Parallel Computing 28 (2002) 1685–1708

PARALLEL COMPUTING

www.elsevier.com/locate/parco

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

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

[a] LASMEA, UMR 6602 CNRS, University Blaise Pascal de Clermont-Ferrand, Campus des Cézeaux, F-63177 Aubière, France

[b] LE2I, FRE 2309 CNRS, University of Burgundy, F-21078 Dijon, France

Received 18 February 2002; received in revised form 22 May 2002; accepted 17 June 2002

Abstract

This paper is a general overview of the SKIPPER project, run at Blaise Pascal University between 1996 and 2002. The main goal of the SKIPPER project was to demonstrate the applicability of *skeleton-based* parallel programming techniques to the fast prototyping of reactive vision applications. This project has produced several versions of a full-fledged integrated parallel programming environment (PPE). These PPEs have been used to implement realistic vision applications, such as road following or vehicle tracking for assisted driving, on embedded parallel platforms embarked on semi-autonomous vehicles. All versions of SKIPPER share a common front-end and repertoire of skeletons—presented in previous papers—but differ in the techniques used for implementing skeletons. This paper focuses on these implementation issues, by making a comparative survey, according to a set of four criteria (efficiency, expressivity, portability, predictability), of these implementation techniques. It also gives an account of the lessons we have learned, both when dealing with these implementation issues and when using the resulting tools for prototyping vision applications.

© 2002 Elsevier Science B.V. All rights reserved.

Keywords: Parallelism; Skeleton; Computer vision; Fast prototyping; Data-flow

1. Introduction

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

* Corresponding author.

E-mail addresses: jocelyn.serot@lasmea.univ-bpclermont.fr (J. Sérot), dginhac@u-bourgogne.fr (D. Ginhac).

0167-8191/02/$ - see front matter © 2002 Elsevier Science B.V. All rights reserved.
PII: S0167-8191(02)00189-8

in remote inspecting robots or vehicles equipped with assisted-driving systems, as presented in [16,25,27]. Although relying on algorithms and programming paradigms encountered in the mainstream of computer vision, these applications raise two specific issues. First, they implement *reactive systems*, operating "on the fly" on digital *streams* of images. This means that they must be able to absorb input data and output results at a minimum frequency and produce responses within a maximal latency. For assisted-driving applications, for instance, the typical frequencies are in the range of 10–30 frame/s and the maximal latency rarely exceeds 50 ms. Second, they must meet stringent operational constraints in terms of volume or power consumption, which often rules out implementations based upon stock-hardware.

These requirements can be met by resorting to embedded parallel machines. The Transvision [15,19] platforms, built between 1992 and 1998 at LASMEA, are examples of this approach. These MIMD architectures, built upon Transputer and Alpha processors, could deliver significant computing power with a limited volume and power-consumption, and provided built-in facilities for video i/o. More recently, we have been investigating the feasibility of a embedded Beowulf-style cluster built upon PowerPC G4 processors and using the IEEE-1394 interface for fast video i/o.

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

The Skipper project was developed in response to the aforementioned problems. Basically, its goal was to "capture"—in a efficient and portable way—the expertise gained by programmers when implementing reactive vision applications using low level parallel constructs, to make it readily available to algorithmicians and image processing specialists. This project has been run at LASMEA from 1996 to now and has produced four skeleton-based parallel programming environments: Skipper-0, Skipper-i, Skipper-ii and Skipper-d. These results have been described in previous papers [9,16,24,25,27] but in a rather separate manner. The goal of this paper is to provide a global presentation of these separate accounts and to provide a comparative assessment of the successive versions of Skipper. It explains in particular why these versions, which share a common formalism for *specifying* parallel programs, differ significantly in the techniques used for *implementing* skeletons. It is organized as follows. Section 2 is a brief recall of Skipper principles and general architecture. Section 3 presents the successive versions of Skipper and the criteria used to assess them. Section 4 summarizes the results of this assessment. Section 5 is a brief review of related work and Section 6 concludes this paper and outlines directions for future work.

2. SKIPPER generic architecture

The Skipper programming methodology is based upon the concept of *algorithmic skeletons* [7,8]. Skeletons are high-level program constructs that abstract common patterns of parallel computation in a parametric way. With this approach, the structure of a parallel application is expressed only as a combination of the skeletons provided. 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. Fig. 1 gives the general software architecture of the Skipper parallel programming environments. The application programmer provides a skeletal, structured description of the parallel program, the set of application-specific sequential functions used to instantiate the skeletons and a description of the target architecture. The Skipper suite of tools turns these descriptions into executable parallel code. The main software components are: a library of skeletons, a compile-time system (CTS) for generating the parallel C code and a run-time system (RTS) providing support for executing this parallel code on the target platform. The CTS can be further decomposed into a front-end, whose goal is to generate a target-independent intermediate representation of the parallel program, and a back-end system, in charge of mapping this intermediate representation onto the target architecture.

2.1. The skeleton library

The Skipper library of skeletons was built "bottom-up", from a careful analysis of a large corpus of existing low-to-mid level vision applications hand-coded in parallel C [27]. It consists of three skeletons: [1]

- the scm (split–compute–merge) skeleton is devoted to fixed data-parallelism, for instance to "geometric" processing of iconic data, in which the input image is split into a fixed number of subimages, each subimage being processed independently, and the final result is obtained by merging the results computed on subimages;
- the df (data-farming) skeleton handles variable data-parallelism, i.e., situations in which the number of data to process is not known at compile time;
- the tf skeleton is a generalisation of the df skeleton, in which the processing of one data item may recursively generate new items to be processed. It is generally used to implement *divide-and-conquer* strategies.

Each skeleton comes with two semantics: a *declarative semantics*, which gives its "meaning" to the application programmer in an implicitly parallel manner, i.e.,

[1] A fourth skeleton (itermem) is described in previous papers. This skeleton does not actually 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—has to be made explicit. It will not be discussed here.

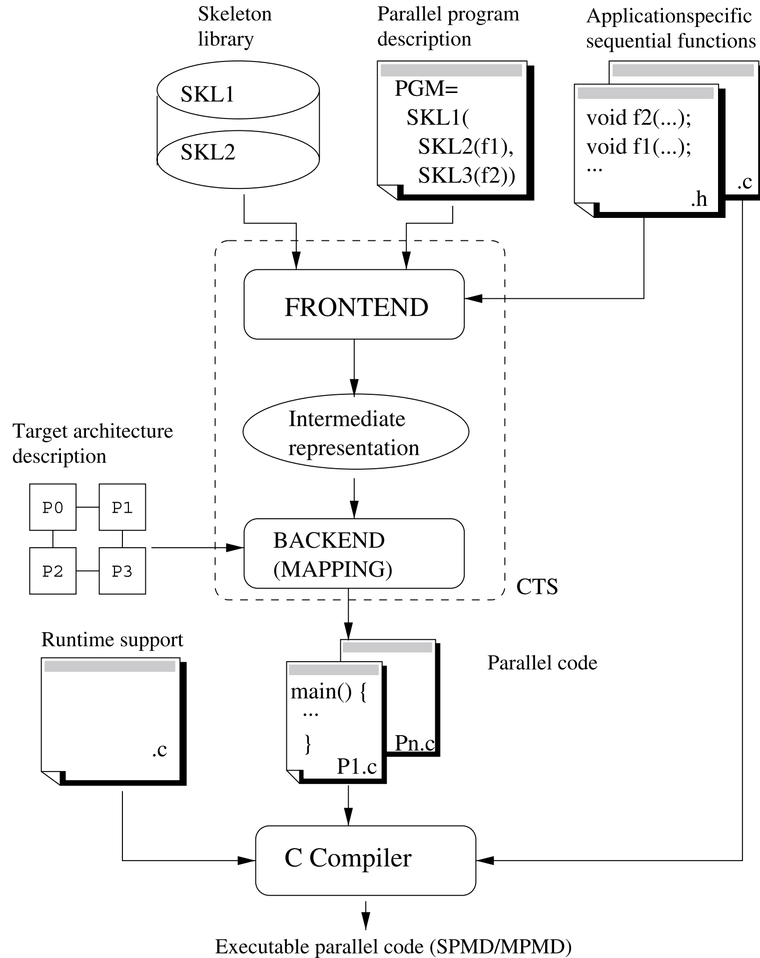


Fig. 1. Skipper general software architecture.

without any reference to an underlying execution model, and an *operational semantics* which provides an explicitly parallel description of the skeleton.

