Machine Vision and Applications (2001) 12: 271–290

Machine Vision and Applications
© Springer-Verlag 2001

Fast prototyping of parallel-vision applications using functional skeletons

Jocelyn Sérot, Dominique Ginhac, Roland Chapuis, Jean-Pierre Dérutin

Laboratoire des Sciences et Matériaux pour l'Electronique, et d'Automatique, Université Blaise Pascal de Clermont Ferrand, UMR 6602 CNRS, 63177 Aubière Cedex France; e-mail: Jocelyn.Serot@lasmea.univ-bpclermont.fr

Received: 22 July 1999 / Accepted: 9 November 2000

Abstract. We present a design methodology for real-time vision applications aiming at significantly reducing the design-implement-validate cycle time on dedicated parallel platforms. This methodology is based upon the concept of algorithmic skeletons, i.e., higher order program constructs encapsulating recurring forms of parallel computations and hiding their low-level implementation details. Parallel programs are built by simply selecting and composing instances of skeletons chosen in a predefined basis. A complete parallel programming environment was built to support the presented methodology. It comprises a library of vision-specific skeletons and a chain of tools capable of turning an architecture-independent skeletal specification of an application into an optimized, deadlock-free distributive executive for a wide range of parallel platforms. This skeleton basis was defined after a careful analysis of a large corpus of existing parallel vision applications. The source program is a purely functional specification of the algorithm in which the structure of a parallel application is expressed only as combination of a limited number of skeletons. This specification is compiled down to a parametric process graph, which is subsequently mapped onto the actual physical topology using a third-party CAD software. It can also be executed on any sequential platform to check the correctness of the parallel algorithm. The applicability of the proposed methodology and associated tools has been demonstrated by parallelizing several realistic real-time vision applications both on a multi-processor platform and a network of workstations. It is here illustrated with a complete road-tracking algorithm based upon white-line detection. This experiment showed a dramatic reduction in development times (hence the term fast prototyping), while keeping performances on par with those obtained with the handcrafted parallel version.

Key words: Parallelism – Computer vision – Fast prototyping – Skeleton – Functional programming – CAML – Road following

Correspondence to: J. Sérot

1 Introduction

It is commonplace to say that parallel programming is difficult. The tasks of partitioning a problem into parallel activities (processes or tasks), mapping and scheduling them onto processors, and keeping them synchronized (using message passing or shared memory) are all non-trivial and place severe strains on programmers. But programming parallel machines dedicated to real-time vision is even more difficult. This is mainly due to the fact that these machines, in order to cope with the stringent constraints made by *on-the-fly* processing of digital video streams – typically 25 512×512 images per second – while meeting some operational constraints such as volume or power consumption are generally built from specific, heterogeneous pieces of hardware – such as digital-signal processors (DSPs) – with very small or no OS-level support. They therefore lack high-level parallel programming models and environments which are available on more traditional stock hardware [38]. The programmer then has to explicitly take into account all aspects of parallelism, including task partitioning and mapping, data distribution, communication scheduling, load-balancing, etc. Such a detailed control of low-level machine resources might be necessary when fine-tuning an application to achieve the highest performances, but is clearly not desirable at the *prototyping* level, when being able to quickly test various algorithmic and/or parallel implementation schemes is more important. Moreover, it is well known that programs explicitly built upon low-level parallel-programming constructs frequently suffer from deadlocks, exhibit non-reproducible errors, and are therefore difficult to test. A consequence of this is that, with the sparse debugging facilities generally available on dedicated platforms, it is often very difficult for the programmer to distinguish *algorithmic* bugs (an intrinsic design error) from *implementation* bugs (a communication buffer overflow for example). As a result, implementing a real-time vision application on a dedicated parallel platform remains a tedious and delicate task, leading to long development cycles (in the range of several programmer-months for the kind of applications presented in Sect. 4). Yet, *short development cycles* are essential at the prototyping level. This is specially true for many time-critical vision applications, for which the need to evaluate the *dynamic* properties of

the algorithm under realistic operating conditions – at least a dozen images per second for the application presented in Sect. 4 – effectively rules out any prototyping phase solely based upon off-line, sequential simulation on stock hardware. Hence, the need for a programming methodology allowing *rapid prototyping* of real-time vision applications on dedicated parallel architectures.

This paper proposes such a methodology, based upon the concept of *parallel skeleton*. Skeletons [2, 13, 39] are high-level programming constructs *encapsulating* certain common forms of parallel computations to make them readily available for the application programmer. Since their introduction by Cole in [13], skeletons have motivated a significant body of work, both on theoretical aspects and implementation issues. However, there are relatively few experiences with the approach applied in practice to realistic, large-scale problems. This paper tries to fill this gap by demonstrating the utility of a skeleton-based parallel-programming methodology in the domain of real-time vision applications.

It is organized as follows: Section 2 is a general presentation of the so-called skeleton-based parallel-programming methodology and of its application to real-time image-processing problems. Section 3 describes a complete parallel-programming environment (SKIPPER) built upon this approach and covering all the stages of program development, from architecture-independent specification to target code generation. The effectiveness of the proposed methodology is assessed in Sect. 4, by means of a realistic case study. Finally, lessons learnt while conducting this work and further work to be done are summarized in Sect. 5.

2 Skeleton-based parallel programming

Almost every paper dealing with skeleton-based parallel programming has its own definition of skeletons. One of the latest is given by Cole in [26]:

> "As with many good ideas, the underpinning observation is, in retrospect, "obvious": within the existing body of parallel algorithms a number of patterns recur frequently. These patterns are composed of computations and the interactions between them and can be conceptually abstracted away from the details of the activities they control. Such abstractions have come to be known as *algorithmic skeletons* or simply *skeletons*."

Another definition, more inspired by implementation issues, is given by Bratvold in [6]:

> "It has been observed [...] that parallel programs written in explicitly parallel languages consist of two different kinds of code, often tightly interwoven: pieces of *task-specific* code implementing the individual steps of the algorithm, and code for structuring the program into *patterns of computation and communication* for parallel execution. It is typically only in the latter kind of code that there is a need for describing low-level resource allocation and for dealing with those aspects of parallel programming known to pose the greatest problems. The number of commonly used patterns is observed to be quite small [...] but the code for their efficient implementation varies greatly between target machines [...]. An appealing approach is to implement efficiently a fixed repertoire of useful patterns, which can be subsequently scaled and instantiated with task-specific code during compilation."

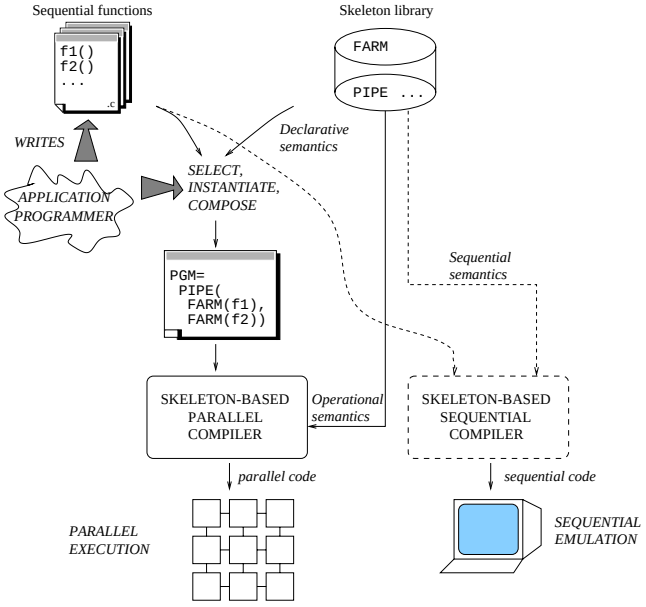


Fig. 1. Overview of a skeleton-based parallel-programming methodology

Thus, the basic idea of skeleton-based parallel-programming methodologies is to provide the programmer with a set of high-level *templates* (the skeletons) that abstract common patterns of parallel computation in a parametric way. Common examples of skeletons are *process farms*, *pipelines*, and *divide-and-conquer trees*. The repertoire of skeletons acts as a sort of "parallel toolbox" from which parallel programs can be built with a minimal concern for low-level details. The overall methodology is illustrated in Fig. 1, with a very small program example built of two sequential functions and two skeletons (a pipeline and a process farm[1]).

This figure shows the different components from which a skeleton-based parallel program is built (basically a skeleton library, a skeleton-based parallel compiler and an optional sequential compiler) and the role of the application programmer in this framework (he writes the application-specific sequential functions and uses skeletons from the provided library to describe the parallel structure of its application). This calls for the following remarks.

First, the work of the application programmer is limited to the selection, instantiation, and composition of skeletons taken from the provided library. This has to be contrasted with the more inventive, but much more tedious and error-prone, task of writing parallel programs from scratch using low-level constructs. The programmer is then spared a

[1] The semantics of these skeletons will be detailed in Sect. 2.1. For now, it suffices to know that PIPE performs in parallel several *stages* of computations – each stage computing a function f_i over all the data items produced by the previous stage – and that FARM applies in parallel its function argument to all the data items appearing at its input.

great deal of programming effort and can try many more implementation alternatives. Skeletons therefore provide a way to achieve **rapid prototyping** of applications. Moreover, since the only parallelism in programs arises from the use of skeletons (all other functions being executed sequentially), the parallel behavior of a program will be entirely given by the behavior of its constituent skeletons thus offering **reliability** (program will no longer crash because of a bug in a "once-again-reinvented" farm process implementation for instance) and **predictability of performances** (the restrictive form of communication patterns generally embodied by skeletons often makes it possible to provide accurate *cost models* in particular). Finally, since skeleton parameters are ordinary sequential functions[2] (written in C in our case), skeletons favor **reusability** of sequential code.

Second, each skeleton comes with three semantics:

- a *declarative semantics*, which gives its "meaning" to the application programmer in a target-independent manner. For instance, using the following notation for list: `[x1;...;xn]`, the declarative semantics of the FARM skeleton could be stated with the equation

  ```
  FARM(f,[x1;...;xn])=[f(x1);...;f(xn)]
  ```

- an *operational semantics* which describes how this skeleton is implemented on a given parallel platform. This definition can be given, as explained in Sect. 3.2, as a *parametric process network*;
- an (optional) *sequential* semantics, conferring a purely sequential interpretation to skeletal programs. This will be used to run them on traditional, sequential platforms such as workstations, as explained in Sect. 3.4.

Ultimately, there will be as many operational semantics as there are parallel target platforms[3], but they will all share the same declarative semantics. This provides a certain degree of **portability** to skeleton-based parallel programs[4]. It is, of course, the skeleton implementor's responsibility to ensure the *compatibility* of these semantics, i.e.,to guarantee that, apart from obvious differences in terms of performance, the corresponding definitions will give the *same* results when applied to the same input sets[5]

Third, an objection to such a methodology could be that it only shifts the burden from the application programmer to the skeleton implementor(s). Implementing a skeleton on a given platform may indeed be a non-trivial task, requiring detailed knowledge about low-level machine resources. But this is only done once and it is therefore reasonable to devote time and energy to it. Moreover, this implementation can be carefully handcrafted to make it both reliable and highly efficient (by taking advantage of architecture-specific features, for instance). Skeletons therefore provide a radical solution to the classical *ease-of-programming* vs *efficiency* problem.

This presentation of the methodology is obviously very general. Many issues that it raises will be discussed in Sect. 3 when describing a suite of tools that practically implements it. But beforehand, we need to see why and how it can be refined to be applied to our specific application domain, namely real-time image processing.

2.1 Parallel skeletons for image processing

Most work on skeletons have proposed "general-purpose" skeletons. It is clear, however, that many application domains, dealing with specific data and control structures, can benefit from a skeleton-based approach to parallel programming, since these structures can be captured and abstracted as skeletons. Working with "domain-specific" skeletons can also help to solve an often mentioned problem with skeleton-based approaches, namely the fact that, in theory, nothing can guarantee that a given set of skeletons will be sufficient to express[6] every parallel algorithm. The profusion of propositions for skeleton basis (evidenced in surveys like [9] for example) clearly shows the difficulties encountered in defining such a basis. Within a fixed application domain, the problem becomes much more tractable, since the definition of the skeleton basis can now be made in a true *bottom-up* manner, starting from an identifiable corpus of applications and/or expert knowledge, rather than trying to identify *a priori* a – still hypothetical – fully general-purpose skeleton basis. In this case, skeletons provide a very effective way of *encapsulating* the expertise gradually gained by parallel programmers in the domain and making it readily available for the rapid prototyping of subsequent applications.

