Managing Algorithmic Skeleton Nesting Requirements in Realistic Image Processing Applications: The Case of the SKiPPER-II Parallel Programming Environment's Operating Model

Rémi Coudarcher, Florent Duculty, Jocelyn Serot, Frédéric Jurie, Jean-Pierre Derutin, Michel Dhome

EURASIP Journal on Applied Signal Processing 2005:7, 1005–1023

Managing Algorithmic Skeleton Nesting Requirements in Realistic Image Processing Applications: The Case of the SKiPPER-II Parallel Programming Environment's Operating Model

Rémi Coudarcher,[1] Florent Duculty,[2] Jocelyn Serot,[2] Frédéric Jurie,[2] Jean-Pierre Derutin,[2] and Michel Dhome[2]

[1] *Projet OASIS, INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 Sophia-Antipolis Cedex, France*
Email: remi.coudarcher@sophia.inria.fr

[2] *LASMEA (UMR 6602 UBP/CNRS), Université Blaise-Pascal-(Clermont II), Campus Universitaire des Cézeaux, 24 avenue des Landais, 63177 Aubiere Cedex, France*
Emails: duculty@lasmea.univ-bpclermont.fr, jserot@lasmea.univ-bpclermont.fr, jurie@lasmea.univ-bpclermont.fr, derutin@lasmea.univ-bpclermont.fr, dhome@lasmea.univ-bpclermont.fr

Received 5 September 2003; Revised 17 August 2004

SKiPPER is a Skeleton-based Parallel Programming EnviRonment being developed since 1996 and running at LASMEA Laboratory, the Blaise-Pascal University, France. The main goal of the project was to demonstrate the applicability of skeleton-based parallel programming techniques to the fast prototyping of reactive vision applications. This paper deals with the special features embedded in the latest version of the project: algorithmic skeleton nesting capabilities and a fully dynamic operating model. Throughout the case study of a complete and realistic image processing application, in which we have pointed out the requirement for skeleton nesting, we are presenting the operating model of this feature. The work described here is one of the few reported experiments showing the application of skeleton nesting facilities for the parallelisation of a realistic application, especially in the area of image processing. The image processing application we have chosen is a 3D face-tracking algorithm from appearance.

Keywords and phrases: parallel programming, image processing, algorithmic skeleton, nesting, 3D face tracking.

1. INTRODUCTION

At Laboratoire des Sciences et Matériaux pour l'Electronique, et d'Automatique (LASMEA), the Blaise-Pascal University's laboratory of electrical engineering, France, we have been developing since 1996 a parallel programming environment, called SKiPPER (SKeleton-based Parallel Programming EnviRonment), based on the use of algorithmic skeletons to provide application programmers with a mostly automatic procedure for designing and implementing parallel applications. The SKiPPER project was originally envisoned to build realistic vision applications for embedded platforms.

Due to the features in the latest developed version of SKiPPER, called SKiPPER-II, it has now turned into a more usable parallel programming environment addressing PC cluster architectures and different kinds of applications as well.

The reason to develop such an environment is that, relying on parallel machines, programmers are facing several difficulties. Indeed, 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,[1] thus hindering a true experimental approach. Parallel programming at a low level of abstraction also limits code reusability and portability. Our environment finally tries to "capture" the expertise gained by programmers when implementing vision applications using low-level parallel constructs, in order to make it readily available to algorithmists and image processing specialists. That is the reason why SKiPPER takes into account low-level implementation details such as task partitioning and mapping, communication scheduling, or load balancing.

[1] Especially when the persons in charge of developing the algorithms are image processing, and not parallel programming, specialists.

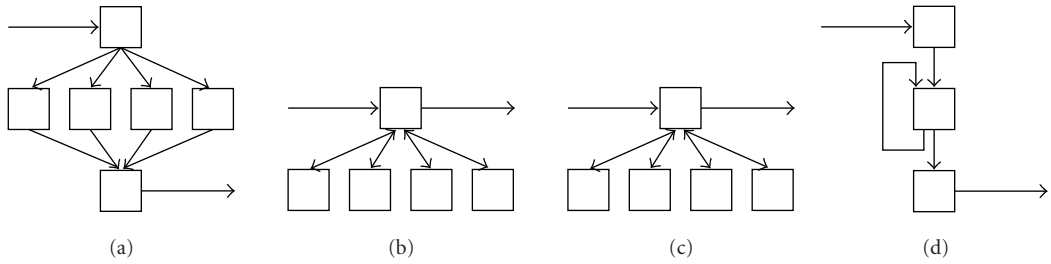


Figure 1: The four skeletons of SKiPPER are, from left to right, split-compute-merge skeleton, data farming skeleton, task farming skeleton, and iteration with memory skeleton.

The SKiPPER-I suite of tools, described in [1, 2, 3, 4], was the first realization of this methodology. It was, however, limited in terms of skeleton composition. In particular, it could not accommodate arbitrary skeleton nesting, that is to say, the possibility for one skeleton to take another skeleton as an argument. The SKiPPER-II implementation [5] was developed to solve this problem. Its main innovative feature is its ability to handle arbitrary skeleton nesting.

Skeleton nesting has always been perceived as a challenge by skeleton implementers and only a few projects have produced working implementations supporting it (see, e.g., [6, 7, 8]). But most of the applications used in these cases were "toy" programs in which skeleton nesting is a rather "artificial" construct needed for benchmarking purposes. By contrast, we think that showing a realistic application which needs such a facility in order to be parallelised has a great importance in validating the concept.

For these reasons, this paper focuses on the parallelisation, using a set of algorithmic skeletons specifically designed for image processing, of a complete and realistic image processing application in which we have pointed out requirements of skeleton nesting. The realistic application we have chosen is a 3D face-tracking algorithm which had been previously developed in our laboratory [9].

The paper is organised as follows. Section 2 briefly recalls the skeleton-based parallel programming concepts used in SKiPPER and describes the suite of tools that has been developed. Section 3 presents the 3D face-tracking algorithm we used as a realistic case study to be parallelised using the SKiPPER-II environment. Only the main features of the algorithm are described here in a way that our design choices (in terms of parallelisation) could be understood. These design choices are described in Section 4. Result analysis can be found in Section 5. Finally Section 6 concludes the paper.

2. THE SKiPPER PROJECT

2.1. Skeleton-based parallel programming and SKiPPER-I

Skeleton-based parallel programming methodologies (see [10, 11]) provide a way for conciliating fast prototyping and efficiency. They aim at providing user guidance and a mostly automatic procedure for designing and implementing parallel applications. For that purpose, they provide a set of *algorithmic skeletons*, which are *higher-order program constructs encapsulating common and recurrent forms of parallelism* to make them readily available for the application programmer. The latter does not have to take into account low-level implementation details such as task partitioning and mapping, data distribution, communication scheduling, and load balancing.

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 overall result is a significant reduction in the design-implement-validate cycle time.

Due to our primary interest in image processing, we have designed and implemented a skeleton-based parallel programming environment, called SKiPPER, based on a set of skeletons specifically designed for parallel vision applications [1, 2, 3, 4, 12]. This library of skeletons was designed from a retrospective analysis of existing parallel code. It includes four skeletons (as shown in Figure 1):

(i) split-compute-merge (SCM) skeleton;
(ii) data farming (DF);
(iii) task farming (TF) (a recursive version of the DF skeleton);
(iv) iteration with memory (ITERMEM).

The SCM skeleton is devoted to regular "geometric" processing of iconic data, in which the input set of data is split into a fixed number of subsets, each of them is processed independently and the final result is obtained by merging the results computed on subsets of the input data (they may overlap). This skeleton is applicable whenever the number of subsets is fixed and the amount of work on each subset is the same, resulting in an even workload. Typical examples include convolutions, median filtering, and histogram computation.

The DF skeleton is a generic harness for *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 them computing 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. 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.

```
let scm split comp merge x =
  merge (map comp (split x))
let df comp acc xs =
  foldll 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 foldll comb (map f xs)
```

Algorithm 1: Declarative semantics of SKiPPER skeletons in Caml.

```
let img = read_img        512 512 ;;
let histo = scm           row_partition
                          seq_histo
                          merge_histo
                          img ;;
let main = display_histo  img histo ;;
```

Algorithm 2: A "skeletal" program in Caml.

The TF skeleton may be viewed as a generalisation of the DF one, in which the processing of one data item by a 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 classical recursive divide-and-conquer algorithms.

The ITERMEM skeleton does not actually encapsulate parallel behaviour, 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. A typical situation is when computations on the nth image depend on results computed on the $n-1$th (or $n-k$th).