The *declarative semantics* of each skeleton is shared by all Skipperversions. It is conveyed using the Caml language, using higher-order polymorphic functions. The corresponding definitions are given in Fig. 2. Potential (implicit) parallelism arises from the use of the `map` and `foldl1` higher-order functions. [2]

[2] These higher-order functions apply a function and iterate a (commutative, associative) binary operator over a list of elements, respectively, $map\ f\ [x_1, x_2, \ldots, x_n] = [f(x_1), f(x_2), \ldots, f(x_n)]$ `foldl1` $\oplus\ [x_1, x_2, \ldots, x_n] = x_1 \oplus x_2 \oplus \cdots \oplus x_n$.

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

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

let rec tf triv solve divide comb xs =
  let f x =
    if (triv x) then solve x
    else tf triv solve divide comb (divide x)
  in foldl1 comb (map f xs)
```

Fig. 2. Declarative semantics of SKIPPER skeletons.

The *operational semantics* of a skeleton varies according to the nature of the intermediate representation used by the CTS. In the successive versions of SKIPPER, we have been experimenting with four types of intermediate representation: static data-flow graphs (for SKIPPER-O), parametric process networks (for SKIPPER-I), hierarchical task graphs (for SKIPPER-II) and tagged-token data-flow graphs (for SKIPPER-D). These representations will be discussed in turn in Sections 3.1–3.4.

SKIPPER also relies on the CAML language for *expressing the parallel (skeletal) structure of the programs*. The programmer indicates which skeletons are used, in what order and, for each skeleton, the sequential functions and/or numeric values given as parameters. [3] This is illustrated in Fig. 3, with a small program making use of the SCM skeletons to process an image. Here `get`, `splitrow`, `filt`, `conv`, `mergerow` and `disp` are the application specific, sequential functions: `splitrow` decomposes an image into horizontal sub-images, `filt` and `conv` respectively apply a median filter and a convolution mask on a (sub)image and `mergerow` concatenates subimages into a single one. The `get` function retrieves the next image from the video input stream and the `disp` function displays the result image on the screen. $\circ$ is the CAML infix operator denoting function composition.

In the previous example, the *application-specific sequential functions* are written in C. This point is of great practical importance since we do not want application programmers to recode their algorithms from scratch (and especially in CAML). The prototype of the functions used in the previous example are given in Fig. 4.

The role of the *back-end* in the CTS is to map the intermediate representation of the parallel program (data-flow graph, process network, etc.) onto the target architecture. For an MIMD target with distributed memory, for example, this involves finding a distribution of the operations/processes on the processors and a scheduling of the communications on the provided medium (bus, point-to-point links, etc.). The distribution and the scheduling can be *static*—i.e., done at compile time—or *dynamic*—i.e., postponed until run-time. Both approaches require some kind of

[3] The need to pass and return functions and values from various types to/from other functions explains the choice of a higher-order, polymorphic language, such as CAML, for specifying skeletons and skeletal programs in SKIPPER.

```
let img = get ();;
let res = scm splitrow (conv o filt) mergerow img;;
let main = disp res;;
```

Fig. 3. A sample skeletal program.