In our case, defining an adequate skeleton basis involved first a restriction of the application domain and second solving a "trade-off" problem:

Restricting the application domain. Given the very wide range of image processing (IP) techniques – including low-level tasks (dealing with regular, iconic data, such as convolutions, filters or regular transforms), intermediate-level tasks (dealing with much more irregular data representations, such as edges or regions) and high-level tasks (manipulating abstract models for object recognition) – it seems almost as illusory to consider a complete skeleton basis for IP as a whole, as is the case for general-purpose programming. Our first decision, to make the problem tractable, was therefore to restrict our approach to the *low* and *intermediate* levels of processing. This deliberate limitation can be further justified by the following reasons. First, low- and mid-level processing form the first stages of any real vision system and therefore have the greatest potential for reuse. Second, they typically process large data sets and so make good candidates for parallelism. Last (but not least !), at the time the project started, we already had substantial experience in the

[2] We will deliberately exclude here the issue of skeleton *nesting*, i.e.,the possibility for a skeleton to take another skeleton as an argument, since this is a rather complex one and the version of SKIPPER described in this paper does not support it.

[3] Most of them will use a common intermediate description level, however, thanks to parallel implementation techniques presented in Sect. 3.2.1–3.2.3.

[4] Though portability of *performances* across very different architectures is a much more difficult problem.

[5] For certain skeletons, ensuring this compatibility may require putting additional constraints on the skeletons' arguments, as explained later in note 11.

[6] In mathematical terms, will form a *complete* basis

related class of algorithms, as well as in the parallel implementation of these algorithms on multi-processor targets for realistic, large-scale applications.

Solving the generality-versus-specificity problem. This is a classic problem when trying to define a skeleton basis, already pointed out in [35]. Generality here means a small number of very general, highly abstract skeletons, while specificity means a larger set of more specialized skeletons. The latter offers the best opportunities for efficiency – because implementations can be closely tailored to meet the requirements of the underlying communication patterns in particular –, but may require the frequent crafting of new problem-oriented skeletons when new applications are to be dealt with. This, in turn, rapidly results in an inflation of the skeleton basis, compromizing both its understandability and portability[7]. On the other hand, more general skeletons have a greater potential for *reuse*, but make efficient implementations harder to find (as well as performance prediction models). Furthermore, it has been observed that highly abstract skeletons (such as the `fold/map` homomorphism of the BMF [37] formalism) often require a non-trivial reformulation of the initial algorithm to accommodate it[8].

A major originality of our work was to solve the latter "trade-off" problem by adopting a true *bottom-up* approach, that is by defining the skeleton basis from a careful analysis of a large corpus of existing low-to-mid-level vision applications. By chance we had such a corpus at hand, in the form of thousands of lines of handcrafted parallel C code implementing various vision applications on several versions of an MIMD-DM[9] machine (the TRANSVISION real-time vision system [30], built on T800 and T9000 transputers). Examples of algorithms involved are low-level pre-processing (filtering, edge detection, etc.), spatio-temporal operators (Markov field analysis, etc.) [10], extraction of geometric primitive and/or perceptual groupings [29], and tracking of mobile objects in real scenes [3].

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).

The **SCM** skeleton is devoted to regular, "geometric" processing of iconic data, in which the input image is split into a fixed number of subimages, each subimage is processed independently, and the final result is obtained by merging the results computed on subimages. The subimages may overlap and/or not cover the entire input image. This skeleton is applicable whenever the number of subimages is fixed and the amount of work associated with each subimage is the same, resulting in a very even workload. Typical examples include convolutions, median-filtering and histogram computation.

The **DF** skeleton is a generic harness for so-called *process farms*. A process farm is a widely used construct for data parallelism, in which a *farmer* process has access to a pool of *worker* processes, each of which computes the same function. The farmer distributes items from an input list to workers and collects results back. The effect is to apply the function to every data item. We use a variant of this scheme in which the results collected by the farmer are accumulated using a specific function instead of being just added to an output list.

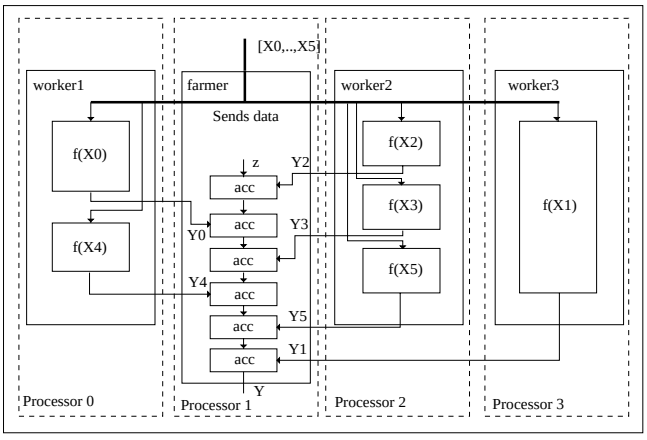


Fig. 2. An interpretation of the DF skeleton

The DF skeleton shows its utility when the application requires the processing of irregular data, for instance, an arbitrary list of windows of different sizes. In this case, a static allocation of tasks[10] to processors (like with the SCM skeleton) is not always possible and would result, anyway, to an uneven workload between processors (which, in turn, results in a poor efficiency). The DF skeleton handles this situation by having the farmer process directly doling out task allocation to worker processes. Typically, the farmer starts by sending a packet to each worker, waits for a result from a worker, and then immediately sends another packet to him. This is done until no packets are left and the workers are no longer processing data. Each worker simply waits for a packet, processes it, and returns the result to the farmer until it receives a *stop* condition from the farmer. This technique gives an inherent, primitive load balancing. It is only efficient, however, if there are more data items than processors.

Figure 2 gives a "static" view of the eminently dynamic behavior of a DF skeleton implemented on four processors The input list is `[X0;...;X5]`. Each worker computes a value `Yi=f(Xi)` and returns it to the farmer, where it is accumulated to the previous results (`z` being the initial accumulator value). Columns represent processors activity, with time flowing top-down and boxes corresponding to the sequential functions. Arrows denote communications. Note in particular how the order in which the results are accumulated in the farmer depends on the various processing times on the workers [11].

The **TF** skeleton may be viewed as a generalization of the DF one, in which the processing of one data item by a

[7] An implementation of each new skeleton must be given for *every* target architecture !

[8] This is admittedly true of the skeletons presented in this paper, but to a much lesser extent.

[9] Multiple Instruction Multiple Data, with Distributed Memory

[10] A task consisting here in the application of a function to a single data item of the list.

[11] For this reason, the operational and declarative semantics of the DF skeleton will only be compatible if the accumulation function `acc` is associative and commutative, since the former semantics may, in general, cause partial results `Yi` to be received and accumulated in any order, whereas this order is generally left unspecified in the latter.

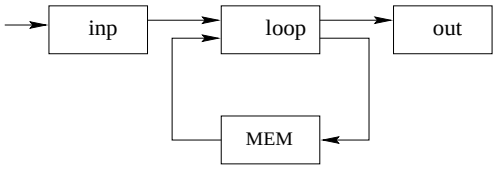


Fig. 3. The ITERMEM skeleton

worker may recursively generate new items to be processed. These data items are then returned to the farmer to be added to a queue from which tasks are doled out (hence the name *task farming*). A typical application of the TF skeleton is image segmentation using the classical recursive split-and-merge algorithm.

The **ITERMEM** skeleton does not, properly speaking, encapsulate parallel behavior, but is used whenever the *iterative* nature of the real-time vision algorithms, i.e.,the fact that they do not process single images but continuous *streams* of images[12] has to be made explicit. A typical situation is when computations on the n^{th} image depend on results computed on the $n-1^{th}$ (or $n-k^{th}$). The overall structure of the application can then be drawn as in Fig. 3, where `inp` is an input process, providing data from the input stream, `out` is an output process, displaying results typically, `loop` the central looping process, and `mem` a memory holding results from previous iterations.

Such "feedback" patterns are very common in tracking algorithms for instance, where a model of the system state is used to predict the position of the tracked objects in the next image. Another example is motion detection by frame-to-frame difference. The ITERMEM skeleton will always be the "top-level" skeleton, taking as parameter either a sequential function or a composition of other skeletons. The latter case is, incidentally, the only situation in which the current SKIPPER implementation supports *nested* skeletons.

3 A skeleton-based programming environment

The overall description of the skeleton-based parallel-programming methodology given in the previous section deliberately remained allusive on several "practical" points. For instance:

- how is the declarative semantics of a skeleton conveyed ? (Q1)
- how the are skeletal programs written (i.e.,how is skeleton instantiation and composition denoted)? (Q2)
- how is the operational semantics of skeletons described ? (Q3)
- what do we exactly mean by "sequential emulation" programs? Why is it useful? (Q4)

This section aims at clarifying these points by giving an in-depth description of the tools which form the SKIPPER parallel programming environment. The general architecture of this environment is depicted in Fig. 4. Questions Q1 and Q2 are answered in Sect. 3.1, dealing with the specification of skeletal programs. Question Q3 is answered in Sect. 3.2,

[12] In our case, this stream comes directly from a CCD camera, through a synchronizing frame grabber

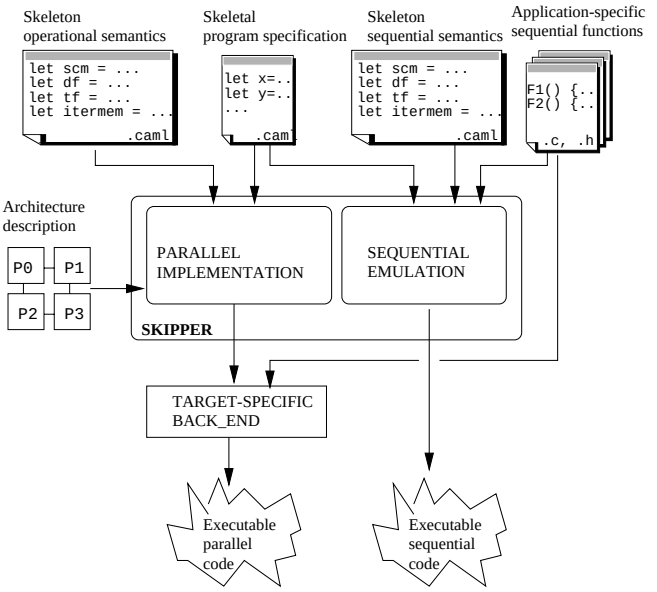


Fig. 4. Overview of the SKiPPER parallel programming environment

dealing with the parallel implementation of these programs. Questions Q4 is answered in Sect. 3.4. The SCM skeleton will be used throughout this section to exemplify the concepts and techniques. Corresponding information for the three other skeletons (DF, TF, ITERMEM) has been moved to Appendices 1 and 2, so as not to clutter the text with too much technical details.

3.1 Program specification

From a programming language perspective, an essential characteristic of skeletons is their *genericity*, i.e.,their ability to be instantiated with application-specific procedures having various types. Skeletons must therefore be *polymorphic* and *higher order* constructs. Polymorphism means the ability to encompass a range of data types using a single declaration. For example, the DF skeleton introduced in Sect. 2.1 will have to operate on list of data of *any* type `t` (not just `int`, `float`, ... but any structured C type, including user-defined types). Higher orderness refers to the ability for a function to accept other functions as parameters and/or to return functions as results. The PIPE skeleton introduced in Sect. 2 will, for instance, accept two functions `f` and `g` (from type `t1` to type `t2` and from type `t2` to type `t3`, respectively) and return a function `PIPE(f,g)` from type `t1` to type `t3`. The need for polymorphic, higher order constructs explains why most of the research on skeletons has been conducted using *functional languages*, within which polymorphic higher order constructs are naturally and elegantly expressed as *higher order functions*. Note that this does not necessarily mean that skeletal programs cannot be written using an imperative language like C or Fortran – as evidenced by the P3L project described in Sect. 5 – but that, in this case, the skeleton definitions require *meta-language* constructs, and hence more complicated compilation techniques. The C language, for instance, supports function pointers, but functions cannot

be created at runtime and cannot have polymorphic type[13]. Moreover, the support for data and function *polymorphism* in C is rather awkward, requiring either code duplication (using compile-time template expansion) or jeopardizing run-time safety (using `void *` types).

We therefore resorted to a functional programming language – CAML – for writing skeleton-based parallel programs. CAML [14] is a dialect of the well known and largely used[14] ML language [33].