Each skeleton comes with two semantics: a declarative semantics, which gives its "meaning" to the application programmer in an implicitly parallel manner, that is, 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 SKiPPER versions. It is conveyed using the Caml language, using higher-order polymorphic functions. The corresponding definitions are given in Algorithm 1. Potential (implicit) parallelism arises from the use of the "map" and "foldl1" higher-order functions.

The operational semantics of a skeleton varies according to the nature of the intermediate representation used by the CTS.

Using SKiPPER, the application designer

(i) provides the source code of the sequential application-specific functions;

(ii) describes the parallel structure of his application in terms of composition of skeletons chosen in the library.

This description is made by using a subset of the Caml functional language, as shown in Algorithm 2, where a SCM skeleton is used to express the parallel computation of a histogram using a geometric partitioning of the input image. In this Algorithm, "row_ partition," "seq_histo," "merge_histo," and "display_histo" are the application-specific sequential functions (written in C) and "scm" is the above-mentioned skeleton. This Caml program is the *skeletal program specification*. In SKiPPER-I, it is turned into executable code by first translating it into a graph of parametric process templates and then mapping this graph onto the target architecture. The SKiPPER suite of tools turn 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 (see Figure 2). The role of the back-end in the CTS is to map the intermediate representation of the parallel program 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, that is, done at compile time, or dynamic, that is, postponed until run time. Both approaches require some kind of 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. For dynamic approaches, it must include more sophisticated mechanisms for scheduling threads and/or processes and dynamically managing communication buffers for instance. For this reason, static approaches generally lead to better (and more predictable) performances. But 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. 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.

Completely designed by the end of 1998, SKiPPER-I has already been used for implementing several realistic parallel vision applications, such as connected component labelling [1], vehicle tracking [3], and road tracking/reconstruction [4].

But SKiPPER-I did not support *skeleton nesting*, that is, the ability for a skeleton to take another skeleton as

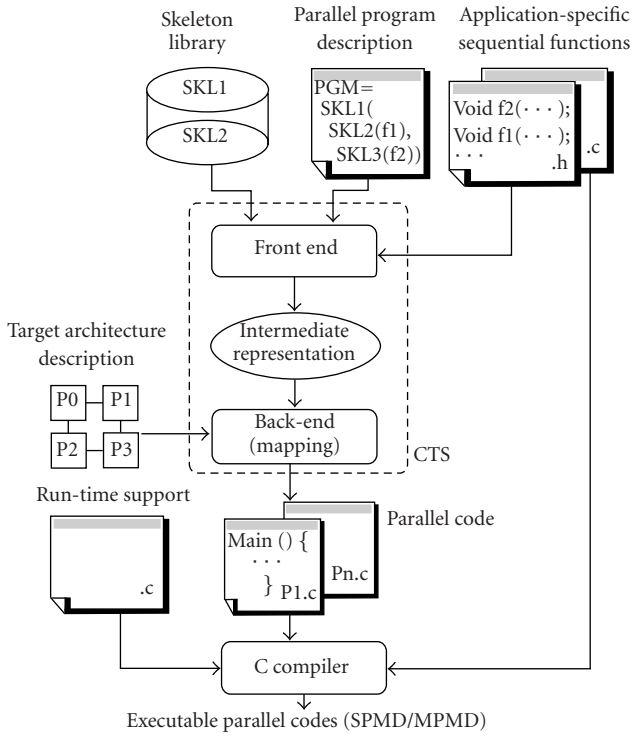


Figure 2: SKiPPER global software architecture.

argument. Arbitrary skeleton nesting raises challenging implementation issues as reported in [6, 8, 13] or [7]. For this reason, SKiPPER-II was designed to support arbitrary nesting of skeletons. This implementation is based on a completely revised execution model for skeletons. Its three main features are

(i) the reformulation of all skeletons as instances of a very general one: a new version of the task farming skeleton (called TF/II),
(ii) a fully dynamic scheduling mechanism (scheduling was mainly static in SKiPPER-I),
(iii) a portable implementation of skeletons based on an *MPI* communication layer (see Section 2.5).

2.2. SKiPPER-II

SKiPPER-I relied on a mostly static execution model for skeletons: most of the decisions regarding distribution of computations and scheduling of communications were made at compile time by a third-party CAD software called SynDEx [14]. This implementation path, while resulting in very efficient distributed executives for "static"—by static we mean that the distribution and scheduling of all communications do not depend on input data and can be predicted at compile-time—did not directly support "dynamic" skeletons, in particular those based on data or task farming (DF and TF). The intermediate representation of DF and TF was therefore rather awkward in SKiPPER-I, relying on *ad hoc* auxiliary processes and synchronisation barriers to hide dynamically scheduled communications from the static scheduler [2].

Another point about the design of SKiPPER-I is that the target executable code was MPMD: the final parallel C code took the form of a set of distinct main functions (one per processor), each containing direct calls to the application-specific sequential functions interleaved with communications.

By contrast, execution of skeleton-based applications in SKiPPER-II is carried out by a *single* program (the "kernel" in the sequel)—written in C—running in SPMD mode on all processors and ensuring a fully *dynamic* distribution and scheduling of processes and communications. The kernel's work is to

(i) run the application by interpreting an intermediate description of the application obtained from the Caml program source,
(ii) emulate any skeleton of the previous version of SKiPPER,
(iii) manage resources (processors) for load balancing when multiple skeletons must run simultaneously.

In SKiPPER-II, the target executable code is therefore built from the kernel and the application-specific sequential functions. Indeed, the kernel acts as a small (distributed) operating system that provides all routines the application needs to run on a processor network.

The overall software architecture of the SKiPPER-II programming environment is given in Figure 3. The *skeletal specification* in Caml is analysed to produce the intermediate description which is interpreted at run time by the kernel; the sequential functions and the kernel code are compiled together to make the target executable code. These points will be detailed in the next sections.

2.3. Intermediate description

Clearly, the validity of the "kernel-based" approach presented above depends on the definition of an adequate *intermediate description*. It is the interpretation (at run time) of this description by the kernel that will trigger the execution of the application-specific sequential functions on the processors, according to the data dependencies encoded by the skeletons.

A key point about SKiPPER-II is that, at this intermediate level of description, all skeletons have been turned into *instances* of a more general one called TF/II. The operational semantics of the TF/II skeleton is similar to the one of DF and TF: a *master* (farmer) process still doles out tasks to a pool of *worker* (slave) processes, but the tasks can now be different (i.e., each worker can compute a different function).

Compared to the intermediate representation used in the previous version of SKiPPER, using a homogeneous intermediate representation of parallel programs is a design choice made in order to overcome the difficulties raised by hybrid representations and to solve the problem of skeleton nesting in a systematic way. More precisely the rationale for this "uniformisation" step is threefold.

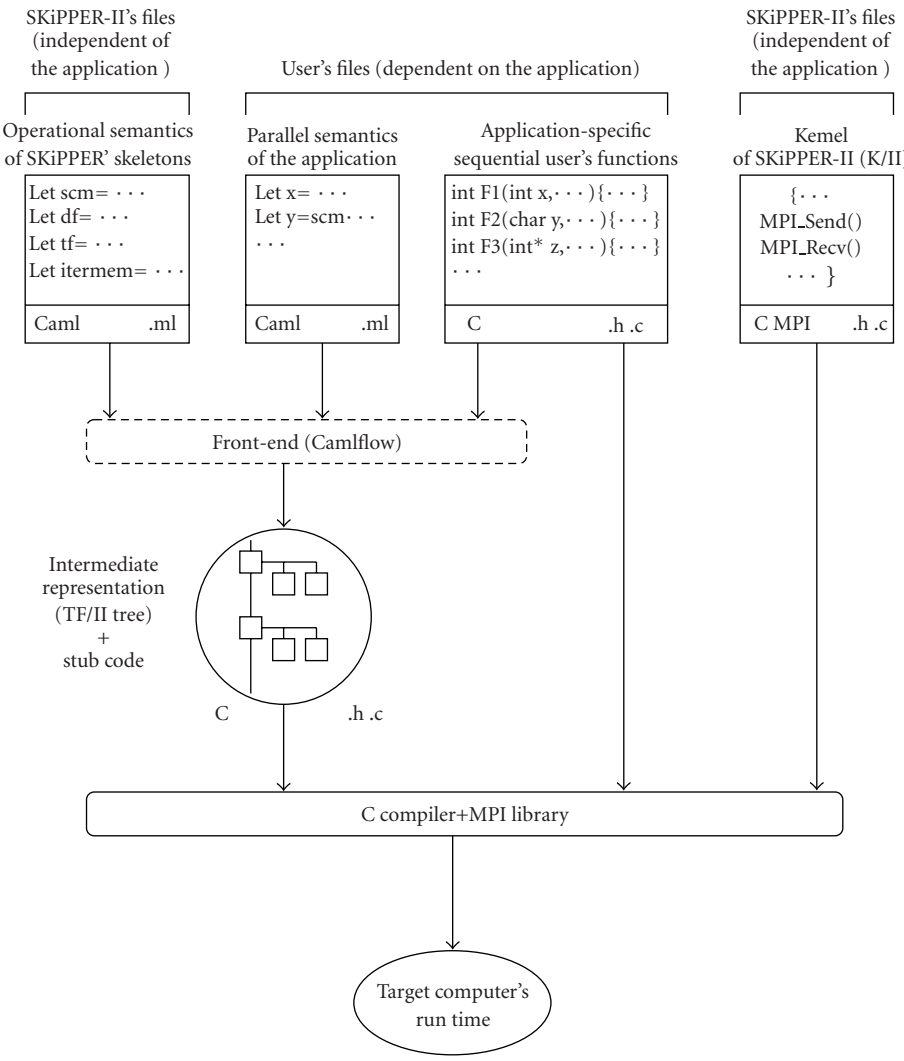


Figure 3: SKiPPER-II environment.