```
void get(/*out*/ img *im);
void splitrow(/*in*/ img *im, /*out*/ imgList *ims);
void filt(/*in*/ img *im1, /*out*/ img *im2);
void conv(/*in*/ img *im1, /*out*/ img *im2);
void mergerow(/*in*/ imgList *ims, /*out*/ img *im);
void disp(/*in*/ img *im);
```

Fig. 4. Prototype of sequential functions.

RTS. For static approaches, the RTS can take the form of a reduced set of *primitives*, providing mechanisms for synchronizing threads of computations and exchanging messages between processors. [4] For dynamic approaches, it must include more sophisticated mechanisms for scheduling threads and/or process and dynamically managing communication buffers, etc. For this reason, static approaches generally lead to better (and more predictable) performances. But, as evidenced in Section 3.1, they may lack expressivity. Dynamic approaches, on the other hand, do not suffer from this limitation but this is generally obtained at the expense of reduced performances and predictability (as evidenced in Sections 3.3 and 3.4). The Skipper project has covered a wide spectrum of distribution and scheduling techniques, ranging from entirely static to fully dynamic, making it possible to assess the relative merits and flaws of these techniques in the context of a skeleton-based methodology.

Depending on the distribution and scheduling technique used in the back-end, the *parallel code* takes the form of a set of either MPMD (one distinct program per processor) or SPMD (the same program for all processors) programs. These programs are linked with the code of the RTS and the definition of the application specific sequential functions to produce the executable parallel code.

3. Comparative assessment

All versions of Skipper share the general architecture described in the previous section. They differ in the type of *intermediate representation* produced by the front-end and in the distribution/scheduling technique used by the back-end. The

[4] These primitives can use architecture-specific instructions or portable OS-level facilities such as MPI for example.

consequences of these implementation choices will be analysed in turn in the following sections according to four criteria:

Efficiency. Efficiency will be assessed either by observing the obtained speedups on realistic or synthetic [5] test applications or by comparing the run-time performances of the "skeletized" application to those obtained with a hand-crafted parallel version using C+MPI. [6]

Portability. Here, we mean the ability to port a given version of the SKIPPER suite of tools onto a new parallel platform. Given the layered software architecture of SKIPPER, these portability issues mainly concern the RTS: the smaller (and the simpler) this RTS, the more portable the corresponding SKIPPER version will be. Moreover, in our context (embedded vision applications), we must eventually consider the possibility of targeting architectures with *little or no OS-level support*, [7] such as machines built from specialized or digital signal processors (DSPs).

Performance predictability. This refers to the possibility to predict the run-time behavior of an application (its latency and frequency for example) without actually running it on the target parallel platform, on the basis of application-specific parameters (such as the duration of the sequential functions) and architecture-specific parameters (such as communication latency). Performance prediction is generally carried out using analytical *cost models* and estimated (typical) durations (as in SKIPPER-I or in most existing skeleton-based PPEs). In the context of *reactive* applications, one may need a more *deterministic* approach, in which *strict* temporal bounds can be computed at compile-time.

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

3.1. Static data-flow—SKIPPER-0

The first version of SKIPPER used an intermediate representation of skeletal programs as static data-flow graphs (DFG). Skeletons were viewed as parameterisable data-flow *graph patterns*, encoded directly in CAML as higher-order functions thanks to a tool called CAMLFLOW. An in-depth description of CAMLFLOW (which is based upon *abstract interpretation*) can be found in [26]. The mapping of the DFG onto the target architecture was handled by a third party software called SYNDEX [17]. Both

[5] By "synthetic", we mean an application written solely for benchmarking purposes, as opposed to an application corresponding to a "real" algorithm. Synthetic applications allow an easier adjustment of key parameters such as ratio between communication and computation costs or data distribution.

[6] The first versions of SKIPPER were developed for the TRANSVISION platforms, for which we did not have a MPI layer at our disposal.

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

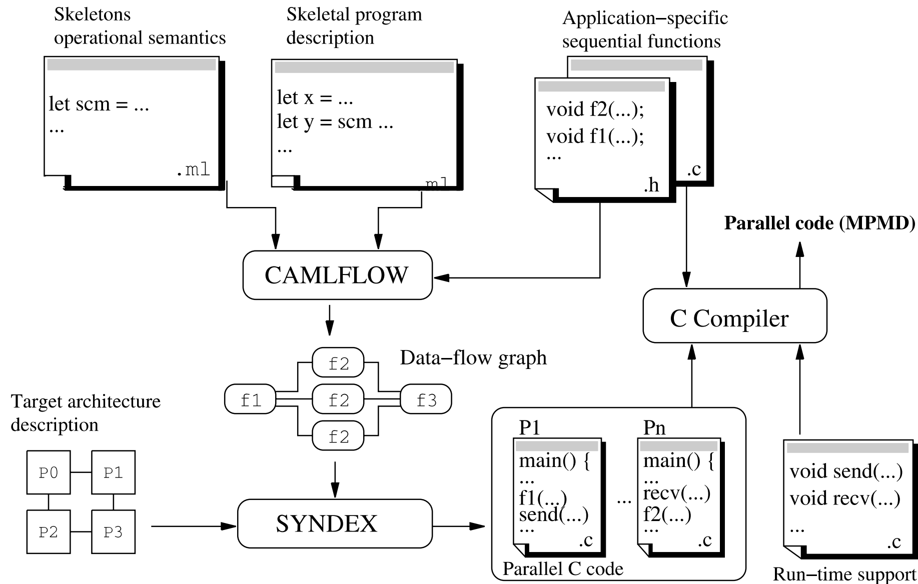


Fig. 5. Compilation path in Skipper-o.

the distribution of the operations (the sequential functions associated with nodes) onto the processors and the scheduling of communications onto inter-processor channels were *static*. The result of Syndex distribution and scheduling is a set of processor-independent programs, [8] one per processor, built from a small kernel of primitives. These primitives offers support for static thread creation, thread communication and call of user-supplied sequential functions. The final parallel C code was obtained by simply providing definitions for these kernel primitives according to the available hardware facilities. [9] The complete compilation path for Skipper-o is illustrated in Fig. 5.

The Skipper-o environment is further illustrated in Fig. 6, which shows the Syndex session used for implementing the program given in Fig. 3. The left window shows the corresponding data-flow graph along with the target architecture (four ring-interconnected C40 processors here). The right window illustrates the static mapping of operations onto processors computed by Syndex (oval boxes represent operations, diagonal lines communications and columns processors).

Assessment. With Skipper-o, the overhead of the run-time system was virtually zero, since all decisions regarding distribution and scheduling were taken at compile-time. This resulted, at least for programs relying on mid and coarse grain fixed data-

[8] *m4* macro-code.

[9] For the Transvision platforms, for example, the primitives used the built-in process switching and channel i/o of the *Transputer*. But the Kernel can be easily ported to other systems, for instance Unix/Linux-based multi-processors communicating trough TCP/IP sockets or MPI.

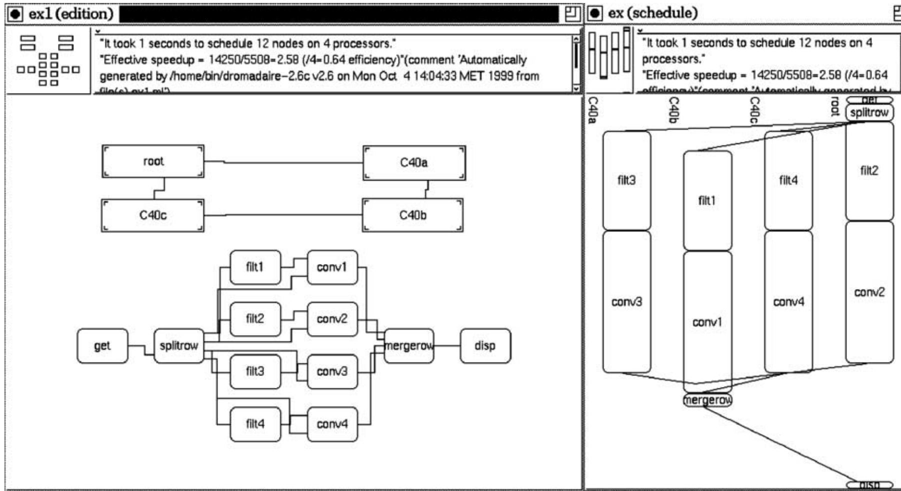


Fig. 6. A SYNDEX session in SKIPPER-O.

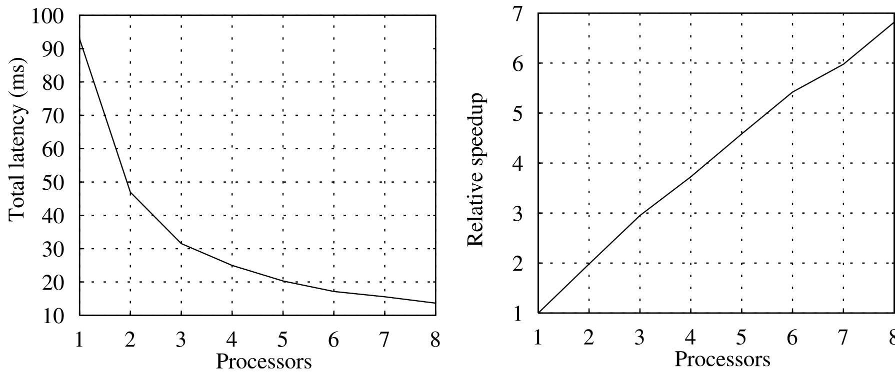


Fig. 7. SKIPPER-O performance figures (**scm** skeleton).

parallelism, in high efficiency. This is evidenced in Fig. 7 which gives the total latency and relative speedup for an application computing histograms of gray level images and implemented on a multi-transputer machine. [10]