To start here is, in CAML, the declarative semantics of the SCM skeleton introduced in Sect. 2.1.

```
let scm n split comp merge x
    = merge n (map comp (split n x))
```

In CAML, the phrase `let f <arg1>...<argn> = <body>` is used to introduce a function taking n arguments and whose definition is given in `<body>`. This definition itself here consists in the application of the `merge` argument to both the `n` argument (the number of subimages) and the result of the `map` application (in CAML, function application is denoted without parenthesis, so that `f(a,b)`, is simply written `f a b`). The `map` built-in higher order function is used to express the parallel application of the `comp` argument to the list of subimages (`split n x`) obtained by the application of the `split` argument to the `n` argument and the `x` input data[15].

Note that 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. It is, at the same time, an *executable specification*, which can be used to give a default sequential semantics to the skeleton, as soon as we have a sequential definition of the `map` higher order function in CAML. This definition is given in Appendix 3. We must underline, however, that this definition is only given here to support our claim concerning the concept of executable specification. The application programmer is definitely not required to inspect (and *a fortiori* to understand) it when *writing* skeletal programs with the SCM skeleton (this would require much more knowledge of CAML that is actually required for using SKIPPER !).

The CAML compiler will infer, from the previous definition, the following *type signature* for the `scm` higher order function:

```
val scm : int
          (* Type of n, the number of domains *)
        -> (int -> 'a -> 'b list)
          (* Type of the split function *)
        -> ('b -> 'c)
          (* Type of the compute function *)
        -> (int -> 'c list -> 'd)
          (* Type of the merge function *)
        -> 'a
          (* Type of the input data *)
        -> 'd
          (* Type of the result *)
```

This type signature (introduced by the `val` keyword) gives the type of all the arguments and result(s) (comments are delimited with `(*` and `*)`). Polymorphism comes from the use of *type variables*, denoted by letters `'a,...,'d`. Function types are denoted with an arrow, `(t1 -> t2 -> ... -> tn -> tr)` being the type of a function taking n arguments of type `t1` ... `tn` and returning a result of type `tr`. For example, the second argument (`split`) of the SCM higher order function is a function of type `int -> 'a -> 'b list`, i.e., a function accepting an integer and a value of any type `'a` and returning a list of values of type `'b`. The type signature expresses all the type constraints that the arguments will have to meet in order to apply the higher order function in a *consistent* manner. The sharing of the `'b` type variable in the signature of the `scm` skeleton will, for instance, preclude its application to a `split` function returning (let say) a list of `int`s and a `comp` function taking (let say) a `float`. This consistency check, called *polymorphic type checking*, is carried out by the CAMLFLOW front-end in SKIPPER (see Sect. 3.2.1).

The declarative definitions for the DF, TF and ITERMEM are given in Appendix 1.

Let us now look at how skeletal *programs* can be described in CAML using the SCM skeleton defined above. This is done by simply describing the dependencies between the operations involved in the algorithm, using function and/or skeleton applications on values defined by successive `let` expressions, as shown in Fig. 5[16].

Here `row_block`, `histo`, `merge_histo`, `get_img` and `display_histo` are the application-specific, sequential functions (written in C). `row_block` decomposes an image into horizontal subimages (blocks of rows, hence its name), `histo` computes the histogram of a (sub)image and `merge_histo` sums the partial histograms computed on each subimage into the final one. The `get_img` and `display_histo` functions, respectively, retrieve the next image from the video input stream and displays the histogram on the screen.

The program – along with the prototype and code for the `row_block`, ... `display_histo` functions in this case – forms the *skeletal program specification* appearing in Figs. 4, 6 and 12.

```
(* This program computes the histogram of an image
 * using a geometric partition
   in 4 horizontal strips *)

let im1 = get_img 256;;
let h = scm 4 row_block histo merge_histo im1;;
let main = display_histo h;;
```

Fig. 5. A simple skeletal program exhibiting the SCM skeleton

[13] There's no way, for instance, to define in C a second-order function taking an arbitrary function and returning the derivative of this function as another function – technically speaking to return a *closure*.

[14] For a list of realistic projects using (Ca)ML, see, for instance, http://www.cs.princeton.edu/∼appel/smlnj/projects.html or http://pauillac.inria.fr/caml/users_programs-eng.html

[15] This `map` higher order function can be formally defined by `map f [x1;x2;...xn] = [f x1;f x2; ... f xn]`, where `[a;b;...]` is the CAML notation for lists

[16] This form of programming is actually very close to the one used in *data-flow* or *single-assignment* languages like Sisal [31], Lucid [41] or Signal [4]. These are typically first-order languages, however, and hence cannot accommodate the skeleton concept.

Before describing how such a program can be turned into a form suitable for parallel execution on a target platform (or sequential emulation on a workstation), the following points must be made.

First, because of the implicitly sequential evaluation order of the *let* phrases, all the parallelism is located (restricted) at skeleton-specific higher order function applications. This is consistent with the approach stated at the beginning of Sect. 2, where the programmer was supposed to be responsible for explicitly annotating sites of useful parallelism.

Second, let us emphasize that relying on application-specific sequential functions written in C is of great practical importance: since real parallel applications are rarely written from scratch – and especially in CAML ! – any skeleton-based programming system not allowing the reuse of existing imperative sequential code is likely to be of little *practical* utility[17]. This point is, surprisingly enough (and apart from the work of Darlington et al. on Fortran-S [18]), rarely evoked in papers dealing with skeleton-based parallel programming.

Finally, note that the choice of the CAML language is by no means mandatory. We could have used plain ML or just any functional programming language. Apart from a matter of personal taste, it is mainly justified by the fact that CAML is publicly available, highly portable, and provides well-designed easy-to-use facilities for interfacing C and ML code (the last point being of great practical importance for the sequential emulation facilities described in Sect. 3.4).

3.2 Parallel implementation

By parallel implementation we refer to the process by which the parallel operational semantics of the skeletons is made explicit. Clearly, this phase strongly depends on an adequate intermediate program representation. This representation must reflect the possibilities of the potential target architectures without being bound to any one in particular. A classic one is *process networks*, in which nodes represent sequential computing processes and edge data transfers.

The whole compilation process transforms a skeletal program such as the one of Fig. 5 into code suitable for execution on a multi-processor architecture. It can be decomposed into four steps: process network generation, process template instantiation, mapping/scheduling and code generation[18]. It is illustrated in Fig. 6 (which is a zoom on the left box of Fig. 4).

3.2.1 Process network generation

This first step turns the CAML functional specification into a (target-independent) process network. It is handled by a modified version of the CAMLFLOW tool described in [36]. CAMLFLOW is a CAML-to-data-flow-graph translator. It performs parsing, type checking and produces a representation of the dependencies expressed in the program in the form of a *data flow graph*, using a technique called *abstract interpretation* [1]. An in-depth description of this technique is clearly out of the scope of this paper and can be found in [36]. Let us only say that it allows the *parametric process network* associated with each skeleton to be described *directly* in CAML. A *parametric process network* (PPN) is a graph of processes explaining the operational semantics of a skeleton. For the SCM skeleton, the PPN appears in Fig. 7. This process graph is parametric in the number of `comp` nodes, in the type of data items exchanged between nodes (denoted with type variables `'a` ... `'d`) and in the sequential functions run on the nodes `scm_s` and `scm_m` (this "parametrization" being denoted with brackets).

With CAMLFLOW this PPN can be completely described (encoded) by the following definition:

```
let scm n split comp merge x =
  (scm_m merge) n (pmap comp ((scm_s split) n x)),
```

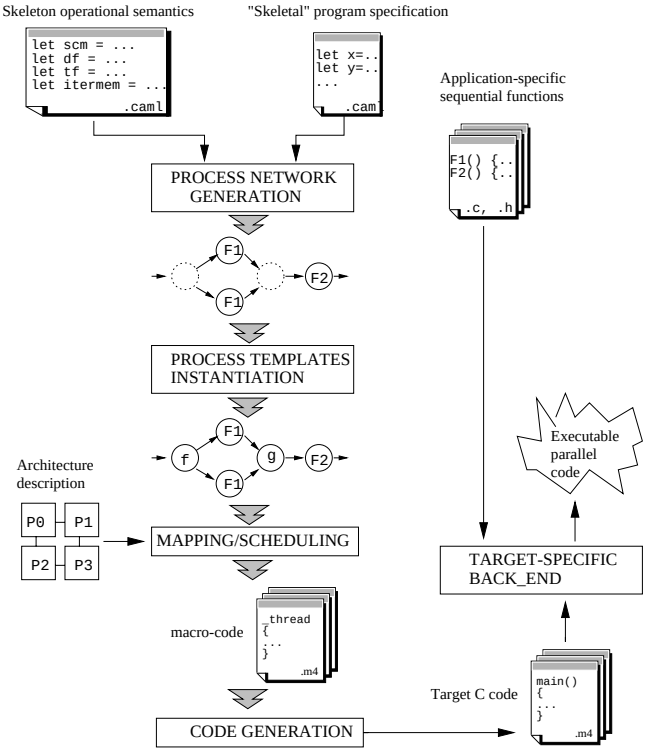


Fig. 6. Parallel compilation scheme in the SKiPPER parallel programming environment

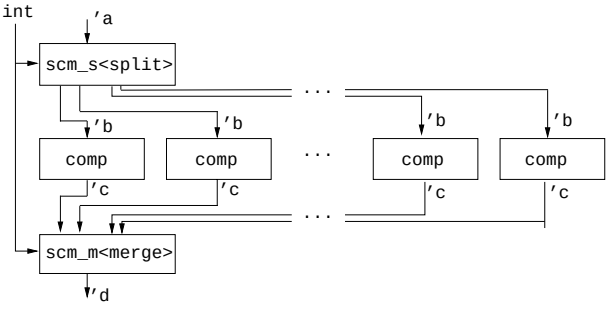


Fig. 7. The operational semantics of the SCM skeleton, as a parametric process network

With CAMLFLOW this PPN can be completely described (encoded) by the following definition:

```
let scm n split comp merge x =
  (scm_m merge) n (pmap comp ((scm_s split) n x)),
```

[17] Without prejudging, of course, its *theoretical* utility.

[18] Practically, these steps are carried out with an automatically-generated *makefile*.

```
SCM_S<split_fn>
  n := read input on port 1
  x := read input on port 2
  xs[1...n] := <split_fn>(n, x)
  for i = 1 to n do
    write xs[i] to output port i
  done
```

```
SCM_M<merge_fn>
  n := read input on port 1
  for i = 1 to n do
    read ys[i] on input port i
  done
  y := <merge_fn>(n, ys[1],...,ys[n])
  write y to output port 1
```

Fig. 8. The two parametric process templates (PPTs) used by the SCM skeleton

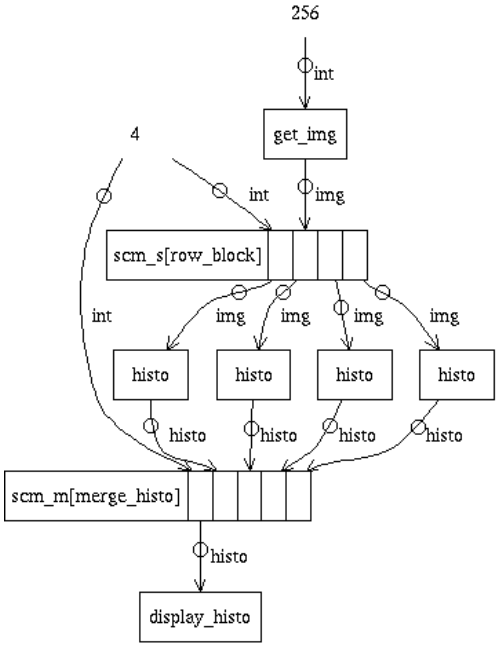


Fig. 9. The process network for the histogram application

where `scm_s` and `scm_m` are SCM-specific higher order functions and `pmap` a built-in higher order function for replicating a computation subgraph[19].

The behavior of the `scm_s` and `scm_m` processes is 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. The pseudo-code of the `scm_s` and `scm_m` PPTs is sketched in Fig. 8 (the actual code is written in C and the *specialization* (instantiation) process is carried out using macro-substitution).

The process graph and PPTs associated with the DF and ITERMEM skeletons are given in Appendix 2.

Applied to the program of Fig. 5, CAMLFLOW generates the process network of Fig. 9. In the latter graph nodes represent either user-defined sequential functions (`get_img` or `display_histo`) or PPTs (`scm_s` and `scm_m`), and edges represent (unidirectional) communications (tagged with the type of the corresponding data)[20].