(i) First, it makes skeleton composition easier, because the number of possible combinations now reduces to three (TF/II followed by TF/II, TF/II nested in TF/II, or TF/II in parallel with TF/II).

(ii) Second, it greatly simplifies the structure of the run-time kernel, which only has to know how to run a TF/II skeleton.

(iii) Third, there is only one skeleton code to design and maintain, since all other skeletons will be defined in terms of this generic skeleton.

The above-mentioned transformation is illustrated in Figure 4 with a SCM skeleton. In this figure, white boxes represent pure sequential functions and grey boxes represent "support" processes (parameterised by sequential functions). Note that at the Caml level, the programmer still uses distinct skeletons (SCM, DF, TF, ITERMEM) when writing the skeletal description of his application.[2] The transformation is done by simply providing alternative definitions of the SCM, DF, TF, and so forth higher-order functions in terms of the TF/II one. Skeleton composition is expressed by normal functional composition. The program description appearing in Figure 5, for example, can be written as in Algorithm 3 in Caml.

The intermediate description itself—as interpreted by the kernel—is a tree of TF/II descriptors, where each node contains informations to identify the next skeleton and to retrieve the C function run by a worker process. Figure 5 shows an example of the corresponding data structure in the case of two nested SCM skeletons.

[2]Typically, the programmer continues to write his architecture/coordination-level program as the following Caml program: let main x = scm s f m x;;.

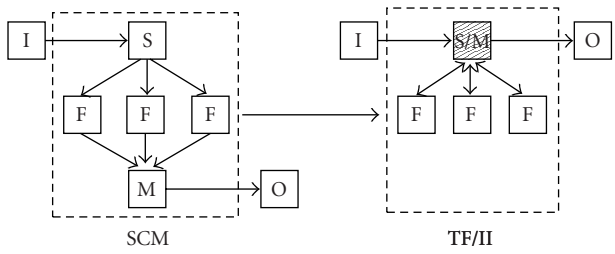


Figure 4: SCM → TF/II transformation.

2.4. Operating model

Within our fully dynamic operating/execution model, skeletons are viewed as concurrent processes competing for resources on the processor network.

When a skeleton needs to be run, and because any skeleton is now viewed as a TF/II instance, a kernel copy acts as the master process of the TF/II. This copy manages all data transfers between the master and the worker (slave) processes of the TF/II. Slave processes are located on resources allocated *dynamically* by the master. In this way, kernel copies interact to emulate skeleton behaviour. In this model, kernel copies (and then processes) can switch from master to worker behaviour depending only on the intermediate representation requirement. There is no "fixed" mapping for dynamic skeletons as in SKiPPER-I. As soon as a kernel copy is released after being involved in the emulation of a skeleton, it can be immediately reused in the emulation of another one. This strongly contributes towards easily managing the load-balancing and then efficiently using the available resources.

This is illustrated in Figure 6 with a small program showing two nested SCM skeletons. This figure shows the role of each kernel copy (two per processor in this case) in the execution of the intermediate description resulting from the transformation of the SCM skeletons into TF/II ones.

Because any kernel copy knows when and where to start a new skeleton without requiring information from copies, the scheduling of skeletons can be distributed. Each copy of the kernel has its own copy of the intermediate description of the application. This means that each processor can start the necessary skeleton when it is needed because it knows which skeleton has stopped. A new skeleton is started whenever the previous one (in the intermediate description) ends. The next skeleton is always started on the processor which has run the previous skeleton (because this resource is supposed to be free and closer than the others!).

Since we want to target dedicated and/or embedded platforms, the kernel was designed to work even if the computing nodes are not able to run more than one process at a time (no need for multitasking).

Finally, in the case of a lack of resources, the kernel is able to run some of the skeletons in a sequential manner, including the whole application, thus providing a straightforward sequential emulation facility for parallel programs.

2.5. Management of communications

The communication layer is based on a reduced set of the MPI [15] library functions (typically MPI_SSend or MPI_Recv), thus increasing the portability of skeleton-based applications across different parallel platforms [16]. This feature has been taken into account from the very beginning of the kernel's design of SKiPPER-II. We use only synchronous communication functions; however, asynchronous functions may perform much better in some cases (especially when the platform has a specialised coprocessor for communications and when communications and processing can overlap).

This restriction is a consequence of our original experimental platform which did not support asynchronous communications. This set of communication functions is the most elementary functions of the MPI toolset which can be implemented onto any kind of parallel computer. In such a way, the portability of SKiPPER-II is increased. Moreover, the usability is also higher due to writing a minimum MPI layer to support the execution of SKiPPER is a straightforward and not time-consuming task.

Multithreads were avoided too. Using multithreads in our first context of development, that is to say, with our first experimental platform was not suitable. This platform did not support multithreads,[3] giving us the minimal design requirement for a full platform compatibility.

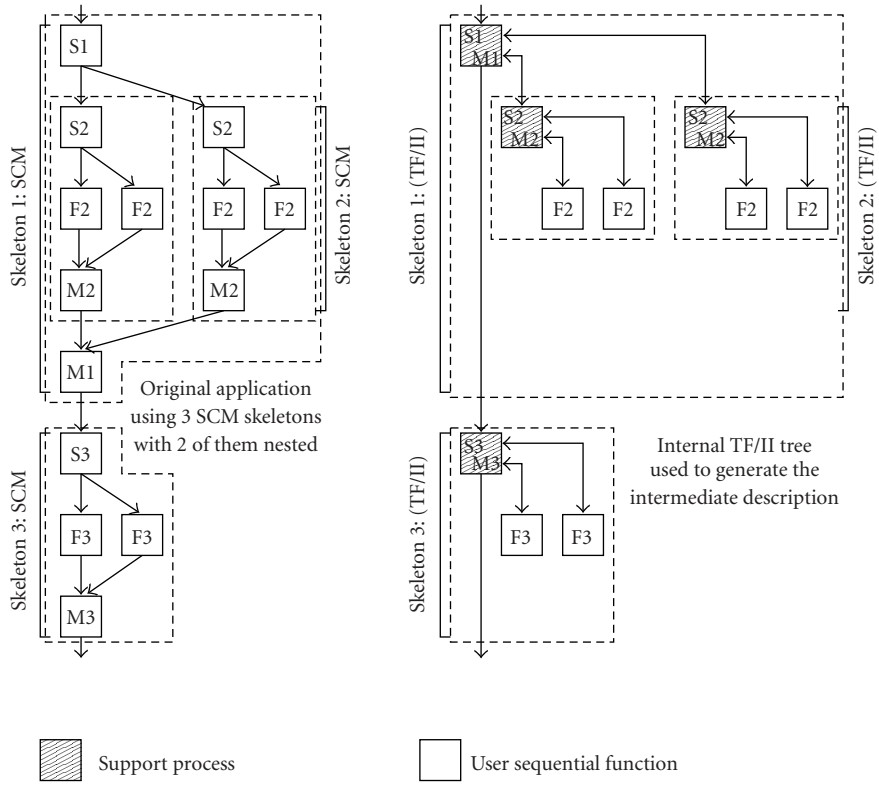
2.6. Comparative assessment

Comparatively to the first version of SKiPPER, SKiPPER-II uses a fully dynamic implementation mechanism for skeleton-based programs.

