SKiPPER: A Skeleton-Based Parallel Programming Environment for Real-Time Image Processing Applications

Jocelyn SÉROT, Dominique GINHAC, Jean-Pierre DÉRUTIN

LASMEA UMR 6602-CNRS
Campus des Cézeaux, F-63177 Aubière Cedex, France
E-mail: Jocelyn.Serot@lasmea.univ-bpclermont.fr
Tel : +33 (0)4 73 40 73 30 Fax : +33 (0)4 73 40 72 62

Abstract. This paper presents SKiPPER, a programming environment dedicated to the fast prototyping of parallel vision algorithms on MIMD-DM platforms. SKiPPER 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. Each skeleton is given an architecture-independent functional (but executable) specification and a portable implementation as a generic process template. The source program is a purely functional specification of the algorithm in which all parallelism is made explicit by means of composing instances of selected skeletons, each instance taking as parameters the application specific sequential functions written in C. SKiPPER compiles this specification down to a process graph in which nodes correspond to sequential functions and/or skeleton control processes and edges to communications. This graph is then mapped onto the target topology using a third-party CAD software (SynDEx). The result is a dead-lock free, optimized (but still portable) distributed executive, which SKiPPER finally turns into executable code for the target platform. The initial specification, written in ML language, can also be executed on any sequential platform to check the correctness of the parallel algorithm. The applicability of SKiPPER concepts and tools has been demonstrated by parallelising several realistic real-time vision applications both on a multi-DSP platform and a network of workstations. It is here illustrated with a real-time vehicle detection and tracking application.
Keywords: *Parallelism, skeleton, Caml, image processing, fast prototyping, vehicle tracking*

1 Introduction

In recent years, there has been a growing interest in so-called *skeleton-based* parallel programming models [1] [11] in which the programmer's task is to select and compose instances of pre-defined *templates*, chosen from a fixed repertoire, rather than to deal with low-level parallel constructs such as message-passing calls or shared-memory access. The idea is that recurring patterns of parallel computations can be encapsulated into higher-order program constructs which can

V. Malyshkin (Ed.): PaCT-99, LNCS 1662, pp. 296-305, 1999.

be parameterized to suit a given parallel application, thus hiding all low-level, error-prone implementation details to the application programmer. Skeleton-based programming models are simple, abstract and make it possible to conciliate *portability* and *efficiency*: skeletons can be defined in a target-independent manner but their implementation on a given platform — being done once — can be carefully handcrafted [12]. However, their applicability to *general-purpose* parallel programming remains an open question, because it seems very difficult (impossible ?) to exhibit a fully generic repertoire of skeletons, *i.e* one sufficient to express *every* parallel algorithm. This limitation does not hold if the class of encompassed algorithms is deliberately restricted to a given application domain. In this case, the definition of the skeleton repertoire can be made in a bottom-up manner, starting from an identifiable corpus of applications and/or expert knowledge. This paper assesses this approach by taking our primary application domain as target, namely real-time image processing. In this context, we have found skeletons to be a very effective programming paradigm for *encapsulating* the expertise gradually gained by parallel programmers and making it readily available for the rapid prototyping of subsequent applications.

The paper is organized as follows: section 2 briefly recalls the most salient features of the parallel skeleton concept and presents a repertoire of such skeletons specifically dedicated to real-time image processing applications. Section 3 presents SKiPPER, a complete parallel programming environment built on this skeleton basis. Section 4 demonstrates the effectiveness of the presented concepts and tools, both in terms of code performance and programmability, through a realistic case study. Section 5 is a brief survey of related work. Section 6 concludes this paper by summerizing the main results of this work and giving hints for further investigations.