For the same reason, predictability was very good, with measured performances never differing from those predicted by more than 5%. Performance prediction in SKIPPER-O actually required two passes: in the first pass, rough estimates of the durations of the sequential functions were given to SYNDEX, which generated a first, sub-optimal, parallel program but with automatic profiling instructions inserted in

[10] T800, 25 MHz processors, 10 Mb/s point-to-point links. At the time SKIPPER-O was designed, this platform was the only one available.

it. This program could then be run on typical data to extract the real durations. These durations were used in turn to obtain the final program by means of a mapping and scheduling heuristic based upon minimization of the total latency. One could also use upper bounds for function durations in order to predict worst case behavior, in order to satisfy hard real time constraints for instance. [11] Finally, portability was also good: because the output macro-code was built on a small set of kernel primitives, re-targeting an application on an architecture built from a new processor type only required (re)writing this set of kernel primitives. This proved to be a straightforward task for the platform we had to deal with. [12]

The main problem with SKIPPER-O was expressivity. Indeed, giving an operational semantics to the DF and TF skeletons in terms of static DFG was problematic. Consider the DF skeleton, for instance. This skeleton is used to apply a function to a list of data items when the size of the list is unknown and/or the time to process one item can vary significantly. [13] In this case, a static allocation of the operations (items) to processors is not always possible and would result, anyway, in an uneven work-load between processors (which in turn results in a poor efficiency). The classical solution is therefore to give the operational semantics of the DF skeleton as a process network and to rely on a *farming* protocol to ensure load-balancing: a *master* process dynamically doles out items to a pool of *worker* processes and collects results back, on a "first done, first served" basis. This model, however, cannot be implemented using a static mechanism, in which all communications must be scheduled at compile-time.

3.2. Template-based implementation—SKIPPER-1

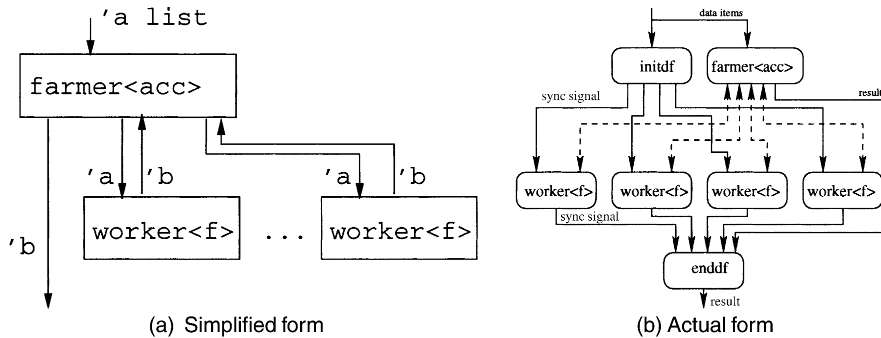
In SKIPPER-I, the limitations of SKIPPER-O were overcome by relying on *process networks* for the intermediate representation of skeletal programs and on *implementation templates* for skeletons. This approach is the most widely used for existing skeleton-based PPEs (like those cited in Section 5). Implementation templates are "*known parametric parallel process networks that efficiently implement a skeleton on a particular parallel target architecture at hand*" [13]. They generally take the form of process graphs that can be parameterized in the parallelism degree (the number of *worker* nodes for instance) and the sequential function(s) associated with each node. The intermediate representation of the application as a process network is then obtained by *instantiating* the skeleton templates. [14] The most often claimed advantage of template-based approaches is that, being written once and for all for a given

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

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

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

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



(a) Simplified form (b) Actual form

Fig. 8. The parametric process network of the DF skeleton.

architecture, they can be carefully hand-crafted to make them both reliable and highly efficient.

The CAMLFLOW front-end of SKIPPER was therefore modified to produce process networks out of CAML skeletal descriptions instead of data-flow graphs. For this purpose, each skeleton was described (in CAML, again) as a *parametric process network*. [15] Fig. 8a gives a parametric process network (PPN) for the DF skeleton. [16] This graph is parametric in the number of `worker` nodes, in the type of data items exchanged between nodes (denoted with type variables `'a`...`'b`) and in the sequential functions run on the nodes `farmer` and `worker` (this "parameterization" being denoted with brackets).

The behavior of the `farmer` and `worker` processes was stored separately as a *parametric process template* (PPT). A PPT is a piece of sequential code whose behavior can be specialized by providing numeric parameters, data types and/or functional parameters. [17]

The compilation path in SKIPPER-I was similar to the one depicted in Fig. 5 for SKIPPER-O, except that the CAMLFLOW front-end produced an intermediate representation in the form of a *process network* instead of a data-flow graph. [18] The back-end tasks were still handled by the SYNDEX software. This may seem contradictory since, as stated in Section 3.1, SYNDEX can only handle static data flow graphs and not process graphs. The solution adopted in SKIPPER-I was in fact a hybrid solution: process graphs were "viewed" by SYNDEX as data-flow graphs and mapped/scheduled as data-flow graphs. In particular, SYNDEX only scheduled (at compile-time) "static" communications (the ones that mark the start and the end of a farming skeleton for instance). The "dynamic" communications (the ones occurring between the master and the workers during the activity of a farming skeleton) were handled by ad hoc processes

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

[16] This graph is a simplified version of the PPN actually in SKIPPER-I, which appears in Fig. 8b (see later).

[17] Specialization is carried out using macro substitution.

[18] A detailed presentation of SKIPPER-I can be found in [27].

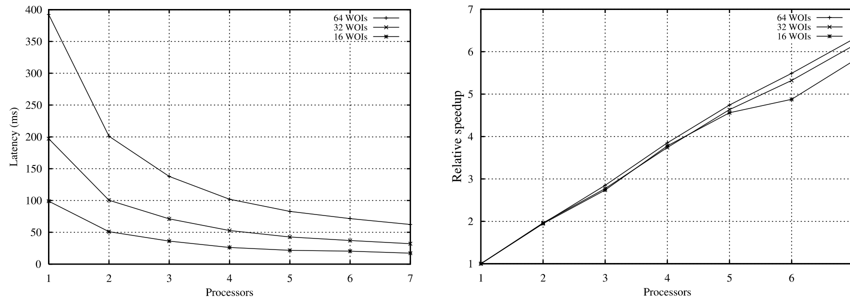


Fig. 9. SKIPPER-I performance figures.

"hidden" in the data-flow nodes. This technique—which amounts to tolerating "critical sections" of dynamically scheduled code within a globally statically scheduled application—is set out in detail in [15]. It is illustrated in Fig. 8b, where "static" communications are denoted with plain lines and "dynamic" ones with dashed lines.

Assessment. The SKIPPER-I version was the first to support the complete set of skeletons described in Section 2 and has been used to implement several realistic reactive vision applications, most noticeably those described in [16] (segmentation by connected component labeling), [25] (vehicle tracking) and [27] (road tracking). Thanks to the SYNDEX back-end, efficiency remained high (with an overhead never exceeding 25% compared with hand-written parallel C code for the applications implemented). For applications making use only of "static" skeletons (such as SCM), this overhead was almost zero, as for SKIPPER-O. Fig. 9 gives the measured performances on a synthetic application, which consists in applying a dummy processing function to a list of windows of interests (WOIs) in an image. Tests were performed on a cluster of eight Sun Ultra-5 workstations with a switched Fast Ethernet connection. Total latency and relative speedup are given for three values (16, 32, 64) of *NBW*, the number of WOIs processed in each image (the greater the number, the more the dynamic farming capabilities of the DF skeleton are solicited).