This has several advantages. In terms of expressivity, since arbitrary nesting of skeletons is naturally and fully supported. The introduction of new skeletons is also facilitated, since it only requires giving their translation in terms of TF/II. Portability of parallel applications across different platforms is extremely high: running an application on a new platform only requires a C compiler and a very small subset of the MPI library (easily written for any kind of parallel platform). The approach used also provides automatic load balancing, since all mapping and scheduling decisions are taken at run time, depending on the available physical resources. In a same way, sequential emulation is straight obtained in just running the parallel application on a single processor. This is the harder case of a lack of resources in which the SKiPPER-II's kernel automatically manages to run application as parallel as possible, running some part of it in sequential on a single processor in order to avoid to stopping the whole application.

The counterpart is essentially in terms of efficiency in some cases and mostly predictability. As regards to efficiency,

[3] SKiPPER-II was running onto several platforms as Beowulf machines and such clusters. But it was initially designed for a prototype parallel computer, built in our laboratory, dedicated to real-time image processing. This parallel computer is running without any operating system and thus applications are running in a stand-alone mode. No facilities encountered in modern operating systems were available.



Intermediate description:

1. Next skeleton = 3
   Split function = S1
   Merge function = M1
   Slave function = None
   Slave function type = User function
   Nested skeleton = 2

2. Next skeleton = None
   Split function = S2
   Merge function = M2
   Slave function = F2
   Slave function type = User function
   Nested skeleton = None

3. Next skeleton = None
   Split function = S3
   Merge function = M3
   Slave function = F3
   Slave function type = User function
   Nested skeleton = None

When 'slave function type' is set 'Skeleton' then 'Nested skeleton' field is used to know which skeleton must be used as a slave, that is to say, which skeleton must be nested in.

Figure 5: Intermediate description data structure example.

```
let nested  x = scm s2  f2        m2 x ;;
let main1   y = scm s1  nested    m1 x ;;
let main2   z = scm s3  f3        m3 y ;;
```

Algorithm 3: Program description appearing in Figure 5.

our experiments [16] have shown that the dynamic process distribution used may entail a performance penalty in some specific cases. For instance, we have implemented three standard programs as they have already been implemented in [2] for the study of the first version of SKiPPER.[4] The first benchmark was done computing a histogram on an image (using the SCM skeleton), the second was performed detecting spotlights in an image (using the DF skeleton), and finally the third one was performed on a divide-and-conquer algorithm for image processing (using the TF skeleton). We have reprinted the results in Figures 7, 8, 9, 10, 11, 12, 13, 14, 15, and 16.

[4] Please refer to [16] for more details about the benchmarks.

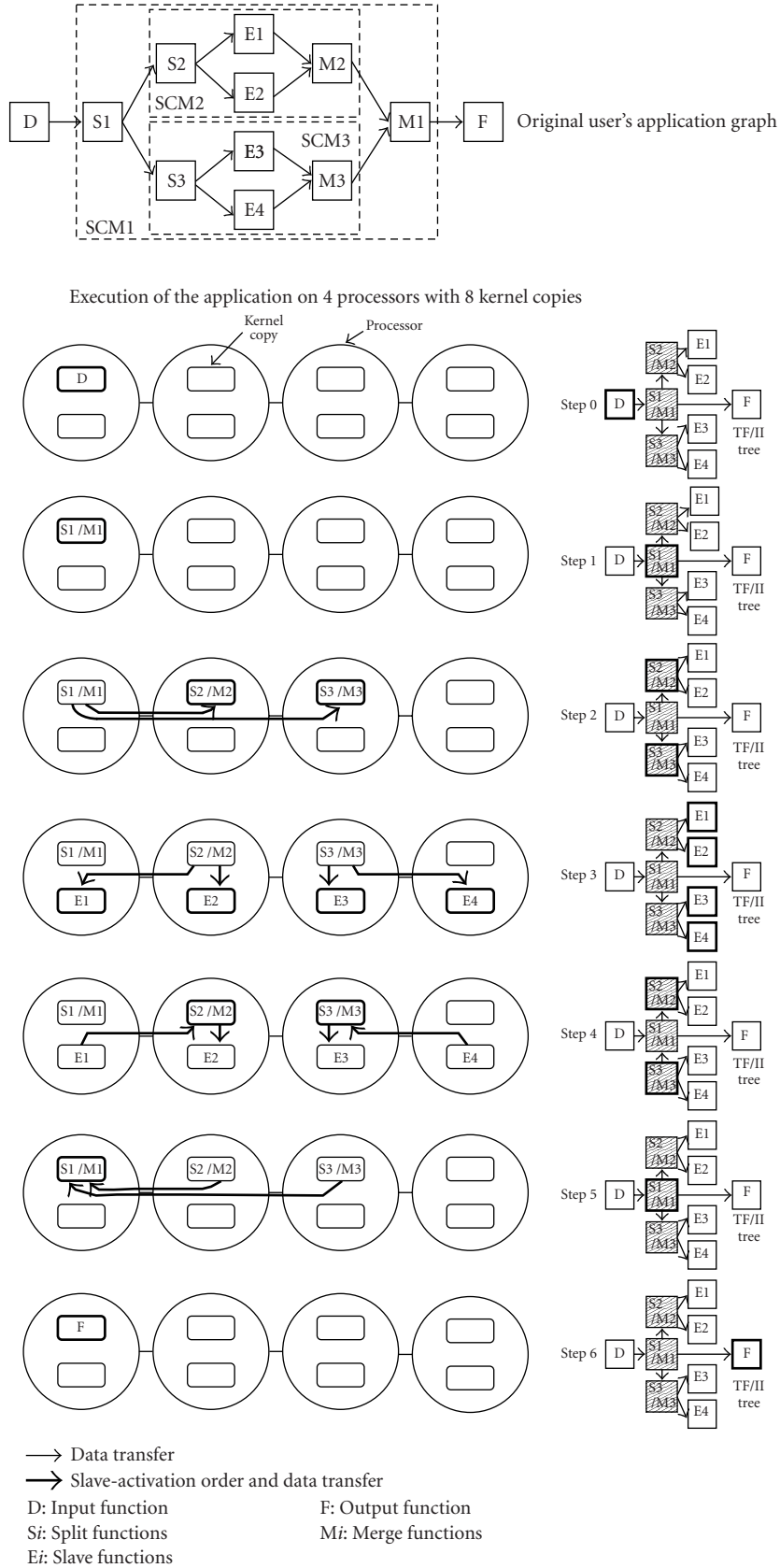


Figure 6: Example of the execution of two SCMs nested in one SCM.

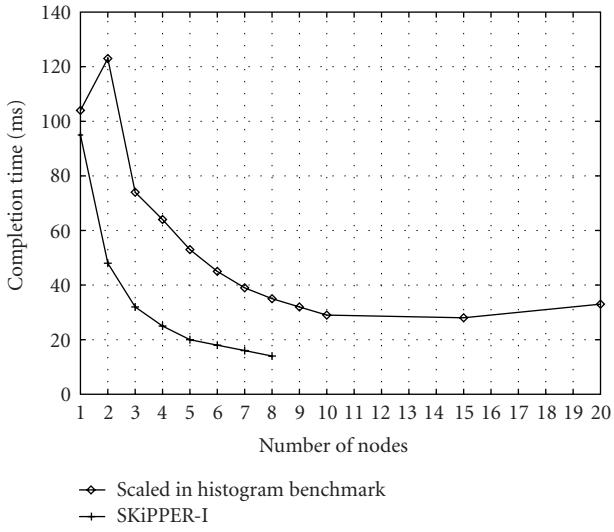


Figure 7: Completion time for the histogram benchmark (extract of [16]) (picture size: 512 × 512/8 bits, homogeneous computing power).

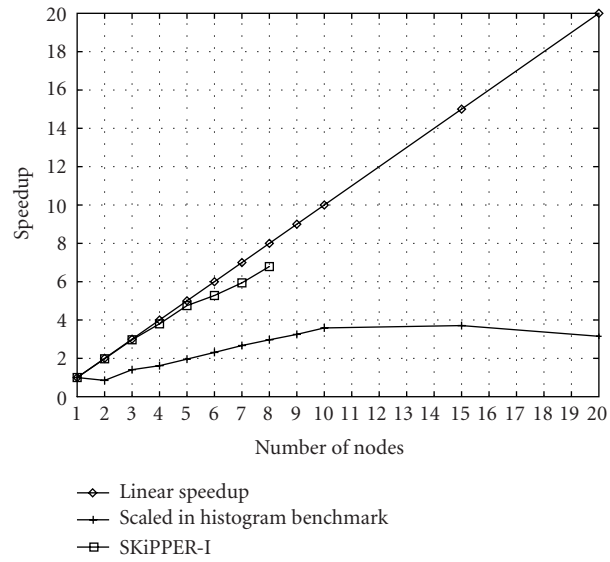


Figure 8: Speedup for the histogram benchmark (extract of [16]) (picture size: 512 × 512/8 bits, homogeneous computing power).