2 Skeletons for parallel image processing

Within our application domain — low and intermediate level image processing — a retrospective analysis of legacy implementations on MIMD-DM platforms (especially the TRANSVISION [8] platforms, for which we had a large corpus of working, hand-coded parallel applications) showed that most of parallel applications were actually built upon a limited number of recurring *patterns*. Three broad classes of patterns could readily be identified:

- Patterns devoted to “geometric” processing of iconic data. These are all instances of an elementary form of *data parallelism* in which the input image is decomposed into sub-domains, each sub-domain is processed independently with the same function, and the final result is obtained by merging those computed on each sub-domain.
- Patterns encapsulating generic parallel control structures such as *data farms* or *task farms*. These typically involve processing lists of features when the size of the list and/or its elements depends on the input data and thus requires some form of dynamic load-balancing to achieve good efficiency.

- Patterns reflecting the iterative nature of the vision algorithms, *i.e.* the fact that an embedded vision system does not process single images but continuous *streams* of images.

From the implementation point of view, each pattern can be be viewed as a fixed, generic communication harness embedding a set of application-specific sequential functions. Following the skeleton approach, it will therefore be abstracted into a reusable parallel construct, *i.e.* a higher-order function encapsulating all its parallel behaviour and accepting as parameters the sequential functions. This led to the following four "elementary" skeletons making the basis of our programming environment:

- The SCM skeleton (Split, Compute and Merge) encompasses the first class of patterns, *i.e.* those dedicated to regular, data-parallel processing. The SCM skeleton has been illustrated for example in [7].
- The DF (Data Farming) skeleton is an abstraction of the processor farm model, devoted to irregular data-parallelism. Its implementation relies on a *master* process dynamically dispatching data packets to a pool of *worker* processes and accumulating partial results until each input data is processed.
- The TF (Task Farming) skeleton is a generalisation of the DF one, in which each worker can recursively generates new packets to be processed. Its main use is for implementing the so-called *divide-and-conquer* algorithms. It will not be discussed here.
- The ITERMEM skeletons is used whenever the stream-based model of computation has to be made explicit, in particular when computations on the n^{th} image depends on results computed on the $n+1^{th}$. Such "looping" patterns are very common in tracking algorithms, based upon system-state prediction, such as the one presented in section 4.

Practically, each skeleton is given *two* definitions: a *declarative* one and an *operational* one.

The goal of the *declarative* definition, which is written once, is to give the skeleton an architecture-independent, purely *applicative* interpretation. Because of its higher-orderness, this definition is classically and elegantly written using a functional language. For example, here's a declarative definition of the DF skeleton in CAML, a well-known dialect of the ML functional language[2]:

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

This definition states the skeleton semantics as a simple combination of calls to its functional arguments[1]. Here, it says that the result of applying `(df n comp acc z)` (*i.e* the parameterized skeleton) to a list `xs` is obtained by first applying the `comp` function to each element of the list[2] and then accumulating all the resulting values[3]. Note that the first argument (`n`), actually related to the oper-

[1] In (CA)ML function application associates to the right and is denoted without parenthesis, so that `f a b` reads `(f a)(b)` or, more simply, `f(a,b)`

[2] `map` is the CAML builtin higher-order function defined by `map f [x1;x2;...xn] = [f x1;f x2; ... f xn]` where `[a;b;...]` is the CAML notation for lists

[3] `fold_left` is the CAML builtin higher-order function defined by `fold_left f z [x1;x2;...;xn] = f ( ... (f (f z x1) x2) ... xn)`

ational definition, is not used here. The CAML definition classically comes with a *type signature*, the goal of which is to express all the generic type constraints that the arguments of the df skeleton will have to meet in order to build *consistent* programs. Here's the signature for the DF skeleton:

```
val df : int                 (* Number of workers *)
       -> ('a -> 'b )        (* Compute function *)
       -> ('c -> 'b -> 'c) (* Accumulating (folding) function *)
       -> 'c                 (* Initial accumulator value *)
       -> 'a list            (* Input list *)
       -> 'c                 (* Result *)
```

Type *variables* (denoted by letters 'a,...,'c) introduce *polymorphism*, *i.e* the ability for the skeleton to accommodate arguments with various (but related) types. For example, if the second argument (comp) of DF has type 'a->'b (*i.e.* function from any type 'a to any type 'b) then its fifth argument (xs) must have type 'a list (*i.e* list of 'a), its fourth argument (z) type 'c and its third argument (acc) type 'c->'b->'c (*i.e.* function from types 'c and 'b to type 'c).