Predictability of performances relied on a set of analytical cost models [15] that provided accuracy in the range of 10–20%. But, unlike SKIPPER-O, strict timing bounds could not always be exhibited: this is clearly the price to pay for accepting dynamically scheduled skeletons such as DF.

The main problem with SKIPPER-I lay in the hybrid nature of the intermediate representation. Because dynamic communications were transparent to SYNDEX, the routing of these communications between distant processors had to be handled explicitly by auxiliary processes (whereas it is done automatically by SYNDEX for static communications). It turned out that including the description of these auxiliary processes in the SYNDEX kernel without compromising too much efficiency was a difficult task. To make the problem tractable, the SKIPPER-I compilation process therefore made assumptions on the topology of the target architecture (it had to be ring-interconnected). These assumptions, along with the increased size and complexity of the SYNDEX kernel, lowered the portability of the applications developed

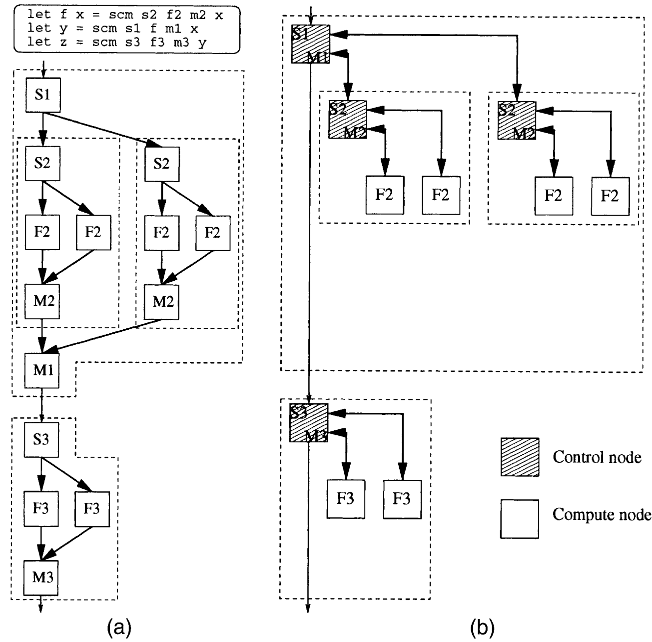


Fig. 10. Intermediate representation of skeletal programs within SKIPPER-II: (a) original program (b) intermediate representation as a tree of TF-II.

with SKIPPER-I (compared to SKIPPER-O). Finally, the hybrid intermediate representation of SKIPPER-I implicitly relied on a "flat" execution model and was definitely not suited for implementing nested skeletons.

3.3. Hierarchical task graphs—SKIPPER-2

In SKIPPER-II, we turned to a homogeneous intermediate representation of programs as *hierarchical task graphs*. This design choice was made in order to overcome the difficulties raised by hybrid representations (such as that used in SKIPPER-I) and to solve the problem of skeleton nesting in a systematic way. To do so, all skeletons of the SKIPPER repertoire were viewed, *at the implementation level*, as specialized *instances* of a generic skeleton, called TF-II. [19] The operational semantics of the TF-II skeleton is basically the one of a task farming skeleton: a *master* process doles out tasks to a pool of *worker* (slave) processes, but here a task can be either a sequential function to be computed or another skeleton to be run. The intermediate representation takes the form of a tree of TF-II skeletons. It is computed by another version of the CAMLFLOW front-end, which uses alternate definitions of the SCM, DF and TF skeletons as specialized calls to the TF-II higher-order function. This step is illustrated in Fig. 10 where a program making using of three SCM skeletons (two of them

[19] For Task Farming, version II.

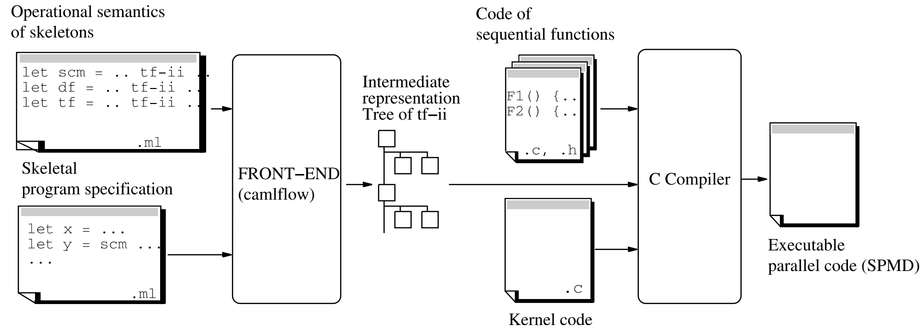


Fig. 11. Compilation path in the SKIPPER-II parallel programming environment.

nested) is turned into a tree of TF-II descriptors. In this tree, nodes correspond to skeleton control processes and leaves to sequential functions (a detailed presentation of the SKIPPER-II system can be found in [9] or in the forthcoming [10]).

Interpretation of the intermediate representation within SKIPPER-II is done at run-time by a specialized program (the "kernel") running in SPMD mode on all processors (see Fig. 11). This kernel—written in C—provides dynamic support for three kind of services: concurrent execution of *master* and *worker* processes, inter-process communication (using a subset of MPI-conformant routines) and handling of shared resources such as the worker pool. Whenever a skeleton needs to be run, either as a "top-level" node (on the spine of the TF-II tree) or as a nested instance, a new copy of the kernel is launched on the local processor. This copy acts as the *master* of the skeleton. It uses the free resources (idle processors) to allocates new workers. When all resources are busy, the execution of *worker* processes is sequentialized on the processor running the *master* process.

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

As regards efficiency, several experiments [9,10] have shown that the dynamic process distribution used in SKIPPER-II may entail a significant performance penalty. This has been proved to be true specially for

- applications exhibiting a low compute vs communication ratio (compared to a C+MPI implementation, the SKIPPER-II kernel performs more communications, for exchanging data between inner and outer *masters* in particular);

- applications relying on fine-grain parallelism (because shared resources are handled in a centralized manner in Skipper-ii, each worker allocation requires a pair of communications to a particular processor; this becomes a bottleneck when the grain decreases, i.e., when the number of *worker* processes increases);
- platforms not supporting multi-processing at the processor level (in this situation, some processors may end up running only one *master* process, with a very small load factor, leading to poor global efficiency. [20]).

Furthermore, the fully dynamic approach used in Skipper-ii makes performance prediction very difficult because, in this model, processors can switch from *master* to *worker* behavior depending only on actual input data (there is no "fixed" mapping for dynamic skeletons as in Skipper-i). Even the interpretation of execution profiles, generated by an instrumented version of the kernel, turned out to be far from trivial. This point raises a pragmatic problem within a programming methodology based upon experimental validation of solutions: here, one not only needs to obtain a running prototype quickly, but also to be able to understand why a given prototype exhibits poor run-time performances. [21]

3.4. Dynamic data-flow—SKIPPER-D

