained by M. Danelutto on the Macro Data-Flow (MDF) execution model for skeletons [12]. This model is very similar to the one used in SKIPPER-0: skeleton-based parallel programs are compiled down to data-flow graphs, in which nodes correspond to sequential functions ("macro-instructions") and arcs to data dependencies between these functions. But, unlike SKIPPER-0,

[11] In [9], this is explained by the fact that the kernel intrinsically performs more communications than raw MPI, for exchanging data between inner and outer *masters* in particular.

[12] If the multi-processing case, the processor can be shared between *master* and *worker* processes.

Danelutto proposes a *dynamic* interpretation mechanism for executing these graphs. This mechanism relies on a set of distributed data-flow interpreters, running in SPMD mode on all processors of the target architecture. SKIPPER-D extends the MDF execution model proposed by Danelutto in order to implement arbitrary nested data or task farm skeletons. For this, the SKIPPER-D runtime relies on the *tagged-token* data-flow interpretation technique [1,2]. This technique basically allows many concurrent activations of a single sequential node to overlap in time; it associates a unique *tag* with each activation and each data token also carries a tag that specifies the particular activation to which it belongs. Skeletons involving run-time bounded iterations and/or recursion, and nested in an arbitrary way, can then be represented as cyclic data-flow graphs. This is illustrated in Fig. 10 with the formulation as a tagged-token MDF graph of a program involving two nested DF skeletons (in this figure, tags are denoted as superscripts). The MDF graph uses a pair of special nodes called `iter` and `endf`. `Iter` accepts a list of data items and generates distinct result tokens, each carrying one data item and a distinct tag. These tokens trigger distinct firings of the subsequent nodes. The tokens resulting from these firings are collected by the `endf` and accumulated using the `acc` sequential function. A more detailed account of this mechanism can be found in [24].

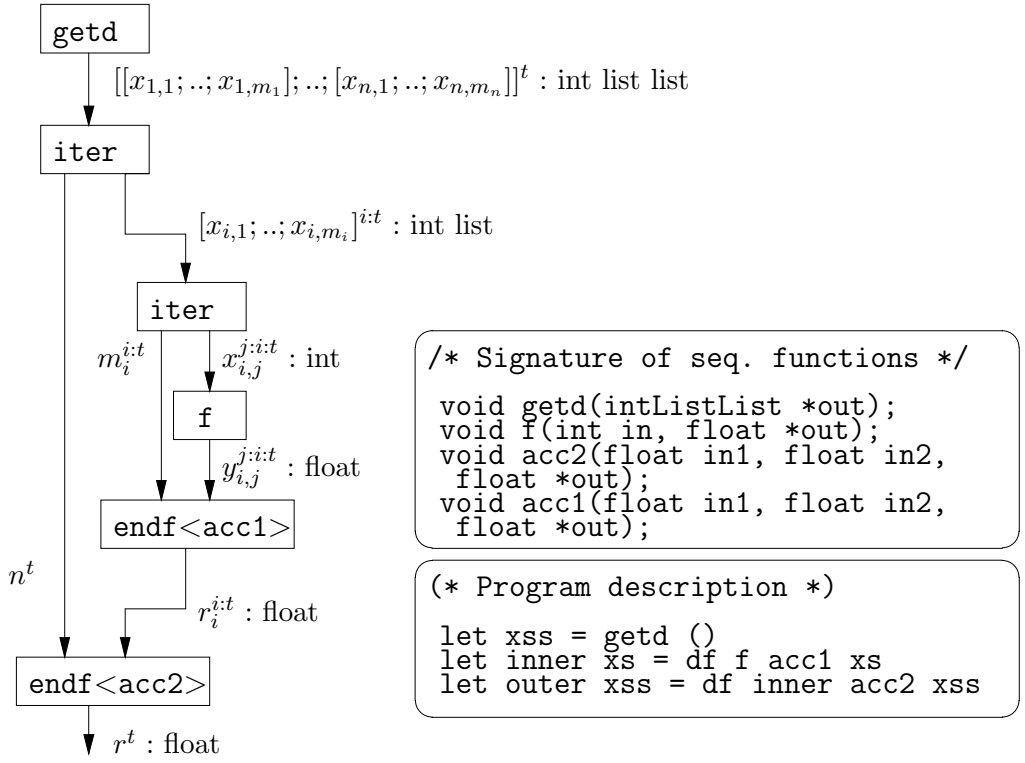


Fig. 10. Nested farm skeletons under the tagged-token MDF model

The SKIPPER-D programming environment relies on two sub-systems: a run-time system (RTS), implementing a (centralized) tagged-token data-flow in-

terpreter and a compile-time system (CTS), producing the MDF graph for this interpreter from a high-level skeletal program specification.