3.2.2 Mapping and scheduling

This step maps the process network of the application onto the target architecture. The target architecture is, in the most general case, a *multi-component* graph, built from different types of processors connected together through a network of different types of communication components (point-to-point serial or parallel links, multi-point shared serial or parallel buses, etc.). This task therefore involves finding a static distribution of processes on processors and a static and/or dynamic scheduling of communications on channels. This is a "classic" problem, for which various heuristics have been proposed. We solved it using an existing third-party software called SynDEx [40]. SynDEx is a CAD software program developed at INRIA, implementing the AAA (Algorithm-Architecture Adequation) methodology. SynDEx accepts specifications of an algorithm and an architecture as graphs and performs a distribution and scheduling of the former on the latter using an heuristic based upon minimization of the global latency[21]. The use of SynDEx tool for mapping and scheduling the graph of Fig. 9 is illustrated in Fig. 10. The process graph can be recognized in the lower part of the upper window (labeled *edition*). The target architecture is specified in the left-upper corner of this window. A ring of four processors (labeled P0 to P3) is used here. The lower window (labeled *schedule*) illustrates the mapping of operations onto processors computed by SynDex, i.e.,the distribution of operations onto processors (one per column) and the scheduling of operations (oval boxes) and communications (diagonal lines) on each processor. On the basis of this mapping, SynDEx predicted a speedup of nearly 3 (0.75 efficiency). To compute this value, SynDEx clearly needs the duration[22] of all the application-specific functions (`row_block`, `histo`, etc.). So, practically, obtaining an accurate prediction requires two passes with SynDEx: A first pass, with *estimations* of these durations provided by the programmer, generates a first, suboptimal, parallel program, but from which *real* durations can be extracted using the automatic profiling mechanism described in Sect. 3.3. A second pass, with these real durations, generates the final program and accurate prediction.

SynDEx uses the result of its distribution and scheduling process to generate parallel code for the application. This code takes the form of a set of processor-independent programs (*m4* macro-code). There is one program per processor of the target architecture. The macro-code is built from a small kernel of processor-independent primitives. These primitives support boot-loading, static thread creation, inter-component communications, thread synchronization, inter-processor communications, and direct call of user-supplied sequential functions.

[19] Details on how this replication can be encoded at the language level can be found in the section of [36] dealing with data parallelism.

[20] This figure was automatically generated with a specific, graphical backend (using the DOT [25] format) of CAMLFLOW.

[21] The *latency* is defined as the critical timing path of the process network

[22] Sequential execution time corresponding to the "height" of the boxes in Fig. 10. Communication times are computed from the related data types.

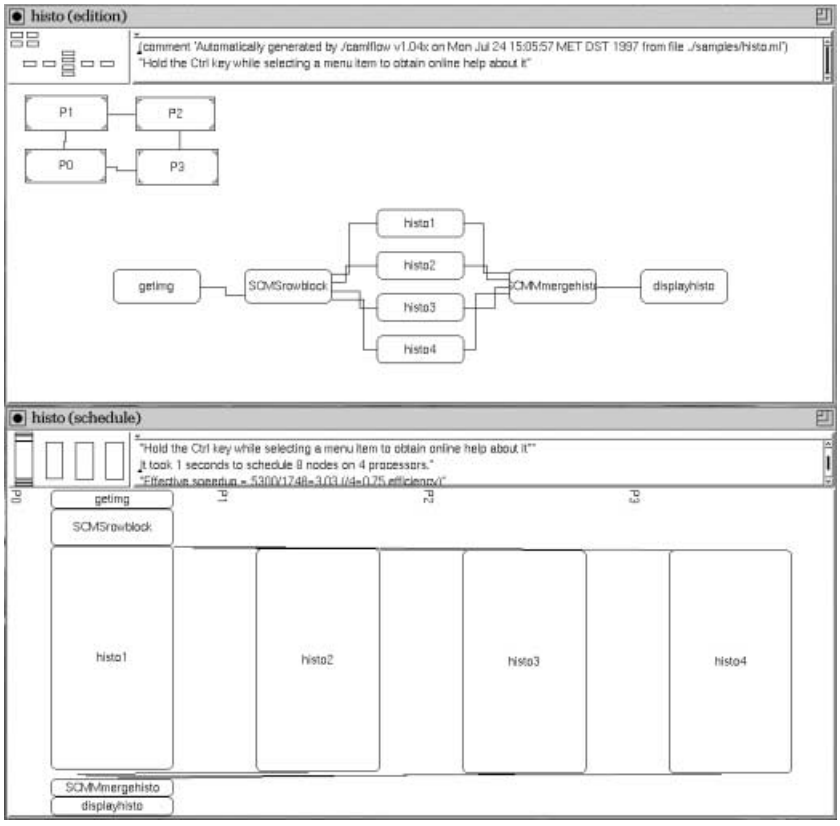


Fig. 10. A SynDEx session

3.2.3 Code generation

The macro-codes generated by SynDEx are finally turned into compilable code by simply inlining kernel primitives. As a result, retargeting an application on an architecture built from a new processor type only requires (re)writing this set of kernel primitives for this processor, taking into account the available hardware facilities and the target language (C, Fortran, assembler, etc.)[23]. The final parallel C code for the target architecture thus takes the form of a set of C source files (one per processor), in which a `main` function contains direct calls to the sequential functions attached to the scheduled operations, interleaved with the architecture-specific communication instructions for exchanging data between processors. Note that, in order to be called directly, the user-supplied sequential functions must adhere to a fixed calling convention. Practically, they must have a `void` return type, all non-atomic[24] must be passed by address, and all results returned by address. For example, the sequential functions of the histogram application discussed in this section have the following prototypes:

```
void get_img(/*in*/ int size, /*out*/ image *im);
void row_block(/*in*/ int n, image *im, /*out*/
     imageList *ims);
void histo(/*in*/ image *im, /*out*/ histo *h);
void merge_histo(/*in*/ int n, histoList *hs,
     /*out*/ histo *h);
void display_histo(/*in*/ histo *h);
```

with the following type declarations being provided elsewhere

```
typedef struct image { int nr; int nc; ... };
typedef struct { int sz; ... } histo;
```

3.3 Profiling

An important "by-product" offered by the SynDEx backend is the possibility to automaticaly insert chronometric instructions into the final code, thus allowing real-time performance measurement of the target applications. This property proved to be very useful insofar as understanding (and optimizing) the runtime behavior of parallel programs can only be done on the basis of accurate profiling information. This is especially true for a skeleton-based prototyping environment, for which the most effective way to assess the suitability of a skeleton in a given application context is simply to instantiate it and visualize its execution profile in the running application. It is sometimes argued against this rather *pragmatical* approach that the choice of the skeletons could (should?) rather be made *a priori* on the basis of theoretical performance models. Such models generally take the form of analytical formulae parametrized in computation and communication costs. Now in most cases, these costs cannot be easily and/or reliably estimated "off-line", i.e., without precisely running the application on a set of "representative" data ! Facilities for rapidly estimating the various cost factors for a given skeleton instantiation with various input data profiles are therefore definitely required. This is precisely what a fast prototyping environment, like

[23] The kernel definition for our T9000 processor is less than 300 lines of m4 code

[24] By *atomic* arguments we mean those having non-structured data types, such as `int`, `float`, etc.

the one described in this paper, offers, thus making performance models if not useless at least not mandatory for the casual application programmer.

Practically, each of the skeletons proposed in Sect. 2.1, comes with a variant *instrumented* version that automatically generates execution profiles at runtime. These profiles can be viewed and analyzed by existing tools such as the *upshot* utility. Figure 11 show the execution profiles generated by the instrumented version of the SCM skeleton in the case of the histogram application run on a ring of two, four, six and eight processors. On these profiles, each line corresponds to a distinct processor (processors are numbered from 0 to $n-1$). The grey boxes represent sequential computations. Larger boxes corresponding the `histo` functions and smaller ones to the `row_block` and `merge_histo` functions. Processors 1 to $n-1$ run only `histo`. Processor 0 runs `row_block`, `histo` and `merge_histo` (apart from `get_img` and `display_histo` which do not appear on the profiles). The most noticeable feature of these profiles is the very good load -balance achieved by the SCM skeleton in this case (the only source of innefficiency being the – inevitable – sequentialization of the functions on processor 0). This can be explained by the fact that the amount of work associated with the `histo` function is the same on each subimage. A detailed analysis of these profiles (using the "zoom" facilities of the `upshot` utility) will also show that computations and communications actually take place in parallel on processor 0 at the beginning of the application[25].

3.4 Sequential emulation

By *sequential emulation* we mean the ability to give a *sequential* interpretation – using a "standard" CAML compiler – to parallel skeletal programs.

This makes sense because – as long as the sequential functions of the programs interact only by means of skeleton composition – the sequential execution order enforced by a standard compiler is nothing but a particular instance of the *partial* execution order specified by the process network of the application. All that is needed is a *sequential semantics*, expressed as an "ordinary" higher order function, for each skeleton. Fortunately, and as already noticed in Sect. 3.1, a default sequential semantics is provided "for free" by the declarative semantics expressed in CAML as soon as the latter is based on "standard" higher order functions (such as `map`, `fold`, etc.), which is the case for all the skeletons defined in SKIPPER (Fig. 12). Technically speaking, this also requires the generation of some kind of *stub code* to convert data representations between C and CAML.

Apart from being parallel specifications, programs like the one given Fig. 5 are therefore *also* valid CAML programs which can be run, tested – by supplying relevant input data, observing results and, if a problem arises, using *sequential* debugging tools to overcome it – on any "traditional" sequential platform before being run on the parallel target. Since the corresponding parallel operational semantics of the involved skeletons are supposed to be correct, the application programmer will not even need to perform any debug task at the parallel level. This possibility of emulating parallel programs on sequential platforms to separate *functional* from "*implementational*" debug has already been proposed by Danelutto *et al.* in [15] under the name "logical debugging".

[25] Thanks to the T9000 DMA capabilites in our case

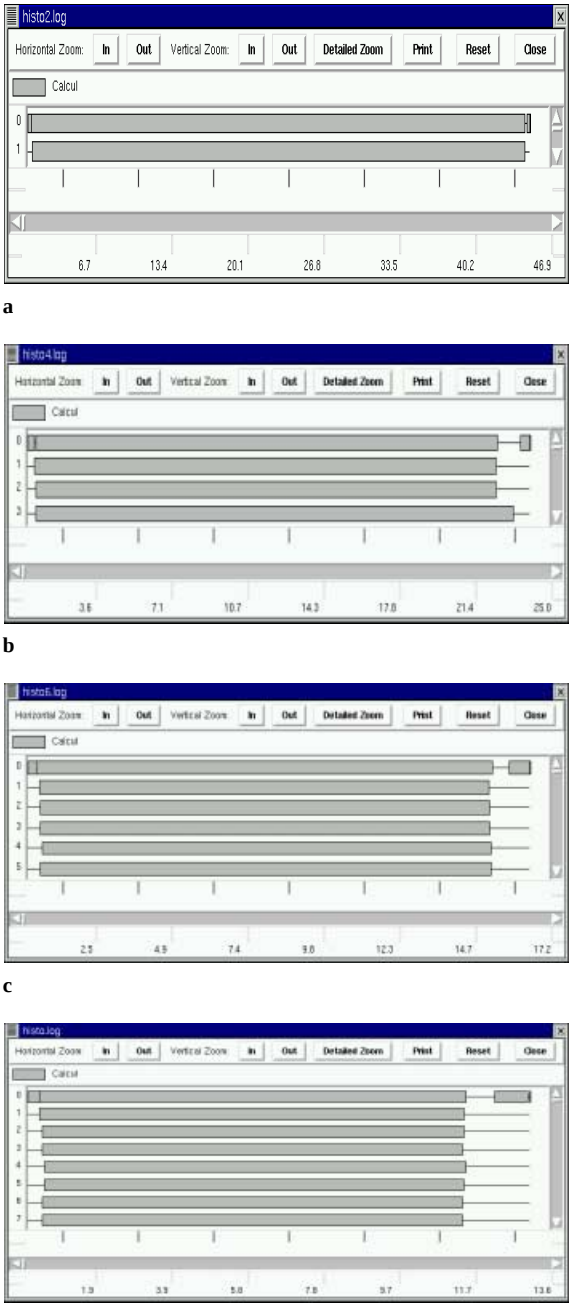


Fig. 11a–d.. Execution profiles of the histogram application: **a** with 2 processors; **b** with 4 processors; **c** with 6 processors; **d** with 8 processors

4 A realistic case study

This section illustrates the use of the proposed methodology for the parallel implementation of a realistic vision application. This application is based on a real-time road-

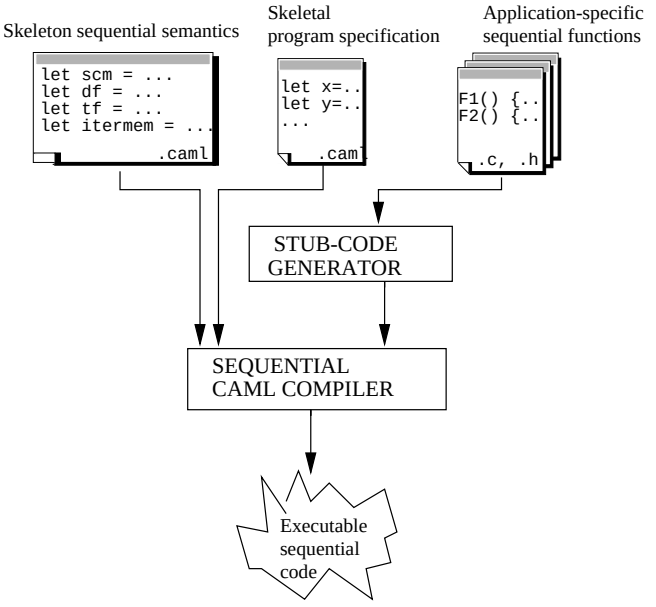


Fig. 12. Sequential emulation scheme in the SKIPPER parallel programming environment

following algorithm and is part of a larger project dedicated to computer-aided driving in semi-autonomous vehicles.