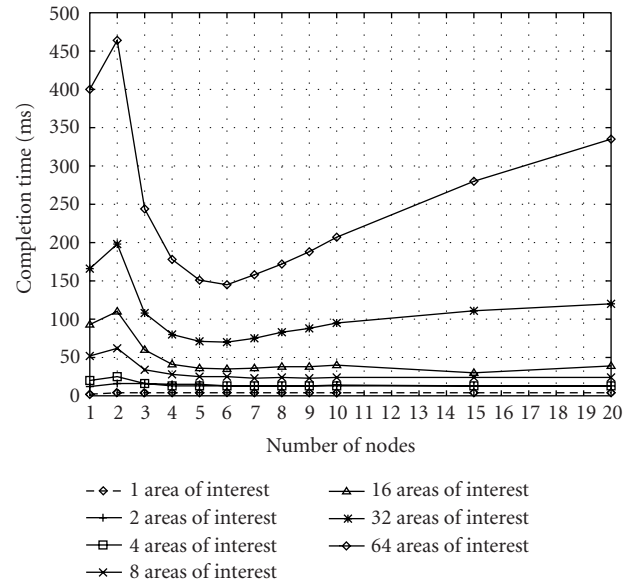


Figure 9: Completion time for the spotlight detection benchmark (SKiPPER-II) (extract of [16]) (picture size: 512×512/8 bits, homogeneous computing power).

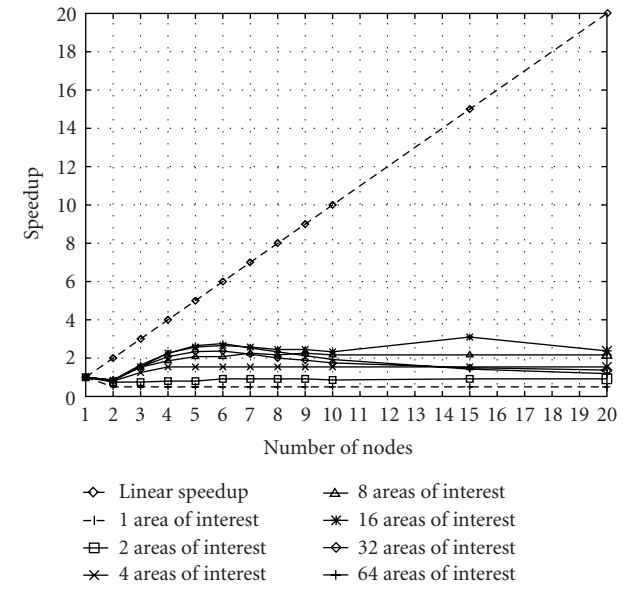


Figure 10: Speedup for the spotlight detection benchmark (SKiPPER-II) (extract of [16]) (picture size: 512 × 512/8 bits, homogeneous computing power).

The main difference between SKiPPER-I and -II is the behaviour of the latest with very few resources (typically, between 2 and 4 processors). This is due to the way SKiPPER-II involves kernel's copy into a skeleton run. Up to the number of processors available when SKiPPER-I benchmarks were performed (1998), the behaviour of SKiPPER-II is very closed (taking into account the difference of computing power between the experimental platform used in 1998 and the one in 2002 (see [16] for details)). Actually, the most counterpart concerning efficiency is exhibited with a low computation versus communication ratio. This has been shown comparing a C and MPI implementation and a SKiPPER-II of a same application. The reason is that the kernel performs more communications in exchanging data between inner and outer masters in case of skeleton nesting. Finally, the cost is mainly in terms of resources involved into the execution of a single skeleton.

As for the predictability of performances, the fully dynamic approach of SKiPPER-II makes it very difficult to obtain. Indeed, dealing with the operating model, processes can switch from master to slave/worker behaviour depending only on the need for skeletons. There is not a "fixed"

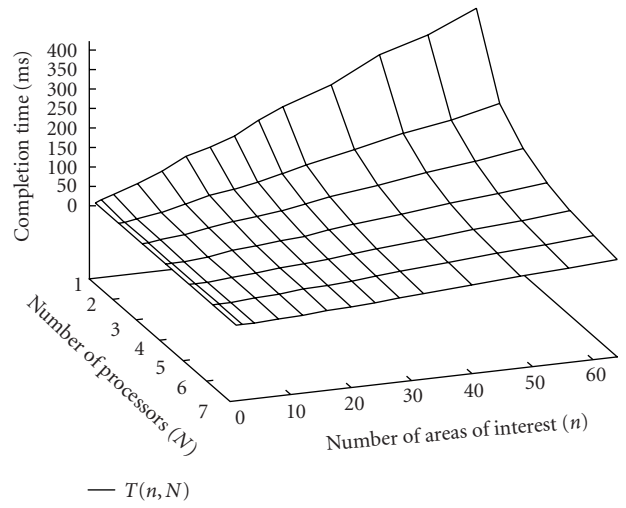


Figure 11: Completion time for the spotlight detection benchmark (SKiPPER-I) (extract of [2]).

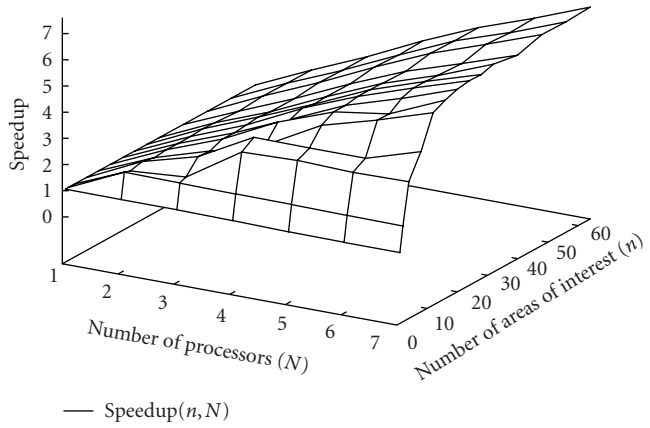


Figure 12: Speedup for the spotlight detection benchmark (SKiPPER-I) (extract of [2]).

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.

3. THE 3D FACE-TRACKING ALGORITHM

3.1. Introduction

The application we have chosen is a tracking of 3D human faces in image sequences, using only face appearances (i.e., a viewer-based representation). An algorithm developed earlier allows to track the movement of a 2D visual pattern in a video sequence. It constitutes the core of our approach. In [9], this algorithm is fully described and experimentally tested. An interesting application is face tracking for the videoconference.

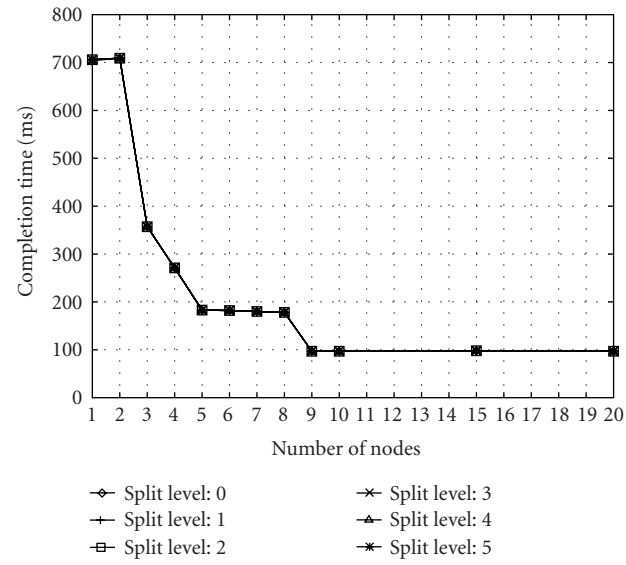


Figure 13: Completion time for the divide-and-conquer benchmark (extract of [16]) (picture size: 512 × 512/8 bits, homogeneous computing power).