Being real CAML code, the applicative definition can be viewed as an *executable* specification, which can be used to assign a target independent semantics to skeleton-based programs. Practically, this gives the programmer the opportunity to *sequentially emulate* a parallel program on "traditional" stock hardware before trying it out on a dedicated parallel target (by supplying relevant input data, observing results and, if a problem arises, using *sequential* debugging tools to overcome it, for example).

The goal of the *operational* definition is to make explicit the parallel behaviour of the skeleton by specifying its actual implementation on a given platform. For this a classical representation of skeletons as *process network templates* (PNTs) is used. PNTs are incomplete graph descriptions, which are parametric in the degree of parallelism (for example, in the number of comp nodes for the DF skeleton), in the sequential function computed by some of their nodes and in the data types attached to their edges. Figure 1 is a representation of a PNT for the DF skeleton on a ring-connected architecture.

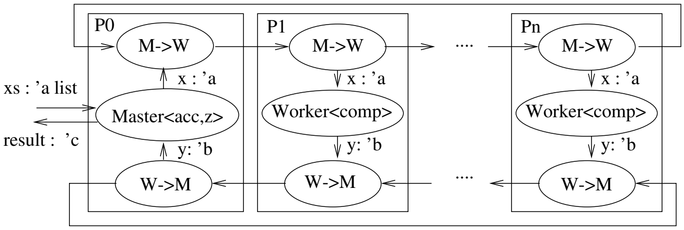


Fig. 1. A process network template for the DF skeleton

Rectangular boxes represent processors (numbered 0 to n), ellipsis sequential processes and arrows communications. Four types of processes are involved: `Master` for dispatching data items and accumulating results, `Worker` for applying the `comp` function and two auxiliary processes (`W->M` and `M->W`) for routing data

The operational definition must be written for each target architecture. It is of course the *implementor*'s responsibility to prove its equivalence with the declarative one (*i.e.* the compatibility of the sequential and parallel semantics). For the DF skeleton, for example, this requires that the `acc` function is commutative and associative, since the accumulation order in the parallel case is intrinsically unpredictable.

3 The software environment

The components of SKiPPER programming environment are depicted in figure 2.

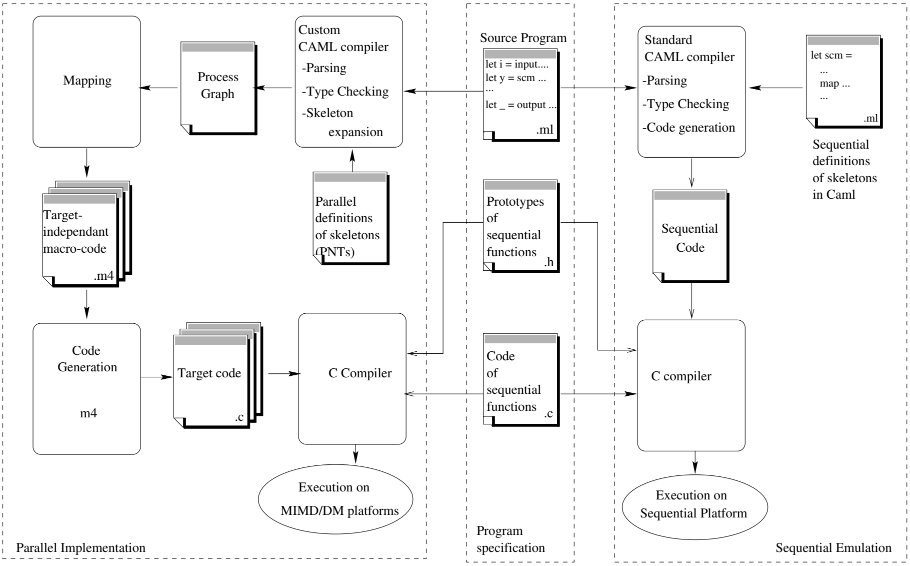


Fig. 2. The skeleton-based programming environment

The source program is a functional specification of the algorithm, in which all parallelism is explicited by means of composing instances of the aforementioned skeletons. Each instance takes as parameters application-specific sequential functions written in C.