4.1 Goal and related work

Road following is an important problem encountered in autonomous vehicles projects as well as in road safety developments. Its main goal is the estimation of a vehicle location on a road in order to anticipate its trajectory.

Numerous projects have been designed for solving the road-following problem by using computer vision. Within the area of highway applications, several experimental vehicles like VAMORS [21, 22] and MOBLAB [7, 8] have been designed. Most of them use road extraction techniques based on the detection of the white lines in the current image and a road model to predict the position of the lanes in the next image [12, 21, 23, 24, 27, 28].

For the work described here, a vehicle is equipped with a monochrome camera which provides $512 \times 512 \times 8$ bits images at video frame rate (25 im/s). In these images, white lines are detected and used to compute the vehicle lateral location x_0, its direction angle ψ and the road curvature C (Fig. 13). We started with an algorithm formulation for which we already had several working implementations: a full-fledged sequential version on workstation platforms, a simplified real-time version for a DSP-based vision system, and a handcrafted full-fledged parallel version for a MIMD-DM parallel platform (TRANSVISION [30]). Such a set of implementations, of course, provides useful reference measures, both in terms of performance and of development effort, for assessing the usefulness of the proposed approach and tools.

4.2 Road modeling

Most of road-following algorithms rely upon an adequate modeling of the road for computing the vehicle location. In the case of highway navigation, this modeling can take advantage of several normalization factors, so that the road can be assumed to be locally flat, each white line being viewed as a curve with a continuous dynamic lateral curvature $C = 1/R$. The 3D model of such a line can then be expressed (Fig. 13b) as:

$x \approx Cy^2/2 + \lambda L$ and $z = 0$,

with L being the lane width (assumed constant) and λ = 0, 1 or -1 for the central, right or left line of the road, respectively.

According to several hypothesis [11, 12], the projection of these equations within the 2D image space of the camera is given by

$$u = e_u \left(-\frac{e_v z_0}{2(v - e_v\alpha)} C + \frac{v - e_v\alpha}{e_v z_0}(x_0 - \lambda L) - \psi \right), \qquad (1)$$

where u, v are the 2D image coordinates of the pixels belonging to a white line, α is the inclination angle of the camera (assumed constant), z_0 the camera height (constant), x_0 the dynamic lateral location of the camera (i.e., vehicle), C the dynamic road curvature, ψ the dynamic vehicle direction angle, and e_u, e_v constant intrinsic parameters of the camera.

Among these parameters, the only dynamic ones are C, x_0 and ψ. The *state vector* of the system will therefore be defined as $\underline{X} = (C, x_0, \psi)^t$.

The main goal of the algorithm will be to periodically compute the state vector $\underline{X}(k)$ at each discret time index k, using the information provided by the white lines in image $\underline{I}(k)$ and Eq. 1.

4.3 Algorithm

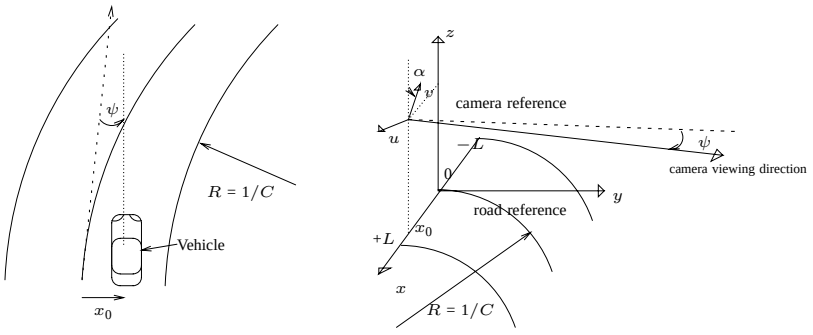
The whole algorithm can be divided into an initialization stage and a core looping process. The former, executed once, provides initial values for the latter. The core looping process can be further divided into three steps: prediction, detection, and updating.

4.3.1 Prediction

The prediction step aims at positioning a set of windows of interest (WOIs) in the image. The detection step will only search for white lines (Fig. 14a) in these windows. Compared to a global search in the whole image, this approach leads to much lower computational times and significantly reduces the risk of false detection for white lines.

Windows are equally dispatched along N_r horizontal rows, each row holding three windows (one per white line). The vertical location (v) of the horizontal rows and the size of the windows on one row are constant and have been experimentally chosen[26], but the size of the windows decreases with the vertical position of the corresponding row.

[26] They are set by the initialization stage



a **b**

Fig. 13a,b. The vehicle on the road and the modeling parameters

The prediction step is then responsible for updating the horizontal positions of the three windows on each horizontal row. This can be done, knowing the predicted state vector $\underline{X}(k|k-1)$, its covariance $\mathbf{C}(k|k-1)$, and by using Eq. 1. The parametric equations (u_λ, v_λ) of each white line in the image are computed and the WOIs are centered on the predicted locations. This requires giving an initial value to the state vector $\underline{X}$ at the first iteration. This value can be deduced from a global image analysis or simply set to a pre-defined constant. We use the second solution and set $\underline{X}(0) = (0, L/2, 0)^t$, assuming the vehicle to be initially located in the middle of the right lane of the road.

To summarize, the inputs of the prediction step are the state vector $\underline{X}(k|k-1)$, its covariance matrix, and the image $I(k)$. Its output is a list of WOIs $WS(k) = \{W_1, W_2, \cdots, W_{N_w}\}$, where N_w is the actual number of WOIs to process[27].

4.3.2 White lines detection

This step performs the detection of the position and orientation of white lines within the selected WOIs. It works by approximating a white line in a WOI by a rectilinear segment (Fig. 14b).

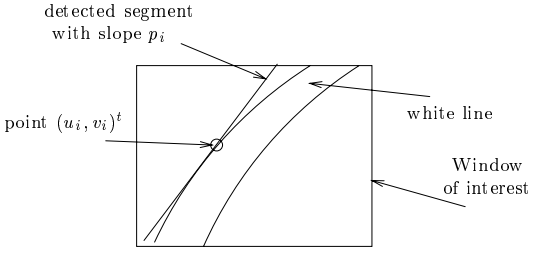
Localization of the approximating segment is carried out by first computing the horizontal gradient value for each point and then selecting the point with the maximum value in each row. The best fitting segment is finally obtained by applying a Hough transform to the resulting set of points. Two (constant) thresholds are set to eliminate false detections: a minimum value for the gradient and a minimum number of candidate points for the Hough transform.

At the end of the procedure, for each WOI i in which the detection has succeeded, the coordinates (u_i, v_i) of one point of the segment and its slope p_i are obtained.

The detection step therefore takes as input a list of WOIs $WS(k) = \{W_1, W_2, \cdots, W_{N_w}\}$ and outputs a list of *detection results* $DS(k) = \{D_1, D_2, \cdots, D_N\}$, with $N \leq N_w$ and $D_i = (u_i, v_i, p_i)$, where (u_i, v_i) are the coordinates of a point belonging to the white line edge in window W_i, and $p_i = \frac{du}{dv}(u_i, v_i)$ the orientation of the approximating segment at this point.

[27] $N_w \leq 3 \times N_r$, due to clipping effects

a



b

Fig. 14a,b. Interest zones used for the detection procedure. **a** location of WOIs; **b** white line detection in each window

4.3.3 Updating

The features $DS(k)$ computed by the detection step in the WOIs are used to update the dynamic parameters (C, x_0 and ψ) of the model. Formally speaking, this involves both providing the best estimations $\underline{X}(k|k)$ and $\mathbf{C}(k|k)$ of the state vector and its covariance for the current iteration and predicting their values $\underline{X}(k+1|k)$ and $\mathbf{C}(k+1|k)$ for the next iteration.

This is classically done using a Kalman filter and a state formalism described by the following equations:

$$\begin{cases} \underline{X}(k+1) = \mathbf{A}\underline{X}(k) + \underline{v}\,, \\ \underline{Y}(k) \quad\;\; = \mathbf{H}\underline{X}(k) + \underline{w}\,, \end{cases}$$

where:

- $\underline{X}(k) = (C, x_0, \psi)^t$ represents the state vector at discrete time k,
- **A**, the state matrix, taking into account the vehicle speed V and the time interval between updates Δ_t. It is given by

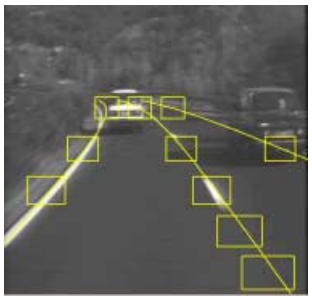


Fig. 15. Superimposed model after updating stage

$$\mathbf{A} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & V\Delta_t \\ 0 & 0 & 1 \end{pmatrix}$$

- $\mathbf{H} = (\mathbf{h}_{1a}, \mathbf{h}_{1b}, \ldots, \mathbf{h}_{ia}, \mathbf{h}_{ib}, \ldots, \mathbf{h}_{Na}, \mathbf{h}_{Nb})^t$ is the *measure matrix* (see later),
- $\underline{Y}(k) = (y_{1a}, y_{1b}, \ldots, y_{ia}, y_{ib}, \ldots, y_{Na}, y_{Nb})^t$ is the *observation vector*. This vector gathers two kinds of measures, y_{ia} and y_{ib}, which are in turn *computed* from the detection results $D_i = (u_i, v_i, p_i)$ using the following equations:
  - $y_{ia} = u_i + e_u \frac{v_i - e_v\alpha}{e_v z_0}\lambda L$. These observations take into account the u_i and v_i detection results. The corresponding measure submatrix is $\mathbf{h}_{ia} = (-\frac{e_u e_v z_0}{2(v_i - e_v\alpha)}, \frac{e_u(v_i - e_v\alpha)}{e_v z_0}, -e_u)$ and $y_{ia} = \mathbf{h}_{ia}\underline{X}$.
  - $y_{ib} = p_i + \frac{e_u}{e_v z_0}\lambda L$. These observations take into account the p_i detection result at coordinates (u_i, v_i). The corresponding measure submatrix is $\mathbf{h}_{ib} = (\frac{e_u e_v z_0}{2(v_i - e_v\alpha)^2}, \frac{e_u}{e_v z_0}, 0)$ and $y_{ib} = \mathbf{h}_{ib}\underline{X}$.
- $\underline{v}$ and $\underline{w}$ are noises assumed white, gaussian and uncorrelated.

Strictly speaking, the algorithm result is the estimated state vector $\underline{X}(k|k)$, since this vector contains, for instance, all the information required for controling an autonomous driving process. For tuning and demonstration purposes, it is also very useful to explicitly display this vector, for instance, by drawing the projection of the 3D reconstructed road model on the acquired image using Eq. 1 (see Fig. 15). This kind of super-imposition of a reconstructed model on the real scenes has proved to be very useful in the prototyping phase of the algorithm.

4.4 Parallelization

Figure 16 is a data-flow-like representation of the algorithm, showing the different steps involved along with the various data dependencies (each edge is labeled with the name and type of the corresponding data).

From this representation, it is clear that the whole process is an instance of the ITERMEM skeleton, in which the input function is `getimg`, the output function is `visualize`, the loop function is `update` ∘ `detect` ∘ `predict`, and the memory holds a value of type `state` (where `state` contains both the state vector $\underline{X}$ and its covariance $\mathbf{C}$).