The **run-time system** of SKIPPER-D is sketched in Fig. 11. Like Danelutto's system, it relies on an SPMD approach: all the processors (nodes) of the target architecture run the same program, which is the result of the compilation of the user code (C sequential functions) and the interpreter code. The interpreter itself involves several *threads* of execution: a `dispatch` thread, which fetches macro-instructions (sequential functions to be computed) from a pool of fireable instructions and sends them to the `worker` threads, a `collect` thread, which receives results from the `worker` threads and updates the instruction pool accordingly and several[13] `worker` threads for computing sequential functions. The `dispatch` thread fetches idle workers from a centralized pool, in which all worker threads register at initialization and which is subsequently updated by the `update` thread upon reception of results.

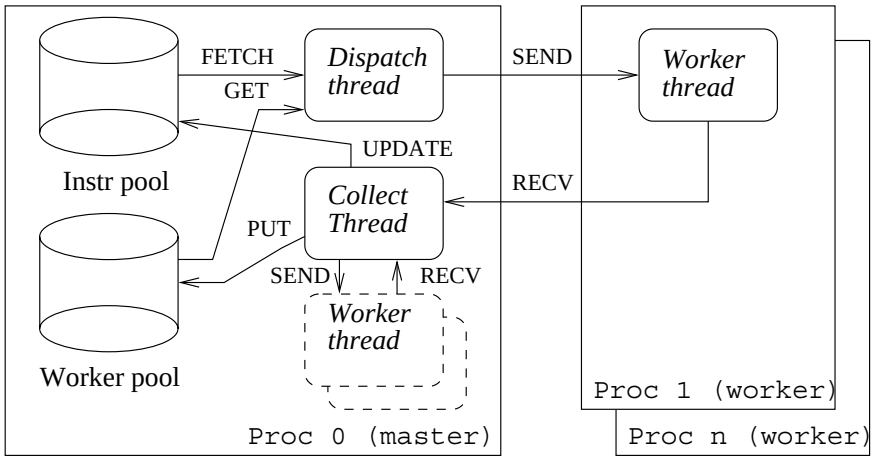


Fig. 11. The run-time system of SKIPPER-D.

The **compile-time system** is sketched in Fig. 12. It produces the application-specific data needed to customize the run-time interpreter, ie. the MDF representation of the program used to build the initial instruction pool and the code of the sequential C functions to be integrated with the custom run-time interpreter. The MDF graph is generated by the CAMLFLOW tool. This offers a way, like in previous versions of SKIPPER to describe skeletons entirely in CAML as higher-order functions.

7.1 Assessment

The main contribution of SKIPPER-D is to provide an all-encompassing intermediate representation for all skeletons. This representation allows arbitrary

[13] At least one per processor.

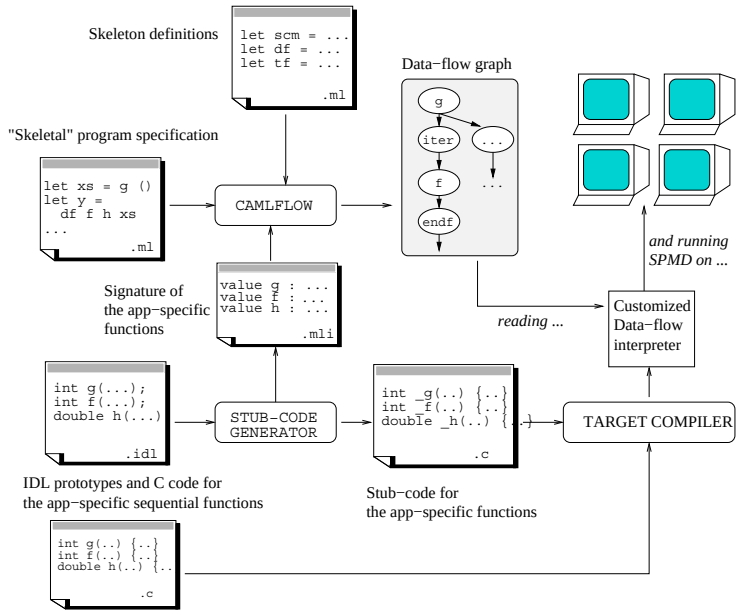


Fig. 12. The run-time system of Skipper-D.