The implementation of Skipper-d started in 2000 in response to the problems identified with Skipper-ii version. The design of Skipper-d was inspired by results obtained by Danelutto on the Macro Data-Flow (MDF) execution model for skeletons [12]. This model is very similar to the one used in Skipper-o: skeleton-based parallel programs are compiled down to data-flow graphs, in which nodes correspond to sequential functions ("macro-instructions") and arcs to data dependencies between these functions. But, like Danelutto and unlike Skipper-o, a *dynamic* interpretation mechanism is used for executing these graphs. This mechanism relies on a set of distributed data-flow interpreters, running in SPMD mode on all processors of the target architecture. Skipper-d extends the MDF execution model proposed by Danelutto in order to implement arbitrary nested data or task farm skeletons. For this purpose, the Skipper-d runtime relies on the *tagged-token* data-flow interpretation technique [1,2]. This technique basically allows several concurrent activations of a single sequential node to overlap in time; it associates a unique *tag* with each activation, and each data token also carries a tag that specifies the particular activation to which it belongs. Skeletons involving run-time bounded iterations and/or recursion, and nested in an arbitrary way, can then be represented as cyclic data-flow graphs. This is illustrated in Fig. 12 with the formulation as a tagged-token MDF graph of a program involving two nested df skeletons (in this figure, tags are denoted as superscripts). The MDF graph uses a pair of special nodes called `iter` and `endf`. `Iter` accepts a list of data items and generates distinct result tokens, each

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

[21] By contrast, the profiling facilities offered by Skipper-i (and set out in detail in [27]) were much easier to exploit.

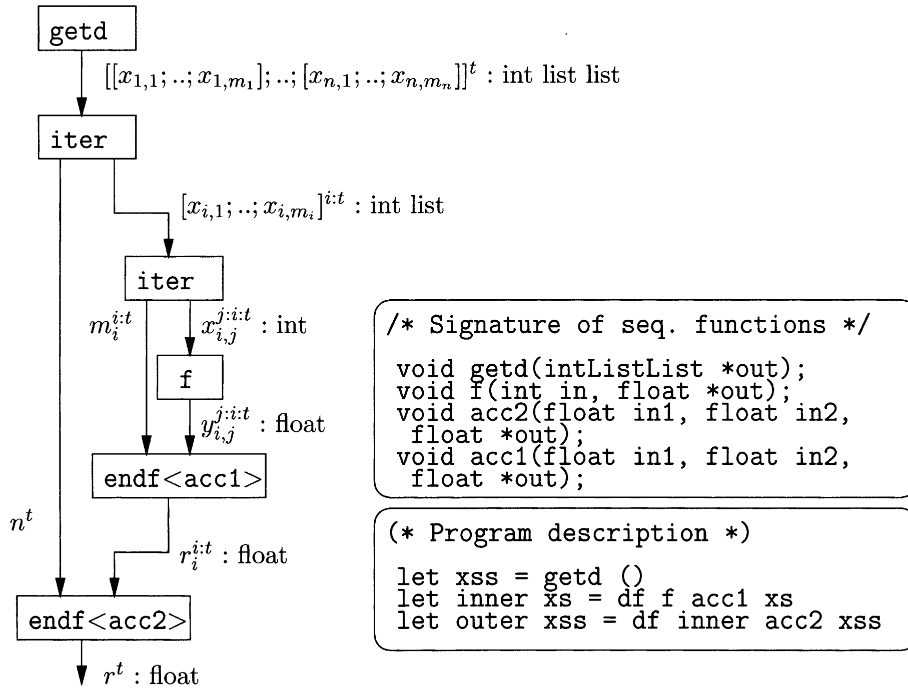


Fig. 12. Nested farm skeletons under the tagged-token MDF model.

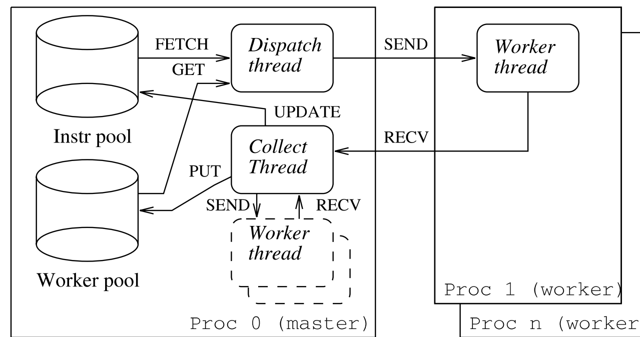


Fig. 13. The run-time system of SKIPPER-D.

carrying one data item and a distinct tag. These tokens trigger distinct firings of the subsequent nodes. The tokens resulting from these firings are collected by the `endf` node and accumulated using the `acc` sequential function. A more detailed account of this mechanism can be found in [24].

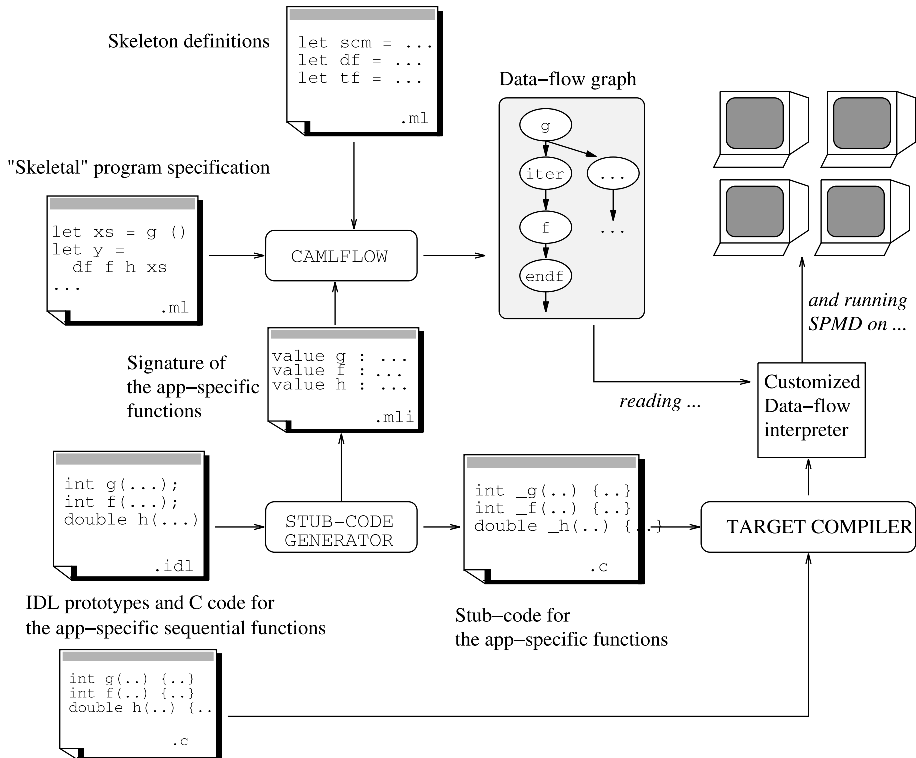


Fig. 14. The compile-time system of SKIPPER-D.

Like its predecessors, the SKIPPER-II system can be divided into a compile-time system and a run-time system. The latter implements a (centralized) tagged-token data-flow interpreter and the former produces the MDF graph for this interpreter from a high-level skeletal program specification.

The *run-time system* of SKIPPER-D is sketched in Fig. 13. Like Danelutto's system, it relies on an SPMD approach: all the processors (nodes) of the target architecture run the same program, which is the result of the compilation of the user code (C sequential functions) and the interpreter code. The interpreter itself involves several *threads* of execution: a `dispatch` thread, which fetches macro-instructions (sequential functions to be computed) from a pool of fireable instructions and sends them to the `worker` threads, a `collect` thread, which receives results from the `worker` threads and updates the instruction pool accordingly and several [22] `worker` threads for computing sequential functions. The `dispatch` thread fetches idle workers from a centralized pool, in which all worker threads register at initialization and which is subsequently updated by the `update` thread upon reception of results.

[22] At least one per processor.

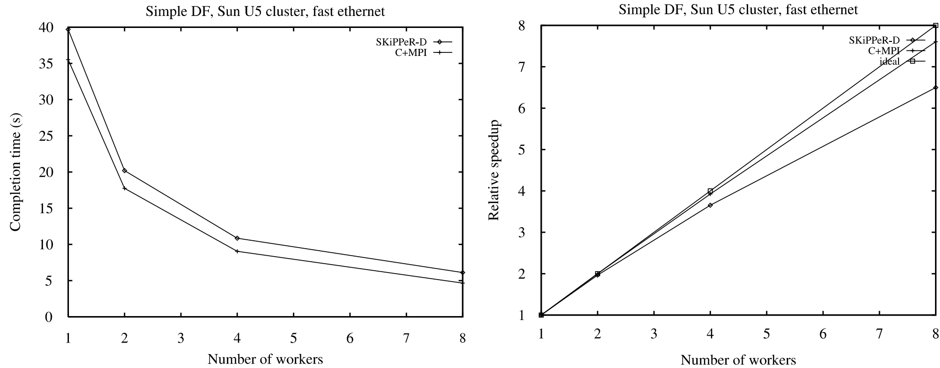


Fig. 15. SKIPPER-D performance figures (DF skeleton).

The *compile-time system* is sketched in Fig. 14. It produces the application-specific data needed to customize the run-time interpreter, i.e., the MDF representation of the program used to build the initial instruction pool and the code of the sequential C functions to be integrated with the custom run-time interpreter. The MDF graph is generated by the CAMLFLOW tool. This offers a way, as in previous versions of SKIPPER, to describe skeletons entirely in CAML as higher-order functions.

Assessment. The main contribution of SKIPPER-D is to provide an all-encompassing intermediate representation for all skeletons. This representation allows arbitrary combination (including nesting) of skeletons, thanks to the tagged-token interpretation mechanism. SKIPPER-D therefore definitely solves the expressivity problem, at least for our repertoire of skeletons. Experimental results, obtained with a prototype run-time system (written in OBJECTIVE CAML) for synthetic applications on a cluster of workstations are reported in [24] and in Fig. 15. They show good speedups and a small overhead (compared with hand-written C+MPI code). Moreover, these experiments have shown that, at least for coarse and medium-grained computation schemes, and contrary to SKIPPER-II, the mechanism used for handling nesting does not entail a significant performance penalty. Together with those reported by Danelutto in [13], these results confirm the merits of dynamic MDF execution models with respect to template-based ones. SKIPPER-D run-time performances could be further improved by integrating some optimization techniques described in [13]. These techniques include a more sophisticated management strategy of the instruction pool (based on high/low water marks), local caching of data on worker nodes and, most noticeably, a distributed interpreter implementation. The current SKIPPER-D implementation relies on a *centralized* data-flow interpreter and a rudimentary scheduling strategy for fireable instructions and is unlikely to provide comparable performances in cases of very irregular fine-grained computations. Performance predictability is clearly harder to obtain than with template-based implementation systems but does not seem an intractable problem (as in SKIPPER-II). The interpretation of profiling results is also easier than with SKIPPER-II, especially if sophisticated visualization tools such as `jumpshot` [28] are provided. The portability of the SKIPPER-D run-

Table 1
Comparative assessment of Skipper versions

	IR	D/S	Eff	Expr	Port	Pred
Skipper-o	SDFG	S/S	4	1	4	4
Skipper-i	PPN	S/S+D	3	2	1	3
Skipper-ii	HTG	D/D	2	4	3	1
Skipper-d	DDFG	D(S)/D	3	4	4	2

time system on architectures built from specialized or digital signal processors is currently limited by the fact that it is written in Objective Caml and uses *bytecode* threads. But the runtime could easily be rewritten in C for these systems. [23] In this case, threads can be emulated using hardware context switching mechanisms (as evidenced by the implementation of the Syndex kernel for DSPs [18]).

4. Comparative assessment

Table 1 summarizes our assessment of the successive versions of Skipper. In this table, we have tried to rate each version in terms of the four criteria explicited in Section 3: efficiency (*Eff*), expressivity (*Expr*), portability (*Port*) and predictability (*Pred*). For this we use a relative "score" between 1 (poor) and 4 (excellent). The first column recalls the underlying intermediate representation (IR): Synchronous Data Flow Graphs, Parametric Process Networks, Hierarchical Task Graphs and Dynamic Data Flow Graphs. The second column gives the distribution and scheduling strategy (S = static, D = dynamic).

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