So, we can start with the following top-level CAML program (values such as $\underline{S}(k|k-1)$ are here represented by identifiers such as `skkp`, where `p` stands for "previous")

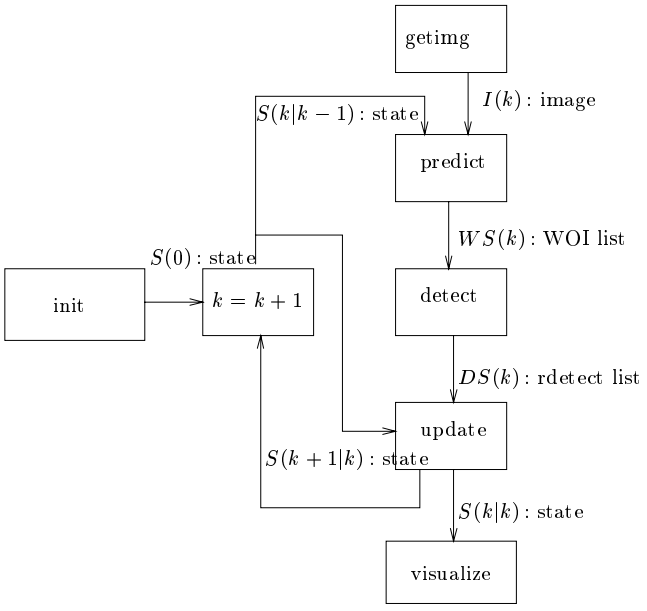


Fig. 16. Data-flow representation of the road-following algorithm

```
let loop (skkp, ik)
    = (* ... still to write ... *);;
let s0 = initstate ();;
let main
    = itermem getimg loop visualize s0 (512,512);;
```

with the following type definitions and prototypes for the sequential C functions:

```
typedef struct { int x,y; int nr,nc; byte *data;
                 ... } image;
typedef struct { float X[3]; float C[9]; } state;
void getimg(/*in*/ int nrows, /*in*/ int ncols,
            /*out*/ image *im);
void init(/*out*/ state *st);
void visualize(/*in*/ state *st);
```

The next step consists in identifying useful parallelism in the central `loop` function.

Given its low complexity and small data set, the `predict` function is only responsible for a very small percentage of the total computation cost of the `loop` function. So we are left with the parallelization of the `update` and `detect` functions.

For `detect`, it can be noted that, because of the clipping effect involved by the retro-projection process[28], the size of its input list (`ws`) may vary from one iteration to another. Furthermore, the size of each WOI, and hence the complexity of the `detect` process itself vary. This results in a rather uneven global workload for the detect process. This naturally calls for a DF skeleton. So we could write

```
let loop (skkp, ik) =
  let ws = predict skkp ik in
  let ds = df detect accum [] ws in
  update skkp ds;;
```

where

[28] Some WOIs may "fall outside" of the 2D image plan. In this case, they are simply discarded.

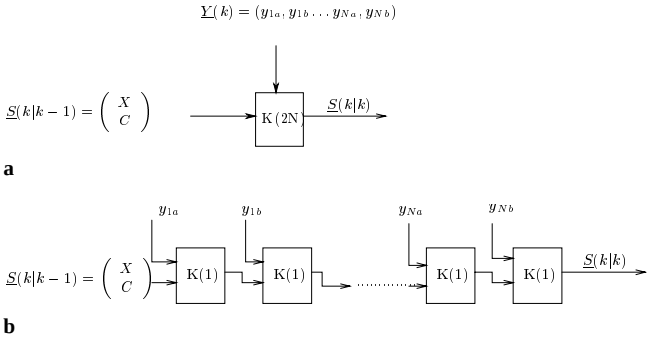


Fig. 17a,b. A possible optimization of the Kalman filter implementation. **a**

– `predict` is the sequential C function implementing the prediction step:

```
void predict(/*in*/ state *skkp,
             /*in*/ image *ik,
             /*out*/ woiList *ws);
```

– `detect` is the sequential C function performing the detection task within one WOI

```
typedef struct { int u,v; int du,dv; ... } woi;
typedef struct { int ui,vi; float pi; ... }
   rdetect;
void detect(/*in*/ woi *w, /*out*/ rdetect *d);
```

– `accum` just performs the accumulation of detected features in a list

```
void accum(/*in*/ rdetectList *old,
           /*in*/ rdetect *d,
           /*out*/ rdetectList *new);
```

– `update` takes into account the detected features to compute both the updated state ($S(k|k)$) and its predicted value for the next iteration ($S(k+1|k)$)

```
void update(/*in*/ state *skkp,
            /*in*/ rdetectList *ds,
            /*out*/ state *skk,
            /*out*/ state *xkkn);
```

– `[]` is a pre-defined atom denoting an empty list

With this solution, the `update` function is responsible for applying the Kalman filter over the whole data set accumulated by the (farmed) `detect` function. If N detection results are obtained, this involves inverting a $2N \times 2N$ matrix. If computed sequentially, this leads to prohibitive execution times for the `update` function[29]. One solution would be to parallelize the matrix inversion, using some well-known algorithm from the parallel literature. Another approach is to rely on the fact that, if we assume that the detection results are not correlated, it is possible to replace the application of one Kalman filter operating on a $2N$ vector by the successive applications of $2N$ Kalman filters operating on scalars. This transformation is depicted in Fig. 17. In this case, the application of the Kalman filter can be carried out as soon as the corresponding detection result is available.

These successive applications can therefore be "lifted up" in the `accum` function of the DF skeleton implementing the detection step. The updating of the state vector is then performed in an incremental manner and in *true parallelism* with the detection functions (the latters running on the *worker* processors, the former on the *master* processor). The `update` function[30] is now only in charge of computing the predicted state $S(k+1|k)$ from its updated value $S(k|k)$.

The `loop` function is now rewritten as follows:

```
let loop (skkp, ik) =
  let ws = predict skkp ik in
  let skk = df detect  accum skkp ws in
  update skk
```

with the following prototypes for the variant C sequential functions

```
void accum(/*in*/ state *before,
           /*in*/ rdetect *d,
           /*out*/ state *after);
void update(/*in*/ state *skk,
            /*out*/ state *skk,
            /*out*/ state *skkn);
```

Note also that an explicit accumulation of the detection results in a list is no longer needed since the previous state value $S(k|k-1)$ now directly serves as the initial value for the `accum` folding function.

4.5 Results

Starting from the formulation given in the previous section, the suite of tools described in Sect. 3 can be used both to check the correctness of the parallelization process (by using the sequential emulation facilities) and to automatically derive parallel implementations. For the latter, we specifically targeted a multi-processor vision machine with real-time video i/o facilities, the TRANSVISION platform [30]. This architecture is built upon T9000 Transputer processors and can be configured according to various physical topologies[31]. The experiment here has been conducted using a ring-topology. All processors can simultaneously synchronize on the video stream, so that true real-time processing (25 images/s) in MPMD (Multiple Program Multiple Data) mode is possible (for a total compute time not exceeding 40 ms, of course).

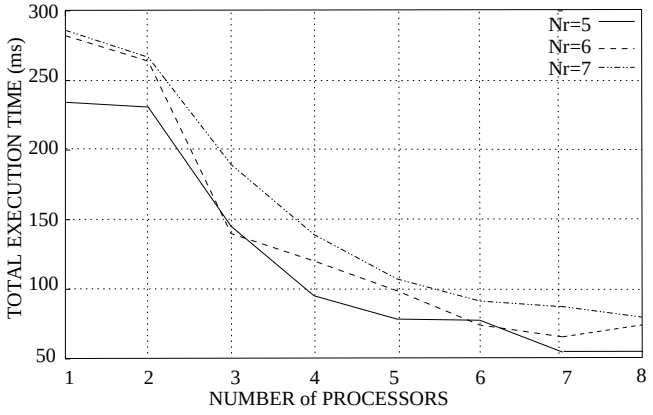
Performance results are shown in Fig. 18a,b for three values of the N_r parameter (number of rows carrying WOIs, as defined in Sect. 4.3.1). The higher N_r is, the more robust the algorithm is, since this corresponds to a larger observation set, and hence allows a more precise reconstruction of the road model (another possibility when increasing the number of WOIs would be to handle more than three white lines, but this case is not illustrated here).

With a ring of N_p = 8 transputers operating on a 25-Hz 512 × 512 video stream, the minimal execution times obtained range from 55 ms (for N_r = 5) to 70 ms (for N_r = 7),

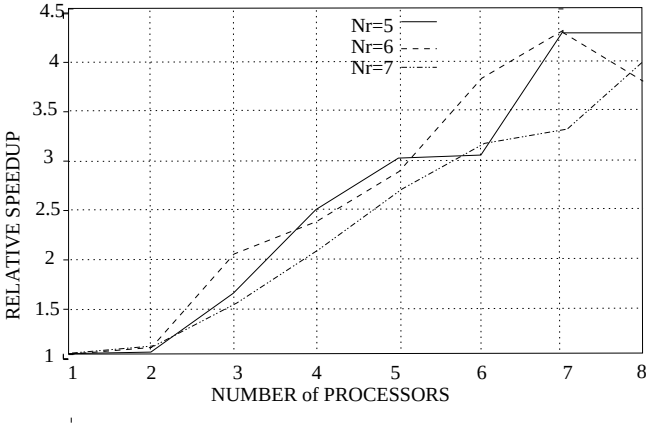
[29] All gains obtained by the parallelization of `detect` are lost due to the cost of the sequential matrix inversion

[30] Strictly speaking, it should not be called `update` any longer

[31] The TRANSVISION machine was the only platform available at the time the work reported in this paper was conducted. Work is underway to port the SKIPPER programming environment to more recent hardware (a DEC-Alpha-AXP21066-based machine and a Beowulf-style network of PCs)



a

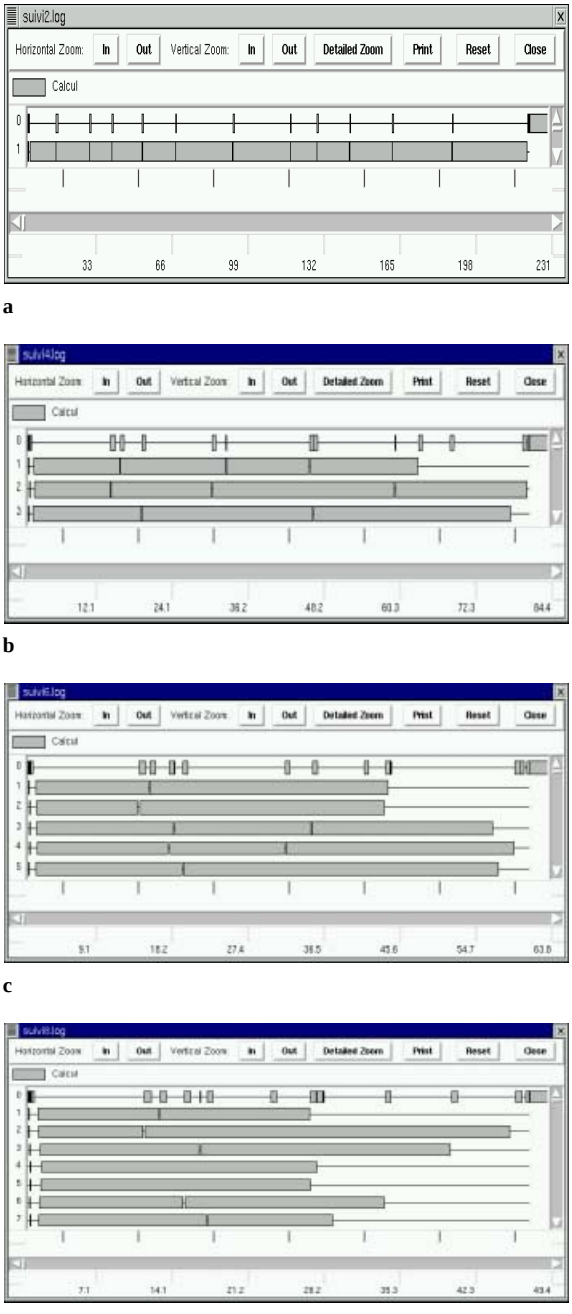


b

Fig. 18a,b. Total latency and speedup for N_r = 5, 6, 7 and N_p = 1 to 8

giving absolute speedups of nearly 4 (0.5 efficiency). This leads to a maximum effective throughput of 12 images per second, with the algorithm processing one image out of two[32]. Such performances satisfy the timing constraints of the target application and are similar to the one obtained by the handcrafted parallel version (the latter being estimated at 40 ms, after application of a sequential correction factor, since it was obtained on a machine equipped with slower T800 processors). Figure 19a–d gives the execution profiles of the application for N_r = 5 (leading to a maximum of 15 WOIs) and N_p = 2, 4, 6, 8 (leading to 1,3,5, and 7 workers for the DF skeleton). Note that these execution profiles were simply and automatically obtained by using the *instrumented* version of the DF skeleton, with absolutely no change to the application code. Only the `acc` (here `accum`) and `f` (here `detect`) DF parameters are represented here as horizontal boxes with length proportional to their execution time. Horizontal lines correspond to processors (processor numbered 0 is the farmer, others are workers). Three points may be underlined: first is the large variability in the execution times of the `detect` function (on processors 1 to 7). This, of course, is a direct consequence of the variability in the size of the `wois` and subsequently justifies the use of the DF skeleton for this application step. Second is the very small execution times for the `accum` function (on processor 0),