SKiPPER starts from this specification for deriving both a parallel implementation on target hardware and an sequential emulated version on a workstation. Only the first possibility will be described further in this paper.

First a custom CAML compiler performs parsing and polymorphic type-checking. The resulting annotated abstract syntax tree is then expanded into a (target-independent) parallel process network by instantiating each skeleton PNT.

This process graph — whose nodes are associated to user computing functions and/or skeleton control processes and edges indicates communication — is then mapped onto the target architecture, which is also described as a graph, with nodes associated to processors and edges representing communication channels. This task is handled by a third-party CAD software called SynDEx[13] which performs a static distribution of processes onto processors and a mixed static/dynamic scheduling of communications onto channels. This tool generates a dead-lock free distributed executive with optional real-time performance measurement. This executive takes the form of processor-independent programs (*m4* macro-code, one per processor) which are finally transformed into compilable code by simply inlining a set of kernel primitives. The code of these primitives — which basically support thread creation, communication and synchronisation and sequentialisation of user supplied computation functions and of inter-processor communications — is the only platform-dependant part of the programming environment, making it highly portable.

4 A realistic case study

SKiPPER has used to parallelize several algorithms for real-time vision applications including connected-component labelling [7], road-following by white line detection [6] and vehicle tracking [9]. The latter is illustrated in this section.

A video camera, installed in a car, provides a gray level image of several lead vehicle (one to three, in practice). Each lead vehicle is equipped with three visual marks, placed on the top and at the back of it (see figure 3).

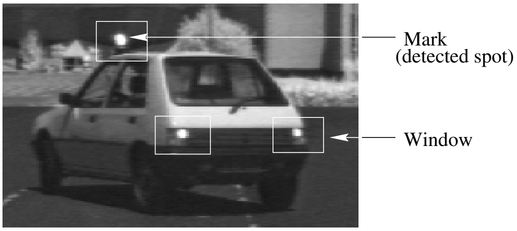


Fig. 3. Tracking algorithm

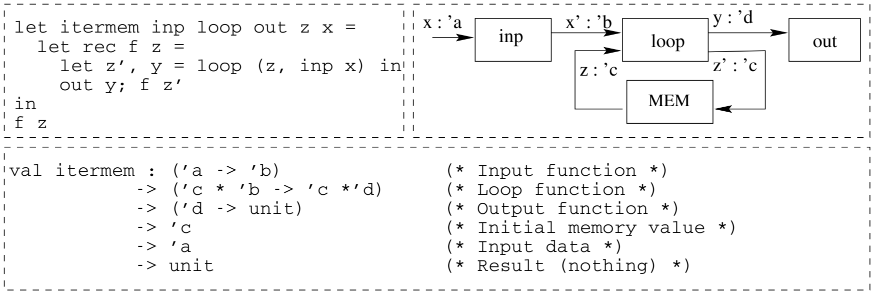
Algorithmically, the application can be divided into two main parts:

- First, detection of the marks in the image. Marks are detected as connected groups of pixels with values above a given threshold. Each mark is then characterized by computing its center of gravity and an englobing frame.

- Second, tracking each lead vehicles by a classical *predict-then-verify* method. The englobing frames of marks detected at iteration `i` are used to predict the position and size of the *windows of interest* in which the detection process will search for marks at iteration `i+1`. This is done using a 3D-modelling of each vehicle trajectory, coupled to a set of rigidity criteria to resolve ambiguous cases (occultations, *etc*). If less than three marks were detected at iteration `i`, it is assumed that the prediction failed, and windows of interests are obtained by dividing up the whole image into `n` equally-sized sub-windows, where `n` is typically taken equal to the total number of processors.