combination (including nesting) of skeletons, thanks to the tagged-token interpretation mechanism. Skipper-D therefore definitely solves the expressivity problem, at least for our repertoire of skeletons. Experimental results, obtained with a prototype run-time system (written in Objective Caml) on a cluster of workstations are reported in [24]. They show that, at least for "synthetic" applications, performances can get very close to hand-written C+MPI code (less than 10 % overhead). Moreover, it turns out that, at least for coarse and medium-grained computation schemes, the mechanism used for handling nesting does not entail a significant performance penalty. Together with those reported by Danelutto in [13], these results confirm the merits of dynamic MDF execution models with respect to template-based ones. Skipper-D run-time performances could be further improved by integrating some optimization techniques described in [13]. These techniques include a more sophisticated management strategy of the instruction pool (based on high/low water marks), local caching of data on worker nodes and, most noticeably, a distributed interpreter implementation. The current Skipper-D implementation relies on a *centralized* data-flow interpreter and a rudimentary scheduling strategy for fireable instructions and is likely not to provide comparable performances in case of very irregular fine-grained computations. Predictability of performances is clearly harder to obtain than with template-based implementation systems but does not seem an intractable problem (like in Skipper-2). The interpretation of profiling results is also easier than with Skipper-2, especially if sophisticated visualization tools such as `jumpshot` [28] are provided. The portability of the Skipper-D runtime system on architectures made of specialized or digital signal processors is currently limited by the fact that it is written in Objective Caml and uses *bytecode* threads. But the runtime

could easily be rewritten in C for these systems [14]. In this case, threads can be emulated using hardware context switching mechanisms (as evidenced by the implementation of the SYNDEX kernel for DSPs [18]).

8 Comparative assessment

Table 1 summarizes our assessment of the successive versions of SKIPPER. In this table, we have tried to rate each version in terms of the five criteria explicited in Section 3: balance between compile-time and run-time system (*Rts/Cts*), efficiency (*Eff*), expressivity (*Expr*) , portability (*Port*) and predictability (*Pred*). For this we use a relative "score" between 1 and 4. For the *Rts/Cts* criteria, 1 means a fully static system – for which all decisions regarding mapping and scheduling of functions are taken at compile-time – and 4 a fully dynamic system for which all these decisions are taken at run-time. For the other criteria 1 means "poor" and 4 "excellent". The second column recalls the underlying intermediate representation (IR): Synchronous Data Flow Graphs, Parametric Process Networks, Hierarchical Task Graphs and Dynamic Data Flow Graphs.

	IR	Rts/Cts	Eff	Expr	Port	Pred
SKIPPER-0	SDFG	1	4	1	4	4
SKIPPER-1	PPN	2	3	2	1	3
SKIPPER-2	HTG	4	2	4	3	1
SKIPPER-D	DDFG	3	3	4	4	2

Table 1
Comparative assessement of SKIPPER versions

The evolution from SKIPPER-0 to SKIPPER-D can be viewed as a progressive shift – evidenced by the growing part of the run-time system in the implementation – from static approaches, offering excellent performances and predictability at the price of a limited expressivity, to more dynamic approaches, trading off efficiency and/or predictability in favor of expressivity

Fully static approaches, like in SKIPPER-0, are attractive in our context of embedded reactive applications because they minimize the resources needed to implement the algorithm and allow strict real-time bounds to be computed. But within a programming methodology dedicated to the fast prototyping of solutions – and mainly intended to algorithmicians, not parallel programming

[14] The current implementation is less than 500 lines of OBJECTIVE CAML code. We think that a re-implementation in C would be in the range of 1000-2000 loc, perfectly suited for small memory-print processors.

specialists – these approaches were finally found too restrictive. For instance, it is often possible to reformulate an existing vision algorithm – defined in terms of dynamic allocated data structures as lists or trees – so that it only uses fixed-size arrays and can be parallelized using a static data partition scheme; but we found that it is not reasonable, even desirable, to do this reformulation at the prototyping level, when being able to quickly test various algorithmic and/or parallel implementation schemes turned out to be more important than obtaining optimal performances. Moreover, some algorithms are intrinsically not amenable to a static implementation because the size of the input data and/or the duration of the sequential functions cannot be reliably estimated at compile time.

On the other hand, the conclusions given in Section 6.1 show that approaches relying on a fully dynamic run-time system, like SKIPPER-2, may raise efficiency and predictability or observability problems that conflicts with our prototyping goals and/or target platforms (although these approaches might prove useful in other application domains).