[32] If it takes between 40 and 80 ms to process image i, image $i+1$ (40 ms later) is lost and all processors will synchronize on image $i+2$ (80 ms later)



a

b

c

d

Fig. 19a–d. Four execution profiles for the road-following applications; **a** with 2 processors; **b** with 4 processors; **c** with 6 processors; **d** with 8 processors

thanks to the Kalman transformation introduced in Sect. 4.4. As a result, processor 0 is idle most of the time. This must not be interpreted as an intrinsic flaw of the DF implementation. Rather, because the `acc` function is computed on the farmer in true parallelism with the `f` one on the workers, the farmer idle time would decrease as soon as this function takes longer to execute, and this *without increasing the total execution time*. Finally, the automatic load-balancing effect provided by the dynamic dispatching of the N WOIs is obvious in these figures, even if a slight degradation of this balancing effect is observed for $N_p > 5$. This can be ex-

plained by the fact that the ratio N/N_p is then too small for allowing a uniform distribution. This also accounts for the rather low efficiency for the largest values of N_p (maximum speedup is 4.2 for eight processors). Note that increasing the N_r number, in order to increase the number of dispatched WOIs – and hence to allow a better load balancing among workers – does not help significantly, because the number of dispatched WOIs (N) is practically limited by clipping effects (even with N_r = 7, it practically never exceeds 17, for a theoretical maximum of $3 \times 7 = 21$).

We must say here that the main lesson drawn from this experiment, however, was not on raw performances but on the effectiveness of the skeleton approach and of the proposed suite of tools for implementing complex real-time vision applications on parallel platforms.

First, let us recall that the programmer's work here reduced to writing seven sequential C functions (`init`, `predict`, `detect`, `accum`, `update` and `visualize`) and the seven-liner top-level CAML specification. Moreover, most of these functions could be derived from an existing sequential implementation of the algorithm. All underlying parallel implementation details (including process placement, communication scheduling, buffer allocation, provision for deadlock avoidance, etc.) were transparently handled by the environment. Two numbers may give an idea of the "added value" provided by this environment:

- first, 6000 lines of parallel C code were generated starting from about 1000 lines of sequential C. This 6:1 ratio can be viewed as a measure of the complexity added when passing from the sequential formulation of an algorithm to a parallel one.
- second, the code generated by the back-end for each processor contains about 50 semaphores acting as synchronization points between one computation thread and four communication threads. This can be attributed to the runtime complexity of the process farm protocol.

The result is that it took less than 2 days to set up a first working implementation on the target platform and that it was then almost instantaneous to obtain variant versions (with different values of N_w, of N_p, *etc*). Compared to the previously handcrafted parallel versions, this represents dramatic savings in development effort, since these version had taken at least ten times longer to implement and cannot be scaled in a straightforward way (modifying the processor number, for instance, required significant changes in the C code). Needless to say that, because of the complexity of the farming protocol, these handcrafted versions were never proved deadlock free.

Second, thanks to the SynDEx retargetable back-end, it would be straightforward to port the application to another parallel platform, provided an executive kernel is available for this platform. This has been demonstrated, at least for a network of Ethernet-ed Sun workstations (although the performance is in this case rather disappointing, due to the very high latency of Ethernet).

Third, the possibility to emulate the parallel code on a sequential workstation (described in Sect. 3.4) has proved to be a very useful approach for debugging the application *functionality* without having to deal with a complex parallel environment. Several bugs in the *sequential* C functions have thus been uncovered. Tracking them down in the *parallel* version would have been much more difficult (if not impossible, given the very limited debugging support offered by our machine).

Last, the availability of *instrumented* (profiled) versions for the chosen set of skeletons, delivering execution profiles without any intervention in the application code, greatly eases the interpretation of the parallel execution results, even for programmers not familiar with the underlying implementation details.

5 Related work

Given the promising features of skeleton-based approaches to parallel programming stated in Sect. 2, it is not surprising that many groups have worked (and are still working) on skeletons, both from a theoretical point of view and on practical implementations. Since a complete survey is out of the scope of this paper, we will only cite here the projects whose goals are closest to ours. A more thorough survey can be found in [9] and the article of Cole in [26].

Darlington's group at Imperial College London has thoroughly explored the issues related to the implementation of a skeletal parallel-programming environment [17, 19, 20]. Their approach shares our "layered" view of skeletons as *coordinating* constructs for sequential functions written in an imperative language. In [18] for example, the SCL (Structured Coordination Language) language is used to coordinate sequential Fortran code. Examples are only given however for rather simple linear algebra algorithms and run on a Fujitsu AP-1000.

The P3L project at Pisa University [2, 16] has developed a fully fledged parallel-programming language based on the concept of skeletons and associated implementation templates. A distinction is made between task-parallel skeletons (`farm`, `pipe`), data-parallel skeletons (`map`, `reduce`) and control skeletons (`loop`). Sequential parts of a P3L application may be written in many sequential languages (C, Pascal, etc.) and skeletons are introduced as special constructs using a C-like syntax. The P3L compilers generate code for transputer-based Meiko machines and for PVM running on a cluster of UNIX workstations. Recently, Danelutto et al. [15] have proposed an integration of the P3L skeletons within the CAML language, making program specifications looking very similar to ours (with a similar concern for sequential emulation, for instance). But both P3L and OcamlP3L require either a good OS-level support (Unix socket) or a generic message-passing library (MPI), for their implementation. This precludes their use on heterogeneous and/or dedicated vision platforms.

Michaelson's group at Heriot-Watt University in Edinburgh has significant experience in the application of functional programming to vision applications [5, 32, 34]. Skeletons are defined as higher order functions in ML and their latest definition of a vision-specific skeleton basis is very similar to ours [with comparable definitions for `df` and `tf`, and `gd` (grid-decomposition) corresponding to our `scm`]. Their work differs from ours in three points, however. First and most noticeably, their goal is an *implicitly* parallel system within which the decision of expanding a skeleton higher

order function into parallel constructs at the implementation level is taken by the compiler[33] and is not in the programmer's hand. Second, no provision appears to be made for (re)using sequential functions written in C (all the program is written in ML). Finally, they never seem to have targeted dedicated platforms with real-time, on-the-fly processing capabilities (results are given for a Meiko CS, a network of PCs and a Fujitsu AP-1000).

6 Conclusion and future work

In this paper, the *skeletal* approach to parallel-program development is assessed in the context of real-time image processing, to see whether it can provide a solution to the problem of fast prototyping of vision applications on dedicated parallel hardware.

This assessment has been carried out by first identifying a small number of domain-specific skeletons and then building a suite of tools capable of turning a high-level, architecture-independent specification of an algorithm into executable parallel code.

The conclusions, supported here by a realistic case study, are very encouraging. First , the "off-the-shelf" style provided by the skeleton approach effectively provides dramatic savings in development effort, allowing the application programmer to both concentrate on truly algorithmic aspects rather than on low-level implementation details and to try and evaluate a large number of parallelization schemes. Second, this significant increase in *programmability* is *not* obtained at the price of a significant decrease in *efficiency*, at least for the examples presented in this paper. The skeletons may, in fact, be viewed as a very effective way to *encapsulate* and reuse the expertise gained by skilled parallel programmers. Third, and thanks to a retargetable back-end, the portability of the tools remains high (the porting effort is here reduced to reimplementing a bunch of primitives for a small executive kernel).

The work described here also helped in identifying several key issues in any skeleton-based parallel-programming environment. These issues, which will form the basis of our future work, are the possibility of freely nesting skeletons (currently limited to the ITERMEM skeleton). This possibility was not needed for the kind of algorithms we dealt with, but may prove essential for more complex applications. In the same vein, the specification and implementation of various inter-skeleton transformational rules [5] for optimization is likely to become inevitable (the simple composition of two `scm` skeletons, for instance, may become very inefficient if both of them have to split and merge large blocks of images, but this can be optimized away if the splitting and merging functions are identical). Finally, it remains to be seen whether the whole approach can be extended to higher levels of image processing, for which the irregularity of algorithms is much higher and may prevent the identification of highly reusable skeletons.

[33] On the basis of profiling information collected by an *instrumentation* phase.

Acknowledgements. The authors would like to thank the anonymous referees for many insightful comments which led to improvements in this paper.

References

1. Abramsky S, Hankin C (1987) Abstract Interpretation of Declarative Languages. Ellis Horwood, Chichester, U.K.
2. Bacci B, Danelutto M, Orlando S, Pelagatti S, Vanneschi M (1995) P^3L: A Structured High-Level Programming Language and its Structured Support. Concurrency – Pract Exper 7(3): 225–255
3. Bellon A, Dérutin JP, Heitz F, Ricquebourg Y (1994) Real-time collision avoidance at road-crossings on board the Prometheus prolab-2 vehicle. In: Intelligent Vehicles Symposium, Paris, Oct 1994
4. Benveniste A, Le Guernic P, Jacquemot C (1991) Synchronous programming with events and relations: the SIGNAL language and its semantics. Science of Computer Programming 16(2): 103–149
5. Bratvold TA (1994) Skeleton-Based Parallelisation of Functional Programs. PhD thesis, Heriot-Watt University, Edinburgh
6. Bratvold TA (1992) Determining useful parallelism in higher order functions. In: Kuchen H, Loogen R (eds) Proceedings of the 4th International Workshop on Parallel Implementation of Functional Languages. Technical Report No. 92–19, RWTH Aachen, Aachen, Germany
7. Broggi A (1995) An image reorganization procedure for automotive road following systems. In: International Conference on Image Processing, Washington, DC, Oct 1995. IEEE Computer Society, Bellingham, N.J.
8. Broggi A, Bertozzi M, Gregoretti F, Passerone F, Sanso C, Reyneri L (1997) A dedicated image processor exploiting both spatial and instruction-level parallelism. In: CAMP'97 – Computer Architectures for Machine Perception, Boston, Oct 1997. IEEE Computer Society, Bellingham, N.J.
9. Campbell DKG (1996) Towards the classification of algorithmic skeletons. Technical Report YCS 276, Department of Computer Science, University of York, York, UK
10. Canals R (1993) Implantation d'algorithmes de segmentation d'images sur la machine parallèle TRANSVISION. PhD thesis, Université Blaise Pascal, Clermont-Ferrand, France
11. Chapuis R (1991) Suivi de primitives image, application la conduite automatique sur route. PhD thesis, Université Blaise Pascal, Clermont-Ferrand, France
12. Chapuis R, Potelle A, Brame JL, Chausse F (1995) Real time vehicle trajectory supervision on the highway. Int J Robotics Res 14(6): 531–542
13. Cole M (1989) Algorithmic skeletons: structured management of parallel computations. Research Monographs in Parallel and Distributed Computing. Pitman/MIT Press, London
14. Cousineau G, Mauny M (1998) The functional approach to programming. Cambridge University Press. Software and documentation available from `http://pauillac.inria.fr/caml`.
15. Danelutto M, DiCosmo R, Leroy X, Pelagatti S (1998) Parallel functional programming with skeletons: the OCamlP3L experiment. In: Proceedings ACM workshop on ML and its applications. Cornell University
16. Danelutto M, Pasqualetti F, Pelagatti S (1997) Skeletons for data parallelism in p3l. In: Lengauer C, Griebl M, Gorlatch S (eds) Proc. of EURO-PAR '97, Passau, Germany, vol. 1300 of LNCS, pp. 619–628. Springer, Berlin Heidelberg New York
17. Darlington J, Field AJ, Harrison PG, Kelly PHJ, Sharp DWN, Wu Q, While RL (1993) Parallel programming using skeleton functions. In: Parallel Architectures and Languages Europe. Springer, Berlin Heidelberg New York
18. Darlington J, Guo Y, To HW, Wu Q, Yang J, Kohler M (1994) Fortran-S: A uniform functional interface to parallel imperative languages. In: Proceedings of the Third Parallel Computing Workshop, Fujitsu Laboratories Ltd, Kawasaki Japan
19. Darlington J, Guo YK, To HW, Yang J (1995) Functional Skeletons for Parallel Coordination. In: EuroPar'95 – European Conference on