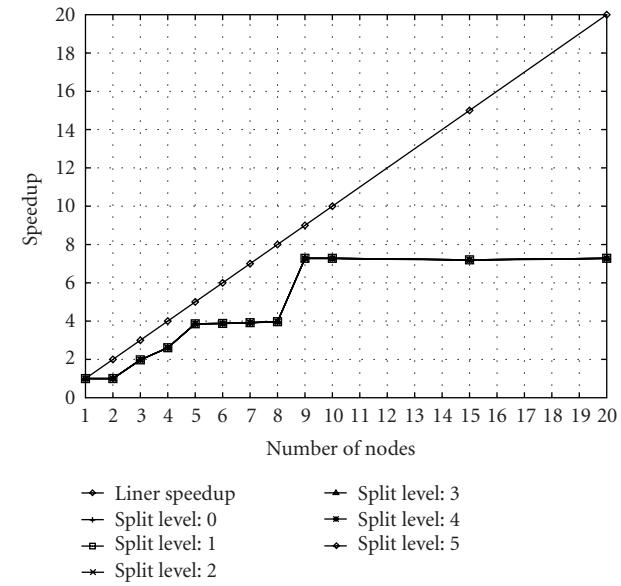


Figure 14: Speedup for the divide-and-conquer benchmark (extract of [16]) (picture size: 512 × 512/8 bits, homogeneous computing power).

In our 3D tracking approach, a face is represented by a collection of 2D images called *reference views* (appearances to be tracked). Moreover, a pattern is a region of the image defined in an area of interest and its sampling gives a gray-level vector. The tracking technique involves two stages. An off-line learning stage is devoted to the computation of an *interaction matrix A* for each of these views. This matrix relates the gray-level difference between the tracked reference pattern and the current pattern sampled in the area of interest to its "fronto-parallel" movement. By definition, a "fronto-parallel" movement is a movement of the face in a plane

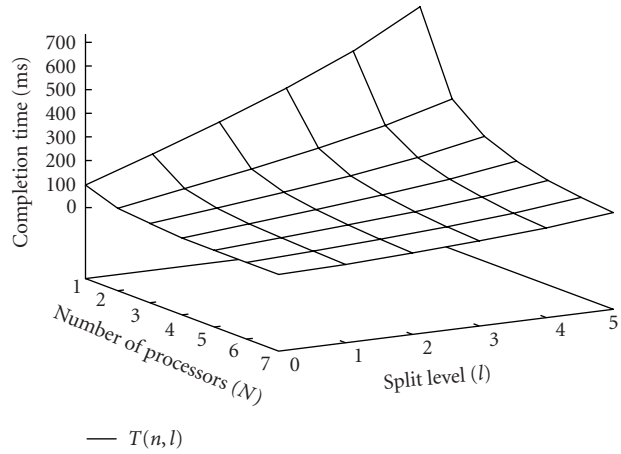


Figure 15: Completion time for the divide-and-conquer benchmark (SKiPPER-I) (extract of [2]).

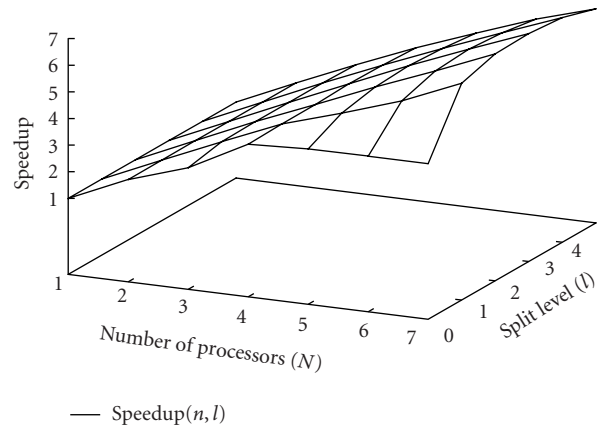


Figure 16: Speedup for the divide-and-conquer benchmark (SKiPPER-I) (extract of [2]).

which is parallel to the image's plane. The global aspect of the pattern representing the tracked face is not modified by this movement. However, the position, the orientation, and the size of the pattern can change. In an independent way, the on-line stage (Figure 17) consists in predicting the position of the face inside an elliptic area in the current image (in position, orientation, and size) and in estimating the correction of ellipse parameters (the target region is supposed to be included in an ellipse) for the current reference pattern and the nearest reference patterns in the collection of images (the previous and the next reference patterns for a movement in roll). Each of these corrections is obtained by multiplying the gray-level difference between the visual pattern in the predicted ellipse and the different reference patterns using the associated interaction matrix. For each of these tested reference patterns, we obtain a new position of the ellipse in the current image which is supposed to overlap the real position of the face. For each new position, we estimate the quadratic error of the gray-level difference ΔVI between the current pattern inside the area of interest VI_c and the associate reference pattern VI_{ref}. The reference pattern giving the smallest quadratic error will be considered as the new reference pattern to be tracked. This simultaneous test on several reference patterns allows to manage the appearance variations due to the movements in roll of the face in the image and to change the reference pattern without stopping the tracking process. As the frequency of the treatment is important compared to the speed of the tracked face, we do not need to use any prediction algorithm. Indeed, the variation of the pattern's position between two successive images remains compatible with the variations recorded during the learning stage.

3.2. Modelling appearance of 3D faces

Faces are highly variable, deformable objects that manifest very different appearances in images depending on pose, lighting, expression, and the identity of the person. In our 3D tracking approach, a face is represented by a collection of 2D images called *reference views*. Each of these images represents one of the reference patterns of the 3D face to be used for a given relative attitude between the face and the camera. These images perform the 2D tracking of possible patterns. The acquisition of intermediate views will then enable us to save, during the learning stage, the different corrected positions of the area of interest for the nearest reference patterns in the collection of images (Figure 18).

So, during the tracking phase, we will be able to position, before correction, the predicted areas of interest of previous and next reference patterns compared to the area of interest of the currently tracked reference pattern. The parallel tracking of three reference patterns will enable us to consider the variations of aspect of the current pattern in the image. The switch between view will be done without stopping the tracking process (Figure 19).

The learning base includes 71 views (8 reference patterns and 63 intermediate views) for a privileged variation in roll of the face movement on a 180-degree range. We assume that we will be able to track a reference pattern in the intermediate views up to the next reference view. We wish to represent the pattern to be tracked by a shape vector (gray-level vector of size N where $N = 170$ is the number of sampled points taken in a region of interest). This representation of the pattern in the image has to be invariant in position, orientation, and scale. This is why we propose to sample the pattern inside an elliptic area. The sample points (white dots in Figure 19) are distributed on a set of ten concentric ellipses from the smallest to the largest and are always numbered in the same order in the shape vector. This uniform sampling will enable us to limit the influences of expression changes during the tracking. The position and the shape of the ellipse (Figure 19) are defined by a vector having five parameters corresponding to the position of the center (X_c, Y_c), the orientation (θ), and the length of the major and minor axes (R_1, R_2). We define $R_2 = k * R_1$, where k is a known ratio given by the user, in order to have only one scale factor. Moreover, to guarantee a certain insensitivity to the affine changes in the illumination conditions of the scene, the shape vector, once sampled, is centered and normalised.

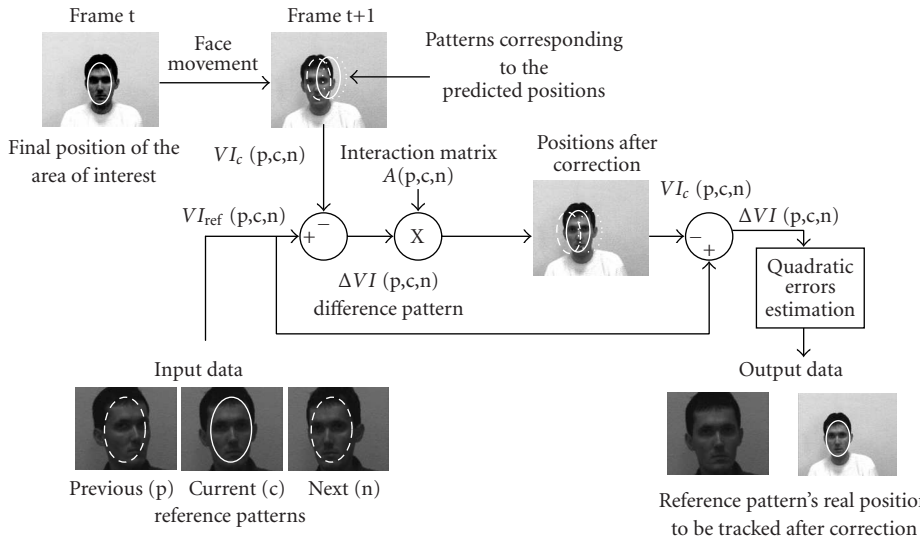


Figure 17: 3D tracking algorithm.