Fully static approaches, as in Skipper-o, are attractive in our context of embedded reactive applications because they minimize the resources needed to implement the algorithm and allow strict real-time bounds to be computed. But within a programming methodology dedicated to the fast prototyping of solutions—and mainly intended for algorithmicians, not parallel programming specialists—these approaches were finally found to be too restrictive. For instance, it is often possible to reformulate an existing vision algorithm—defined in terms of dynamically allocated data structures as lists or trees—so that it only uses fixed-size arrays and can be parallelized using a "static" skeleton (like scm); but we found that it is not

[23] The current implementation is less than 500 lines of Objective Caml code. We think that a re-implementation in C would be in the range of 1000-2000 loc, perfectly suited for processors with limited memory.

reasonable, even desirable, to do this reformulation at the prototyping level, when being able to quickly test various algorithmic and/or parallel implementation schemes turns out to be more important than obtaining optimal performances. Moreover, some algorithms are intrinsically not amenable to static implementation because the size of the input data and/or the duration of the sequential functions cannot be reliably estimated at compile time.

On the other hand, the conclusions given in Section 3.3 show that approaches relying on a fully dynamic run-time system, as SKIPPER-II, may raise efficiency and predictability or observability problems that conflict with our prototyping goals and/or target platforms (although these approaches might prove useful in other application domains).

In this light, we believe that the SKIPPER-D approach offers the best trade-off between the conflicting above-mentioned criteria. The data-flow interpretation mechanism is "mostly dynamic" [24] but its run-time behavior can be more easily modelized and performances do not suffer from hardly understandable performance drops due to unpredictable process allocation. [25]

5. Related work

In the past decade, the issues related to skeleton-based parallel programming have been investigated by several research groups. [26] But few of them have produced full-fledged software environments that can be used to implement complex, realistic applications.

The Pisa Parallel Programming Language (P3L) project [3] is one of these projects. The P3L system includes both task parallelism (*farm*, *pipe*) and data parallelism (*map*, *reduce*, *scan*). Some control skeletons (*loop*, *seq*) allow the definition of sequential P3L modules and the iteration of skeleton compositions. Like SKIPPER, P3L uses C to express the sequential parts of the application but, unlike SKIPPER, the skeletal structure of the application is denoted using C-like syntax of data types and skeletons. The first compilers generated code for a Transputer-based Meiko CS/1 MIMD machine and for PVM running on a cluster of UNIX workstations. A more recent version [6] generates C+MPI code for PC running Linux and Fujitsu AP1000. P3L has been used to implement applications such as optical character recognition [11], ray tracing and circuit test generation.

The Heriot–Watt group has investigated the use of skeletal-based methodology for the parallelisation of vision algorithms [20,22,23]. Parallelism is extracted and exploited from programs written entirely in Standard ML. Unlike SKIPPER or P3L, in which skeletons are viewed as *explicit* indications to the compiler of which parallelism will be deployed and where—they take an *implicit* approach, in which skeletons

[24] Scheduling is done at run-time but mapping of threads to processors is done at compile-time.

[25] As in SKIPPER-II.

[26] See for example [29] for a comprehensive survey.

are viewed as *possible* realizations of common higher-order functions (the decision is taken by the compiler, on the basis of profiling information collected by an *instrumentation* phase). Results have been given for a Meiko CS, a Fujitsu AP-1000 and a 32-node Beowulf.

The Skil project [4] is another system relying on skeletons to provide high level parallel programming. Skil is an imperative, C-based language enhanced with a series of functional features such as higher-order functions and polymorphism. Compile-time instantiation of these features results in very efficient code (approaching the efficiency of direct C implementations). Skil focuses on data-parallelism and provides built-in types for manipulating distributed data-structures. On numerical applications such as PDE solvers [5] Skil has demonstrated good absolute performances and scalability (24 speedup for 32 processors, 87 on 128 processors) on a 1024-node Parsytec multi-processor.

6. Conclusions and future work

Several lessons were learnt when developing and using the Skipper system, both at the application level (from a user's point of view) and at the implementation level (from an implementor's point of view).

At the application level, the Skipper project has provided a convincing demonstration of the merits of skeleton-based parallel programming techniques. These conclusions are supported by realistic case studies, carried out with the help of full-fledged parallel programming environments, by people who were not parallel programming specialists in the first place. First, the "off-the-shelf" style provided by the skeleton approach effectively provides dramatic savings in development efforts. These savings make it possible to adopt a truly experimental approach in the design and implementation of applications, a key property in our context. The price to pay is a decrease in performance (compared to hand-crafted parallel code) but, for most of the realizations presented here this can be kept reasonable and was viewed as acceptable, anyway, with regard to the above-mentioned benefits. Second, within a given application domain, such as reactive embedded vision, skeletons may be viewed as a very effective way to *encapsulate* and reuse the expertise gained by skilled parallel programmers. This pragmatically solves the classic "completeness" problem often associated with skeleton-based parallel programming methodologies—namely the fact that, in theory, nothing can guarantee that a given set of skeletons will be sufficient to express every parallel algorithm: in our case, the definition of the skeleton basis was made in a *bottom-up* manner starting from an identifiable corpus of applications and/or expert knowledge and was explicitly targeted towards low to mid-level vision algorithms. Finally, the explicit, "menu-driven" approach proposed by Skipper could be criticised for requiring a minimum understanding of the skeleton operational semantics to be used and therefore that it cannot be used as a fully automatic parallelizing tool. Our answer, motivated by our experience in developing complex vision applications with algorithmicians, is that skeletons actually provides an effective *common ground for sharing*