Parallel Processing, vol. 966 of Lecture Notes in Computer Science, pp 55–69, Stockholm, Sweden. Springer, Berlin Heidelberg New York
20. Darlington J, Guo YK, To HW, Jing Y (1995) Skeletons for structured parallel composition. In: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming
21. Dickmanns ED, Behringer R, Brudigam C, Thomanek F, Holt V (1993) An all-transputer visual autobahn-autopilot/copilot. In: 4th ICCV, Berlin, Germany, pp 608–615
22. Dickmanns ED, Mysliewetz BD (1992) Recursive 3d road and relative ego-state recognition. Trans Pattern Anal Mach Intell 14(2): 199–213
23. Diebolt F (1996) Road markings recognition. In: International Conference on Image Processing 96, Lausanne, Switzerland, Sep 1996
24. Ekinci M, Thomas B (1996) Road junction recognition and turn-offs for autonomous road vehicle navigation. In: International Conference on Pattern Recognition 96, Vienna, Austria, Aug 1996
25. Ellson J, Gansner E, Koutsofios E, North S. Graphviz. Available in the World Wide Web at `http://www.research.att.com/sw/tools/graphviz`.
26. Hammond K, Michaelson G (eds) (1999) Research Directions in Parallel Functional Programming. Springer, London
27. Herman M, Nashman M, Hong TH, Schneiderman H, Coombs D, Young GS, Raviv D, Wavering AJ (1997) Visual navigation: from biological systems to unmaned ground vehicles., Chapter 10: Minimalist vision for navigation, pp 275–316. Aloimonos Y. (ed) Lawrence Erlbaum Associates, Mahwah, NJ
28. Huang CC, Gillies DF (1995) Determination of an autonomous vehicle position by matching the road curvature. In: Proceedings of SPIE, Vol. 2591 Mobile Robot X, pp 14–24
29. Legrand P (1995) Schémas de parallélisation d'applications de traitement d'iamges sur la machine parallèle TRANSVISION. PhD thesis, Université Blaise Pascal, Clermont-Ferrand, France
30. Legrand P, Canals R, Dérutin JP (1993) Edge and region segmentation processes on the parallel vision machine Transvision. In: Computer Architecture for Machine Perception, pp 410–420, New-Orleans, La.
31. MCGraw J, Skedzielewski S, Allan S, Oldehoeft R, Glauert J, Kirkham C, Noyce B, Thomas R (1985) Sisal : Streams and iteration in a single assignment language. Technical report m-146, Lawrence Livermore National Laboratory
32. Michaelson GJ, Scaife NR (1995) Prototyping a parallel vision system in standard ML. J Funct Program 5(3): 345–382
33. Paulson LC (1991) ML for the Working Programmer, second edition, Cambridge University Press, New York
34. Scaife NR (2000) A dual source parallel architecture for computer vision. PhD thesis, Heriot-Watt University, Edinburgh
35. Scaife NR, Michaelson GJ, Wallace AM (1997) Four skeletons and a function. In: Davie T, Clack C, Hammond K (eds) International Workshop on Implementation of Functional Languages, pp 529–538, Sep 1997
36. Sérot J (2000) Camlflow: a caml to data-flow translator. In: Gilmore S (ed) 2nd Scottish Functional Programming Workshop, Jul 2000
37. Skillicorn DB (1995) Foundations of parallel programming. Number 6 in International series on parallel computation. Cambridge University Press
38. Skillicorn DB, Talia D (1998) Models and languages for parallel computation. ACM Comput Surv 30(2): 123–169
39. Skillicorn SB (1990) Architecture-independent parallel computation. IEEE Comput 23(12): 38–50
40. Sorel Y (1994) Massively parallel systems with real time constraints. The "Algorithm Architecture Adequation" Methodology. In: Proc. Massively Parallel Computing Systems, Ischia Italy
41. Wadge WW, Ashcroft EA (1985) Lucid, the Data Flow Programming Language. Academic Press, London

Appendix 1

This section gives the declarative semantics (in the form of CAML definitions) of the DF, TF and ITERMEM skeletons. These definitions also provide a default sequential semantics.

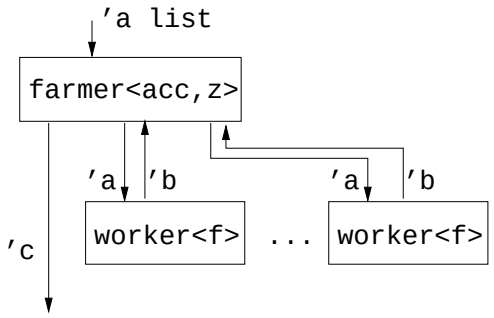


Fig. 20. The parametric process network for the DF skeleton

The DF skeleton

Its definition can be written

```
let df comp acc z xs
    = fold_left acc z (map comp xs),
```

where

- `xs` is the list of data items to process,
- `comp` is the function applied to each item,
- `acc` performs the accumulation of partial results,
- `z` is initial value of the accumulator.

The `fold_left` higher order function is the CAML built-in function for iterating a binary operator over a list of elements. It can be formally defined with the following equation:

$$fold\_left\ f\ z\ [x_1, x_2, \ldots, x_n] = (\ldots(f(f\ z\ x_1)\,x_2)\ldots x_n)\,.$$

For example, if `add` denotes the function computing the sum of two integers, then

```
fold_left add 0 [1;2;3] = ((0+1)+2)+3 = 6,
```

and if `mul2` denotes the function multiplying its argument by 2,

```
df mul2 add 1 [4;5;6] = ((1+(4*2))+(5*2))+(6*2).
```

The signature of the DF higher order function is

```
val df : ('a -> 'b )
         (* Type of the compute function *)
       -> ('c -> 'b -> 'c)
          (* Type of the accumulating function *)
       -> 'c
          (* Type of the accumulator value *)
       -> 'a list
          (* Type of the input list *)
       -> 'c
          (* Type of the result *)
```

The TF skeleton

Its definition can be written

```
let rec tf h solve divide accum z xs =
  let f x =
    if (h x) then accum z (solve x)
    else tf n h solve divide accum z (divide x)
  in
  fold_left accum z (map f xs)
```

```
FARMER<acc_fn,init_acc>
  xs := read input list on slot 1
  idle_workers := list of workers
  busy_workers := 0
  while ( xs not empty ) do
    x := head(xs)
    xs := tail(xs)
    if ( idle_workers = empty ) then
      yi :=  receive result from worker i
      send x to worker i
      y := <acc_fn>(y, yi)
    else
      w := head(idle_workers)
      idle_workers := tail(idle_workers)
      send x to worker w
      busy_workers := busy_workers+1
  done
  for i = 0 to busy_workers-1 do
    yi = receive result from worker
    y := <acc_fn>(y, yi)
    done
  send "stop" packet to every worker
  write y to output slot 1
```

```
WORKER<comp_fn>
  running := true
  While(running = true) do
    x := recv packet from farmer
    if x = "stop" then
      running := false
    else
      y := <comp_fn>(x)
      send y to farmer
  done
```

Fig. 21. The two parametric process templates used by the DFM skeleton

The construct `let v = <defn> in <body>` is used to introduce local definitions in CAML (the `f` function, applied to each item of the list, here).

Basically, TF uses the same farming scheme as DF, but each packet received by a worker is first tested using a predicate `h`. If it passes the test, the `solve` function is applied and the result accumulated to the current result. Otherwise, a `divide` function is used to recursively generate new packets.

The signature of the TF higher order function is

```
val tf : ('a -> bool)
         (* Type of the predicate function *)
      -> ('a -> 'c)
         (* Type of the solve function *)
      -> ('a -> 'a list)
         (* Type of the divide function *)
      -> ('b -> 'c -> 'b)
         (* Type of the accumulating function *)
      -> 'b
         (* Type of the accumulator value *)
      -> 'a
         (* Type of the input data *)
      -> 'b
         (* Type of the result *)
```

The ITERMEM skeleton

A possible definition in CAML of the ITERMEM skeleton is

```
let itermem inp loop out z x =
  let rec f z =
    let z', y = loop (z, inp x)
    in
    out y ; f z'
  in
  f z
```

This makes uses of two nested local definitions (one for the recursive `f` function, the other for `z'` and `y`, the latter two denoting the outputs of the central looping function `loop`. The explicit notion of iteration[34] is encoded using the CAML sequencing construct: `exp1 ; exp2` means "evaluate exp1, discards the result, then evaluate exp2 and return its result".

The signature of the ITERMEM higher order function is

```
val itermem : ('a -> 'b)
              (* Type of the input function *)
            -> ('c * 'b -> 'c * 'd)
            (* Type of the central loop function *)
            -> ('d -> unit)
            (* Type of the output function *)
            -> 'c
            (* Type of the memory value *)
            -> 'a
            (* Type of the input data *)
            -> unit
            (* Type of the result (nothing) *)
```

Note that the ITERMEM skeleton does not return anything (output is supposed to be handled by its `out` argument. This is evidenced by the `unit` type in CAML.

Appendix 2

We give here the parametric process networks (PPNs) related to the DF, TF and ITERMEM skeletons

The DF skeleton

The PPN template for the DF skeleton is given in Fig. 20.

The pseudo-code for the two involved PPTs are given in Fig. 21

[34] Iteration is implicit with SCM, DF and TF

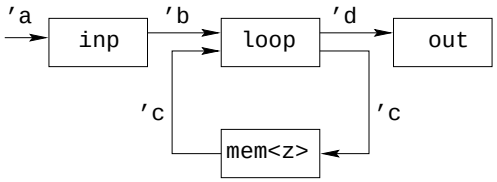


Fig. 22. The parametric process network for the ITERMEM skeleton

```
MEM<init>
  mem := <init>
  While(true) do
    write mem to output slot i
    mem := read input on slot 1
    done
```

Fig. 23. The parametric process template used by the ITERMEM skeleton

The TF skeleton

The PPN and PPTs for the TF skeleton are similar to that of the DF skeleton. The main differences come from conditional execution (implying dynamic thread scheduling) of the `solve` function on the workers and of the `divide` and `accum` functions on the farmer. For this reason, it will not be reproduced here.

The ITERMEM skeleton

The PPN of the ITERMEM skeleton closely follows its representation given in Fig. 22.

This only involves one PPT, the one of the `mem` node (Fig. 23). It memorizes the result of the $n-1$st iteration to make it available at iteration n.

Appendix 3

We give here, just for information purposes, the definition of the `map` and `fold_left` "standard" higher order functions in CAML. These definitions are used within SKIPPER to give a default sequential semantics to the DF and TF skeletons. Let us recall that understanding these definitions is not required of the application programmer, insofar as the declarative semantics of the skeletons can be defined with the "formal" definitions of `map` and `fold_left` given in the text.

```
let rec map f xs = match xs with
    [] -> []
  | x :: xs -> f x :: map f xs

let rec fold_left f z xs = match xs with
   [] -> z
 | x :: xs -> fold_left f (f z x) xs
```

Jocelyn Sérot graduated from IRESTE, Nantes in 1989 and received the PhD degree from the University of Paris-Sud in 1993. He was appointed Assistant Professor at Blaise Pascal University, Clermont-Ferrand, France in 1994 and joined the computer vision group of the LASMEA (Laboratoire des Sciences et Materiaux pour l'Electronique, et d'Automatique) CNRS laboratory. His major research interests are in functional programming, parallel architectures, and computer vision.

Dominique Ginhac received his PhD from the Blaise Pascal University, Clermont-Ferrand, France. Since 2000, he has been Assistant Professor at University of Burgundy, Dijon, France. His research interests include parallel image processing and development of software dedicated to the fast prototyping of vision algorithms on MIMD/DM platforms.

Roland Chapuis obtained his PhD in 1991 from Blaise Pascal University (Clermont-Ferrand, France). He is now assistant-Professor in Electrical Engineering at Blaise Pascal University and Researcher at LASMEA. He is working on real-time pattern recognition applied to outdoor environments. since 1988, he has been working in vision-based road-following algorithms in collaboration with PSA Peugeot-Citroën.

Jean Pierre Dérutin is a professor of micro-electronic design at the Engineer school CUST-Université Blaise Pascal and he is doing his research activities in the LASMEA-UMR 6602 CNRS at the same university. His main research interests are in the field of dedicated parallel machines for image processing, especially with a MIMD-DM approach. The experimental domains focus on applications with hard constraints in terms of real time, volume, and electric power like intelligent vehicles. Current interests are fast prototyping of vision applications on dedicated parallel systems of image processing and new architectures of dedicated MIMD-DM machines for image processing.