In this light, we believe that the SKIPPER-D approach offers the best trade-off between the conflicting abovementioned criteria. The data-flow interpretation mechanism is "mostly dynamic" [15] but its run-time behavior can be more easily modelized and performances do not suffer from hardly understandable performances drops due to unpredictable process allocation [16]. Moreover, we are investigating the possibility of developing transformational rules to derive automatically a static formulation of an algorithm (using a *synchronous* data-flow execution model) from a dynamic one (based upon a *tagged-token* execution model). Our ultimate goal, motivated by our experience and needs in reactive vision applications, is to be able to specify, with the *same* skeletal formalism both "hard" (time-critical) parallel applications (built from static skeletons such as SCM) and "softer" applications (built from dynamic skeletons such as DF) which can tolerate the run-time unpredictability implied by interpreter-based implementation techniques. Recent work on *graph factorization* techniques [14] has provided some insights on how to do this in the context of compile-time bounded iterations. We are currently working to extend this scheme to generic data and task farming skeletons (the fundamental issue being: what constraints do we have to put on the tagged-token data-flow graph formulation of an algorithm — that can always be interpreted dynamically — to make it amenable to static implementation).

[15] Scheduling is done at run-time but mapping of threads to processors is done at compile-time.
[16] As for the *master* processes in SKIPPER-2.

9 Conclusion

This paper has focused on implementation issues, investigating in particular the relative merits and flaws of *static* and *dynamic* approaches for skeletons. One of its conclusion is that a macro data-flow representation of skeleton-based parallel programs is probably the best choice, because it can be associated with a wide spectrum of operational semantics (from purely static synchronous to dynamic tagged-token). This conclusion is similar to the one drawn by Najjar *et al* in [22] who underline the "universality" of the data-flow model by exhibiting potential application domains both in the "software" domain (parallel programming on clusters of workstations for instance) and in the "hardware" domain (design of application-specific circuits for instance).

Beside these implementation issues, the Skipper project has also provided some useful insights on the applicability of skeleton-based parallel programming techniques. These conclusions are supported by realistic case studies, carried out with the help of full-fledged parallel programming environments, by people who were not parallel programming specialists at the first place. First, the "off-the-shelf" style provided by the skeleton approach effectively provides dramatic savings in development effort. These savings make it possible to adopt a truly experimental approach in the design and implementation of applications, a key property in our context. The price to pay is a decrease in performances (compared to hand-crafted parallel code) but, for most of the realizations presented here this can be kept reasonable and was viewed as acceptable, anyway, with regard with the above mentioned benefits. Second, within a given application domain, such as reactive embedded vision, skeletons may be viewed as a very effective way to *encapsulate* and reuse the expertise gained by skilled parallel programmers. This pragmatically solves the classical "completeness" problem often associated with skeleton-based parallel programming methodologies – namely the fact that, in theory, nothing can guarantee that a given set of skeletons will be sufficient to express every parallel algorithm: in our case, the definition of the skeleton basis was made in a *bottom-up* manner starting from an identifiable corpus of applications and/or expert knowledge and was explicitly targeted towards low to mid-level vision algorithms. Finally, it could be objected to the explicit, "menu-driven" approach proposed by Skipper that it requires a minimum understanding of the skeleton operational semantics to be used and therefore that it cannot be used as fully automatic parallelizing tool. Our answer, motivated by our experience in developing complex vision applications with algorithmicians, is that skeletons actually provides an effective *common ground for sharing expertise* between image processing and parallel programming specialists: the former no longer have to deal with implementation details and the latter can treat application-specific functions as black boxes.

References

[1] Arvind and K. P. Gostelow. The U-interpreter. *IEEE Computer*, 15(2):42–49, Feb. 1982.

[2] Arvind and R. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Transactions on Computers*, 39(3):300–318, Mar. 1990.

[3] 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.

[4] B. Bacci, M. Danelutto, S. Pelagatti and M. Vanneschi. SkIE: an heterogeneous environment for HPC applications. *Parallel Computing*, 25:1827–1852, Dec 1999.

[5] A. Benveniste, P. Le Guernic, and C. Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming*, 16(2):103–149, Sep 1991.

[6] S. Ciarpaglini, M. Danelutto, L. Folchi, C. Manconi, and S. Pelagatti. anacleto: a template-based p3l compiler. In *Proceedings of the Seventh Parallel Computing Workshop (PCW '97)*, Australian National University, Canberra, August 1997.