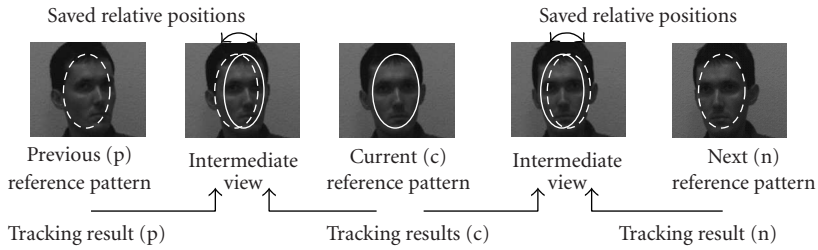


Figure 18: Intermediate views used during the learning stage.

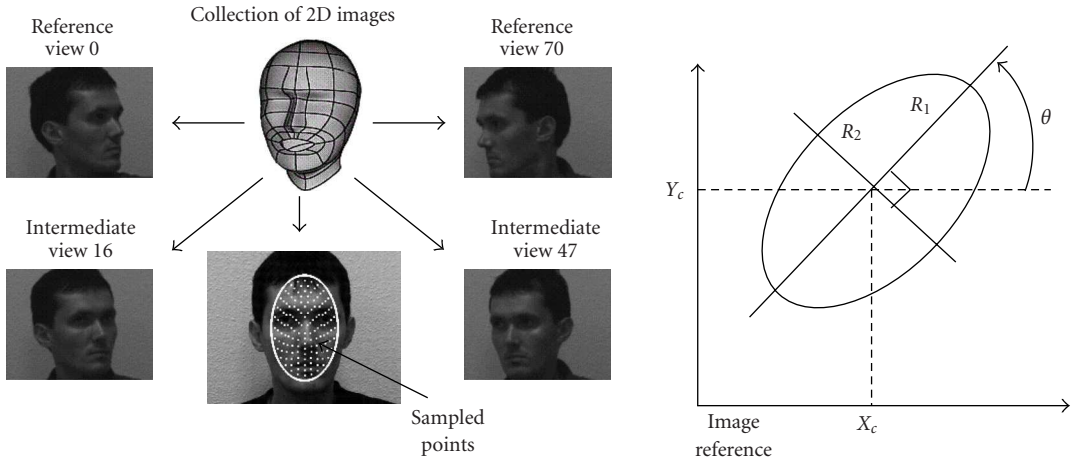


Figure 19: Model of a 3D face and sampling of a pattern.

In the next part, we develop succinctly the theoretical aspect of the 3D tracking algorithm in combining the tracking of a 2D visual pattern for a given reference pattern and the switching between reference patterns [7, 9].

3.3. Tracking principle and geometrical interpretation

We have just seen that the tracked pattern is framed in an ellipse whose form and position in the image are given by the vector of parameters μ of size p (here $p = 4$) with

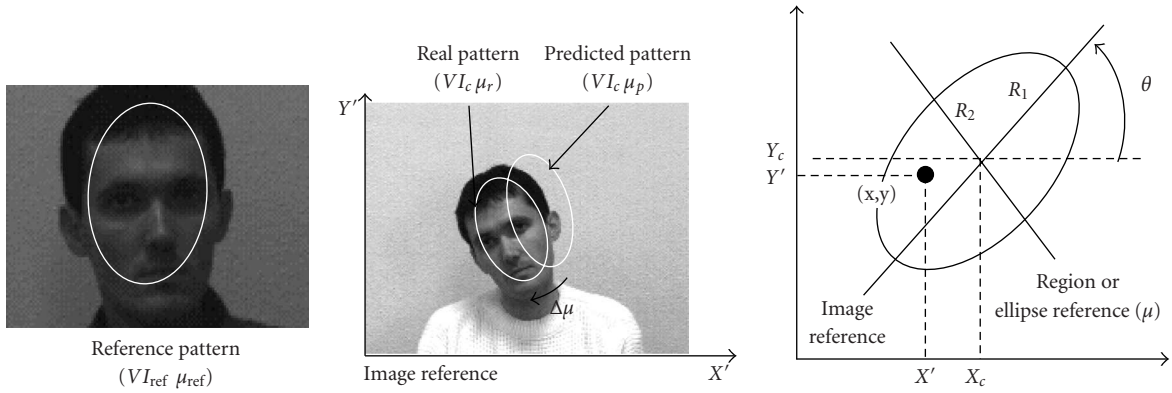


Figure 20: Tracking principle: tracking a 2D reference pattern.

$\mu = (X_c, Y_c, R_1, \theta)^t$ and $R_2 = k * R_1$. Also, we note that μ_r is the vector of parameters of the real position of the pattern in the image, μ_p is the predicted vector of parameters, and the difference $\Delta\mu = \mu_r - \mu_p$. Moreover, the visual pattern inside the predicted ellipse is sampled to give the current-shape vector VI_c of size N (here $N = 170$). The shape vector of the tracked reference pattern is denoted VI_{ref}. We denote by $\Delta VI = VI_{\text{ref}} - VI_c$ the difference between these two gray-level vectors. It is interesting to find out if we can determine $\Delta\mu$ knowing ΔVI. In that case, in measuring the difference ΔVI between the tracked reference pattern and the predicted current pattern, we are able to determine the correction $\Delta\mu$ to update the prediction and obtain the real position of the pattern $\mu_r = \mu_p + \Delta\mu$ with $\Delta\mu = A\Delta VI$ (Figure 20). We thus formulate the tracking problem, as the determination of an offset vector $\Delta\mu$, by supposing that the position variations of the face in the image correspond to the parameter variations of a geometrical transformation. In our particular case, we use a rigid affine transformation where the parameters of the ellipse are the parameters of the geometrical transformation (Figure 20). Here, A is an *interaction matrix* $(p \times N)$ corresponding to the computation of a linear relation between a set of gray-level differences ΔVI and a correction $\Delta\mu$ of the parameters of the vector μ during an off-line learning stage.

3.4. Computation of interaction matrix A for a given reference pattern

The computation of the interaction matrix A is done during an off-line learning stage. One of the originalities of the proposed computation method is that we do not use Jacobian matrices of the reference view as used in the work of Hager and Belhumeur [17] or Dellaert and Collins [18]. We estimate the matrix A by least-square minimisation using an algorithm based on a singular value decomposition. We observed that in this case, the field of convergence was much larger [9].

This matrix makes it possible to update the parameters of the rigid affine transformation during the face tracking. At the beginning of this stage, an ellipse is aligned manually by the user on the reference pattern, then sampled in order to obtain the reference shape vector VI_{ref}. This initialisation also enables us to fix the ratio k between the two radii of the ellipse ($k = R_2/R_1$). The position of the ellipse is perturbed around its position of reference while the coefficient k remains constant (see Figure 21). For each perturbation i, the parametric variations of the transformation $\Delta\mu^i = (\Delta X_c^i, \Delta Y_c^i, \Delta R_1^i, \Delta\theta^i)^t$ as well as the values of the sampled current pattern VI_c^i inside the ellipse are memorised. Thus, if we take M measurements of this type for N sampled points in a shape vector, it is possible to estimate A as soon as $M \geq N$. In practice, we conduct 500 parametric perturbations and samplings of the pattern on 170 points. Therefore, an overdetermined system of M equations in N unknowns has to be solved for each parameter of the transformation (four in our case). Actually, the resolution of a single linear system, or more exactly, the computation of only one pseudoinverse matrix, is necessary.

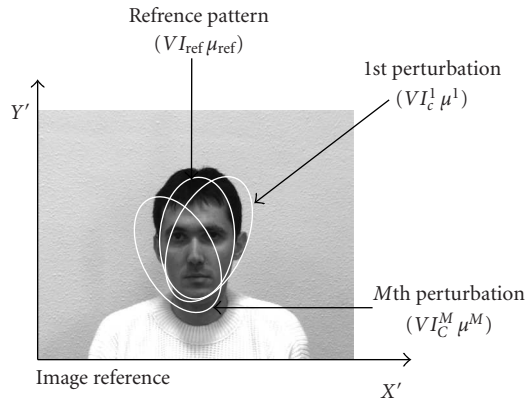


Figure 21: Computation of the interaction matrix A: perturbations of the ellipse parameters.

Indeed, we note $\Delta VI^j = (\Delta i_1^j, \Delta i_2^j, \ldots, \Delta i_N^j)^t$, the difference vector between the reference pattern VI_{ref} and the pattern corresponding to the jth perturbation VI_c^j and the interaction matrix: $A = (AX_c, AY_c, AR_1, A\theta)^t$.