expertise between image processing and parallel programming specialists: the former no longer have to deal with implementation details and the latter can treat application-specific functions as black boxes.

At the implementation level, the SKIPPER project has led us to thoroughfully investigate the relative merits and flaws of *static* and *dynamic* approaches for implementing skeletons. As stated in Section 3.4, we now believe that a macro data-flow representation of skeleton-based parallel programs is probably the best choice, because it can be associated with a wide spectrum of operational semantics (from purely static synchronous to dynamic tagged-token). This conclusion is similar to that drawn by Najjar et al. in [21] who underline the "universality" of the data-flow model by exhibiting potential application domains both in the "software" domain (parallel programming on clusters of workstations for instance) and in the "hardware" domain (design of application-specific circuits for instance). In this context, we are now investigating the possibility of developing transformational rules to derive a static formulation of an algorithm automatically (using a *synchronous* data-flow execution model) from a dynamic one (based upon a *tagged-token* execution model). Our ultimate goal, motivated by our experience and needs in reactive vision applications, is to be able to specify, with the *same* skeletal formalism both "hard" (time-critical) parallel applications (built from static skeletons such as SCM) and "softer" applications (built from dynamic skeletons such as DF) which can tolerate the run-time unpredictability implied by interpreter-based implementation techniques. Recent work on *graph factorization* techniques [14] has provided some insights on how to do this in the context of compile-time bounded iterations. We are currently working to extend this scheme to generic data and task farming skeletons (the fundamental issue being: what constraints do we have to put on the tagged-token data-flow graph formulation of an algorithm—that can always be interpreted dynamically—to make it amenable to static implementation).

References

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

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

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

[4] G.H. Botorog, H. Kuchen, SKIL: an imperative language with algorithmic skeletons for efficient distributed programming, in: International Symposium on High Performance Distributed Computing, IEEE Computer, pp 243–252.

[5] G.H. Botorog, High-level parallel programming and the efficient implementation of numerical algorithms. Ph.D. Thesis, RWTH-Aachen, 1998.

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

[7] M. Cole, Algorithmic Skeletons: Structured Management of Parallel Computation, MIT Press, Cambridge, MA, 1989.
[8] M. Cole, Algorithmic skeletons, in: G.J. Michaelson, K. Hammond (Eds.), Research Directions in Parallel Functional Programming, Springer-Verlag, Berlin, 1999.
[9] R. Coudarcher, J. Sérot, J.P. Dérutin, Implementation of a Skeleton-based Parallel Programming Environment Supporting Arbitrary Nesting. in: Proceedings of the 6th International Workshop on High-Level Parallel Programming Models and Supportive Environments, San Francisco, Lecture Notes in Computer Science, vol. 2026, Springer, Berlin, April 2001, pp. 71–85.
[10] R. Coudarcher, Composition de squelettes algorithmiques: application au prototypage rapide d'applications de vision. Ph.D. Thesis, Université Blaise Pascal Clermont-Ferrand (France), 2002, in press.
[11] M. Danelutto, S. Pelagatti, R. Ravazzolon, A. Riaudo, Parallel OCR in P3L: a case study, in: High Performance Computing and Networking, vol 1067, Lecture Notes in Computer Science, Springer, Berlin, 1996, pp. 1017–1019.
[12] M. Danelutto, Dynamic run time support for skeletons, in: Proceedings of the ParCo99 Conference, Delft, The Netherlands, August 1999.
[13] M. Danelutto, Efficient run-time support for skeletons on workstation clusters, Parallel Processing Letters 11 (1) (2001) 41–56.
[14] A. Dias, C. Lavarenne, M. Akil, Y. Sorel, Optimized implementation of real-time image processing algorithms on field programmable gate arrays, in: ICSP'98 Fourth International Conference on Signal Processing, Beijing, China, 1998.
[15] D. Ginhac, Prototypage rapide d'applications parallèles de vision artificielle par squelettes fonctionnels. Ph.D. Thesis, Université Blaise Pascal Clermont-Ferrand (France), 1999.
[16] D. Ginhac, J. Sérot, J. Dérutin, Fast prototyping of image processing applications using functional skeletons on an MIMD-DM architecture, in: IAPR Workshop on Machine Vision and Applications, Chiba, Japan, 1998, pp. 468–471.
[17] T. Grandpierre, C. Lavarenne, Y. Sorel, Optimized rapid prototyping for real time embedded heterogeneous multiprocessors, in: Proceedings of 7th International Workshop on Hardware/Software Co-Design, Rome, 1999.
[18] C. Lavarenne, Y. Sorel, Modèle d'exécutif distribué temps-réel pour SynDEx INRIA Research Report, RR-3476, 1998.
[19] P. Legrand, R. Canals, J.P. Dérutin, Edge and region segmentation processes on the parallel vision machine Transvision, in: Computer Architecture for Machine Perception, New-Orleans, USA, 1993, pp. 410–420.
[20] G.J. Michaelson, N.R. Scaife, Prototyping a parallel vision system in standard ML, Journal of Functional Programming 5 (3) (1995) 345–382.
[21] W.A. Najjar, E.A. Lee, G.R. Gao, Advances in the dataflow computational model, Parallel Computing (25) (1999) 1907–1929.
[22] N. Scaife, P. Bristow, G. Michaelson, P. King. Engineering a parallel compiler for SML. Proceedings of the 10th International Workshop on Implementation of Functional Languages, September, 1998, pp. 213–226.
[23] N. Scaife, A dual source parallel architecture for computer vision Ph.D. Thesis, U. Heriot-Watt, Edinburgh, 2000.
[24] J. Sérot, Tagged-token data-flow for skeletons, Parallel Processing Letters 11 (4) (2001).
[25] J. Sérot, D. Ginhac, J. Dérutin. Skipper: a skeleton-based parallel programming environment for real-time image processing applications, in: Proceedings of 5th International Conference on Parallel Computing Technologies, 6–10 September, 1999, Lecture Notes in Computer Science, vol. 1662, Springer, Berlin, 1999, pp. 296–305.
[26] J. Sérot, CamlFlow: a Caml to data-flow graph translator, in: S. Gilmore (Ed.), Trends in Functional Programming, Intellect, vol 2, 2001.
[27] J. Sérot, D. Ginhac, R. Chapuis, J. Dérutin, Fast prototyping of parallel vision applications using functional skeletons, Journal of Machine Vision and Applications 12 (6) (2001) 271–290.

[28] O. Zaki, E. Lusk, W. Gropp, D. Swider, Toward scalable performance visualization with Jumpshot, High Performance Computing Applications 13 (2) (1999).
[29] Online bibliography available at http://hypatia.dcs.qmw.ac.uk/SEL-HPC/Articles/SkeletonArchive.html.