Two skeletons can be put into operation in this application:

- The input of the detection process is a list of windows. This list may vary in length (from 3,6 or 9 in normal tracking to `n` for the reinitialization phase) and each window may itself vary widely in size (its size depends on the apparent size of the marks, which in turn depends in the distance to the lead vehicle). Such dynamic behaviour, involving a very uneven work load, calls for a DF skeleton.
- The top-level prediction exhibits iterative behaviour, in which results computed at iteration `i` are used at iteration `i+1`. This is exactly what the ITERMEM, whose definition is given on figure 4, is designed for.



```
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
```

```
val itermem : ('a -> 'b)              (* Input function *)
           -> ('c * 'b -> 'c *'d)     (* Loop function *)
           -> ('d -> unit)            (* Output function *)
           -> 'c                      (* Initial memory value *)
           -> 'a                      (* Input data *)
           -> unit                    (* Result (nothing) *)
```

Fig. 4. The ITERMEM skeleton

The functional specification of the application can then be expressed as follows in Caml:

```
let nproc = 8;;
let s0 = init_state ();;
let loop (state, im) =
  let ws = get_windows nproc state im in
  let marks = df nproc detect_mark accum_marks empty_list ws in
  predict marks;;
let main = itermem read_img loop display_marks s0 (512,512);;
```

where

- **init_state** returns the initial state value for initiating the prediction algorithm (this state contains all the information required for positioning the windows),
- **get_windows** extracts the windows of the current image,
- **detect_mark** and **accum_marks** respectively detects and accumulates the position and size of marks in the selected windows
- **predict** returns both the position of the detected marks at the current iteration for display and the updated state value for the next iteration.

The associated C prototypes are:

```
void read_img(/*in*/ int nrows, /*in*/ int ncols, /*out*/ img *im);
void init_state(/*out*/ state *s);
void get_windows(/*in*/ int np, /*in*/ state *s, /*in*/ image *im,
  /*out*/ windowList *ws);
void detect_mark(/*in*/ window *w, /*out*/ mark *m);
void accum_marks(/*in*/ markList *old, /*in*/ mark *m,
  /*out*/ markList *new);
void predict(/*in*/ markList *marks, /*out*/ markList *ms,
  /*out*/ state *st);
void display_marks(/*in*/ markList *ms);
```

Starting from the above CAML specification and C code, SKiPPER has been used both to check the correctness of the parallelisation process (by using the sequential emulation facilities mentioned in section 3) and to derive a parallel implementations on a parallel vision machine with real-time video i/o facilities, the TRANSVISION platform [8]. This architecture is built upon Transputer processors and can be configured according to various physical topologies. The experiment here has been conducted using a ring-topology.

With a ring of 8 Transputers (T9000, 20MHz) operating on a 25 Hz 512×512 video stream, the minimal latencies obtained is 30ms for the tracking phase and 110 ms for the reinitialization phase, with the application processing each image of the video stream in first case, and one image out of 3 in the second. These performances are similar to the ones obtained by an existing hand-crafted parallel version of the algorithm and satisfy the timing constraints of the target application.

The main lesson drawn from this testbench, however was not on raw performances but on the effectiveness of the skeleton approach for writing complex portable parallel applications:

First, the programmer's work here reduced to writing 6 sequential C functions and the CAML specification given above. All underlying parallel implementation details (including process placement, communication scheduling, buffer allocation, provision for deadlock avoidance, etc.) were transparently handled by the environment. The result is that it took less than one day to get a first working implementation on the target platform and that it was then almost instantaneous to get variant versions with different numbers of processors. The previously hand-crafted parallel version had required at least ten times longer to implement. Moreover, it could not be scaled in a straightforward way (modifying the number of processors, for instance required significant changes in the C code).

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.

Third, the possibility to emulate the parallel code on a sequential workstation, though not described here has proven 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 been uncovered this way. 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).