[7] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1989.

[8] M. Cole. Algorithmic skeletons. In G. J. Michaelson and K. Hammond, editors, *Research Directions in Parallel Functional Programming*. Springer Verlag, 1999.

[9] R. Coudarcher, J. Sérot and J.P. Dérutin. Implementation of a Skeleton-based Parallel Programming Environment Supporting Arbitrary Nesting. *6th International Workshop on High-Level Parallel Programming Models and Supportive Environments*, Apr. 2001, San Francisco. Volume 2026 of LNCS, pp 71–85, Springer.

[10] R. Coudarcher. *Composition de squelettes algorithmiques : application au prototypage rapide d'applications de vision*. PhD thesis, Université Blaise Pascal Clermont-Ferrand (France), 2002. To appear.

[11] M. Danelutto, R. DiCosmo, X. Leroy, and S. Pelagatti. Parallel functional programming with skeletons: the OCamlP3L experiment. In *Proceedings ACM workshop on ML and its applications*. Cornell University, 1998.

[12] M. Danelutto. Dynamic run time support for skeletons. In *Proceedings of the ParCo99 Conference*, Delft, The Netherlands, August 1999.

[13] M. Danelutto. Efficient run-time support for skeletons on workstation clusters. *Parallel Processing Letters*, 11(1):41-56, Feb. 2001.

[14] A. Dias, C. Lavarenne, M. Akil, and Y. Sorel. Optimized implementation of real-time image processing algorithms on field programmable gate arrays. In *ICSP'98 Fourth International Conference on Signal Processing*, Beijing, China, Oct 1998.

[15] D. Ginhac. *Prototypage rapide d'applications parallèles de vision artificielle par squelettes fonctionnels*. PhD thesis, Université Blaise Pascal Clermont-Ferrand (France), 1999.

[16] D. Ginhac, J. Sérot, and J. Dérutin. Fast prototyping of image processing applications using functional skeletons on a MIMD-DM architecture. In *IAPR Workshop on Machine Vision and Applications*, pp 468–471, Chiba, Japan, Nov 1998.

[17] T. Grandpierre, C. Lavarenne, and Y. Sorel. Optimized rapid prototyping for real time embedded heterogeneous multiprocessors. In *7th Intl Workshop on Hardware/Software Co-Design*, Rome, May 1999.

[18] C. Lavarenne and Y. Sorel. Modèle d'exécutif distribué temps-réel pour SynDEx *INRIA Research Report*, RR-3476, Aug. 1998.

[19] P. Legrand, R. Canals, and J.P. Dérutin. Edge and region segmentation processes on the parallel vision machine Transvision. In *Computer Architecture for Machine Perception*, pages 410–420, New-Orleans, USA, Dec 1993.

[20] G.J. Michaelson and N.R. Scaife. Prototyping a parallel vision system in standard ML. *Journal of Functional Programming*, 5(3):345–382, 1995.

[21] G.J. Michaelson, N. Scaife, P. Bristow and P. King. Nested algorithmic skeletons from higher order functions. *Parallel Algorithms and Applications*, 16:181-206, Aug 2001.

[22] W.A. Najjar, E.A. Lee, and G.R Gao. Advances in the dataflow computational model. *Parallel Computing*, (25):1907–1929, 1999.

[23] N. Scaife, P. Bristow, G. Michaelson and P. King. Engineering a parallel compiler for SML. Proc. *10th International Workshop on Implementation of Functional Languages*, Sep 1998, pp 213-226;

[24] J. Sérot. Tagged-token data-flow for skeletons. *Parallel Processing Letters*, 11(4), Dec. 2001.

[25] J. Sérot, D. Ginhac, and J. Dérutin. Skipper: a skeleton-based parallel programming environment for real-time image processing applications. In *5th International Conference on Parallel Computing Technologies*, volume 1662 of *LNCS*, pp 296–305. Springer, 6–10 Sept. 1999.

[26] J. Sérot. CamlFlow: a Caml to data-flow graph translator. In S. Gilmore ed., *Trends in Functional Programming, Vol 2*, Intellect, 2001.

[27] J. Sérot, D. Ginhac, R. Chapuis, and J. Dérutin. Fast prototyping of parallel vision applications using functional skeletons. *Journal of Machine Vision and Applications*, 12(6):271-290, Jun. 2001.

[28] O. Zaki, E. Lusk, W. Gropp, and D. Swider. Toward scalable performance visualization with Jumpshot. *High Performance Computing Applications*, 13(2), 1999.