5 Related work

The concept of algorithmic skeletons is not new and many researchers have worked (and are still working) to demonstrate their usefulness for portable parallel programming. Darlington's group at Imperial College [5] shares our view of skeletons as *coordinating* constructs for sequential functions written in C or Fortran, but mainly targets numerical applications with no real-time constraints. Michaelson's group at Heriot-Watt University [10] use skeleton in ML programs to denote sites of *potential* parallelism, leaving the responsibility of expanding them into parallel constructs to the compiler, on the basis of profiling information collected by an *instrumentation* phase. The P3L project at Pisa University [3] has developed a complete skeleton-based parallel programming language, in which sequential functions are written in C and skeletons are introduced as special constructs. Recently, Danelutto *et al.* [4] have proposed an integration of the P3L skeletons within the CAML language. Their work is very similar to ours. It is more general both in terms of the target application domain and expressibility (their skeletons can be freely nested ours not, in particular) but the provided implementation requires either a good OS-level support (Unix sockets) or a generic message passing library (MPI), thus precluding their use on embedded an/or dedicated vision platforms.

6 Conclusion

This paper has presented a methodology dedicated to the rapid prototyping of image processing applications on dedicated MIMD-DM architectures, based upon the concept of algorithmic skeletons. This methodology provides a tractable solution to the parallelisation problem, by restricting the expression of parallelism to a few forms admitting both a well-defined abstract semantics and one or more efficient implementations. A prototype system level software has been developed to support this methodology. It uses both a custom ML to process network compiler and an existing distributing/scheduling tool to turn a high-level skeletal specification into executable code. Preliminary results of this system — illustrated here with a realistic vision application — are encouraging, showing a dramatic reduction in development time while keeping satisfactory performances. Further developments are needed, however, first to see whether the approach can be extended to higher levels of image processing — for which the higher irregularity of algorithms may require more complex skeletons, and second to study inter-skeleton transformational rules, which are needed when applications are built by composing and/or nesting a large number of skeletons.

References

1. M. Cole. *Algorithmic skeletons: structured management of parallel computations.* Pitman/MIT Press, 1989.
2. G. Cousineau and M. Mauny. *The functional approach to programming.* Cambridge University Press, 1998 - see also: `http://pauillac.inria.fr/caml`.
3. M. Danelutto, F. Pasqualetti, and S. Pelagatti. Skeletons for data parallelism in p3l. In C. Lengauer, M. Griebl, and S. Gorlatch, editors, *Proc. of EURO-PAR '97, Passau, Germany*, volume 1300 of *LNCS*, pages 619–628. Springer, August 1997.
4. Marco Danelutto, Roberto DiCosmo, Xavier Leroy, and Susanna Pelagatti. Parallel functional programming with skeletons: the OCamlP3L experiment. In *Proceedings ACM workshop on ML and its applications.* Cornell University, 1998.
5. J. Darlington, Y. K Guo, H. W. To, and Y. Jing. Skeletons for structured parallel composition. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1995.
6. D. Ginhac. *Prototypage rapide d'applications de vision artificielle par squelettes fonctionnels.* PhD thesis, Univ. B. Pascal, 1999.
7. D. Ginhac, J. Sérot, and J.P. Dérutin. Fast prototyping of image processing applications using functional skeletons on a MIMD-DM architecture. In *IAPR Workshop on Machine Vision and Applications*, pages 468–471, Chiba, Japan, Nov 1998.
8. P. Legrand, R. Canals, and J.P. Dérutin. Edge and region segmentation processes on the parallel vision machine Transvision. In *Computer Architecture for Machine Perception*, pages 410–420, New-Orleans, USA, Dec 1993.
9. F. Marmoiton, F. Collange, P. Martinet, and J.P. Dérutin. A real time car tracker. In *International Conference on Advances in Vehicle Control and Safety*, Amiens, France, July 1998.
10. G. J. Michaelson and N. R. Scaife. Prototyping a parallel vision system in standard ML. *Journal of Functional Programming*, 1995.
11. D. B. Skillicorn. Architecture-independent parallel computation. *IEEE Computer*, 23(12):38–50, December 1990.
12. D. B. Skillicorn and D. Talia. Models and languages for parallel computation. *Computing Surveys*, June 1998.
13. Y. Sorel. Massively parallel systems with real time constraints. The "Algorithm Architecture Adequation" Methodology. In *Proc. Massively Parallel Computing Systems*, Ischia Italy, May 1994.