To obtain line $A\theta$ of the interaction matrix relative to the orientation of the ellipse, we write down the following linear system:

$$\begin{pmatrix} \Delta i_1^1 & \cdots & \Delta i_N^1 \\ \vdots & \ddots & \vdots \\ \Delta i_1^M & \cdots & \Delta i_N^M \end{pmatrix} \begin{pmatrix} A\theta_1 \\ \vdots \\ A\theta_N \end{pmatrix} = \begin{pmatrix} \Delta\theta^1 \\ \vdots \\ \Delta\theta^M \end{pmatrix}, \tag{1}$$

which can be shortly expressed as

$$M_{\Delta VI} * A\theta = \Delta\theta. \tag{2}$$

The final solution is then obtained by

$$\begin{aligned} A\theta &= \left(M_{\Delta VI}^t M_{\Delta VI}\right)^{-1} M_{\Delta VI}^t \Delta\theta = M_{\Delta VI}^+ \Delta\theta, \\ AX_c &= M_{\Delta VI}^+ \Delta X_c, \\ AY_c &= M_{\Delta VI}^+ \Delta Y_c, \\ AR_1 &= M_{\Delta VI}^+ \Delta R_1. \end{aligned} \tag{3}$$

The matrix $M_{\Delta VI}^+$ is the so-called pseudoinverse of $M_{\Delta VI}$.

3.5. Switching between reference views

Various reference appearences of faces can be tracked by switching between views stored in our image base. In order to do that, we compare permanently the quadratic errors of the tracking results between the currently tracked reference pattern and its nearest neighbours in the learning base. The reference pattern giving the smallest error between its shape vector and the current pattern sampled inside the corrected ellipse will be considered as the reference pattern to be tracked in the next image. But, it is necessary to add an intermediate stage to compute the corrections on the predicted position of the face in the image for each of the reference patterns close to the currently tracked reference pattern in the collection of 2D views. For that, during the off-line learning stage, we compute, for the intermediate views placed in the middle of the nearest reference patterns in the collection of 2D images, the different positions of the ellipse corresponding to the tracking results for each of the reference patterns (Figure 18). We choose, in particular, these intermediate images because we suppose that the change of the reference pattern, during the on-line tracking stage, happens around these variations of the appearance. During the tracking phase, these different results are used with the predicted parameters of the ellipse corresponding to the currently tracked reference pattern to compute the predicted positions of the face for the previous and next reference patterns in the current image before correction and estimation of the associate quadratic error. In this additional stage, we use scale and reference changes.

4. PARALLELISATION OF THE 3D FACE-TRACKING ALGORITHM USING ALGORITHMIC SKELETON NESTING

4.1. Principle

Here we only consider the second stage of the tracking algorithm as a candidate for parallelisation purposes. This stage can be parallelised as follows.

(1) The computations on the previous, current, and next reference patterns can be done independently and in parallel (see Section 3.2 for details about reference patterns). These computations are independent and involve a similar workload. The first parallelisation level, therefore, matches a first data-parallel skeleton (skeleton A in Figure 22). This skeleton will be used to carry out the comparison of all the reference patterns in parallel.

(2) The matrix multiplication step involved in the processing of each reference pattern can be further parallelised. This second parallelisation level matches another nested data-parallel skeleton (skeleton B in Figure 22).

It is important to realise that these two parallelisation levels cannot be merged into a single one because the three inner matrix multiplications cannot be merged into a single matrix multiplication. More precisely, three different interaction matrices (named A in the previous section) and three different gray-level vectors (named V diff) have to be considered. For each reference pattern, only one vector must be multiplied by one matrix. So merging the three matrices into a single one and the three vectors into a single one will not only result in extra computing time but will also produce erroneous results.

At both levels, the data-parallel computations are very regular (i.e., they process data sets whose size is known at compile time and their complexity does not depend on the values of these data). This makes them perfect candidates for an implementation with a SCM skeleton. The parallel structure of the application is therefore made up of two nested SCM skeletons, the inner SCM skeleton playing the role of the *compute* function for the outer skeleton.

The *split* function of the outer SCM skeleton (skeleton A in Figure 22) selects the three tested reference patterns and transmits the pattern numbers to every *split* function of the inner SCM skeleton (skeleton B in Figure 22). We assume here that the input image is available in the memory of all nodes. The *split* function of the inner SCM skeleton samples its pattern within the predicted position of the ellipse in the image, computes the gray-level difference between the visual pattern and the tested reference pattern, and transmits this vector, along with the associated interaction matrix and the line number of the matrix to every inner *compute* function. These functions estimate the correction of one ellipse parameter. The results are sent to the inner *merge* function, which computes a new gray-level difference using the corrected ellipse and the associated quadratic error, and sends the result to the *merge* function of the outer SCM skeleton. The sent results include the corrected ellipse parameters, the quadratic error, and the number of the associated reference pattern. The outer *merge* function eventually selects the reference pattern giving the smallest quadratic error and hence the real position of the tracked pattern in the image.

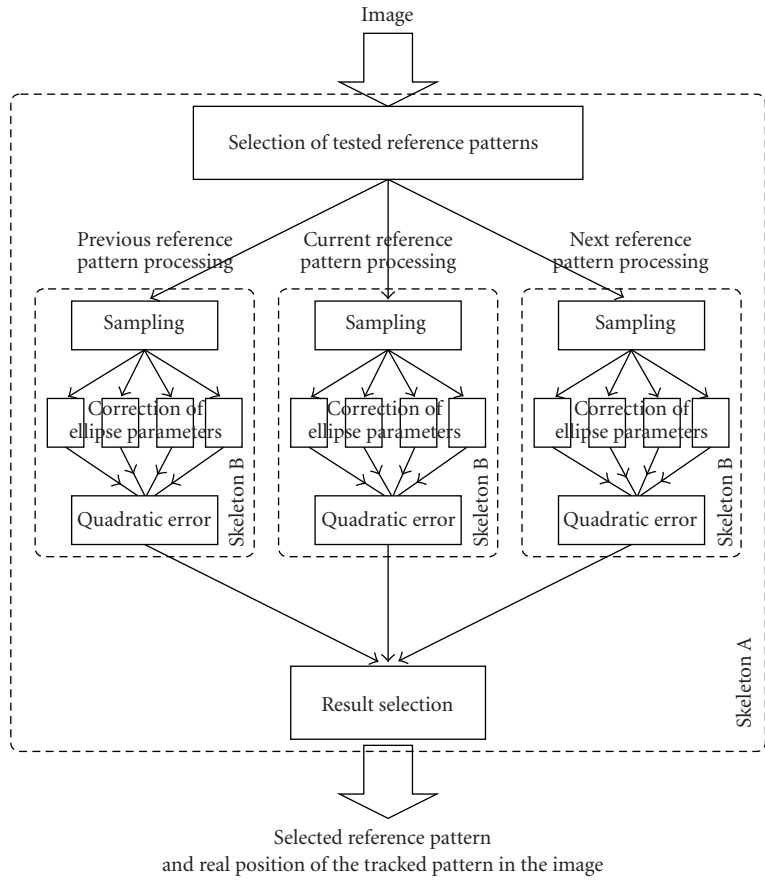


Figure 22: General parallel structure.

4.2. Implementation

Once the parallel structure of the application has been identified, the SKiPPER-II environment can be used to obtain a parallel implementation. From a programmer's point of view, this involves

(1) expressing this parallel structure using some kind of description language (i.e., specifying which skeletons are used and in what order),
(2) providing the application-specific sequential functions to be used as arguments to the skeletons.

In previous SKiPPER versions, expressing the parallel structure was carried out using a subset of the Caml language [4]. The same approach is intended to be used for SKiPPER-II. In this case, the intermediate description of the application (as a tree of TF/II skeletons) will be generated by a modified version of the Camlflow tool [19]. In the current version of SKiPPER-II, this step is still handled manually, that is, the intermediate description is provided by the programmer in the form of a C descriptor which can be used directly by the kernel. This descriptor encodes, in the form of a C array, the tree of TF/II skeletons that matches the skeletal structure of the application. For the tracker application, the corresponding descriptor[5] is given in Algorithm 4 (Figure 23 recalls the skeletal structure of the application). There is one line per skeleton. On each line,

(i) the first column is the skeleton ID (for reference),
(ii) the second column indicates the skeleton "continuation," that is, whether its results must be sent to an upper level or to another skeleton at the same level,[6]
(iii) the third column tells whether this skeleton acts as a slave (i.e., is nested) or as a master,

```
#define SKL_NBR 2
SK2_Desc  app_desc [SKL_NBR] =
{
  { SKO, END_OF_APP, MASTER, SK1 },
  { SK1, UPPER, SLAVE, NIL }
};
```

Algorithm 4: Encoding the parallel structure of the tracker application. C encoding for SKiPPER-II.

[5]This is a slightly edited version—for readability—of the actual code.

[6]Here, END_OF_APP is a special case meaning that the current skeleton is the last one.

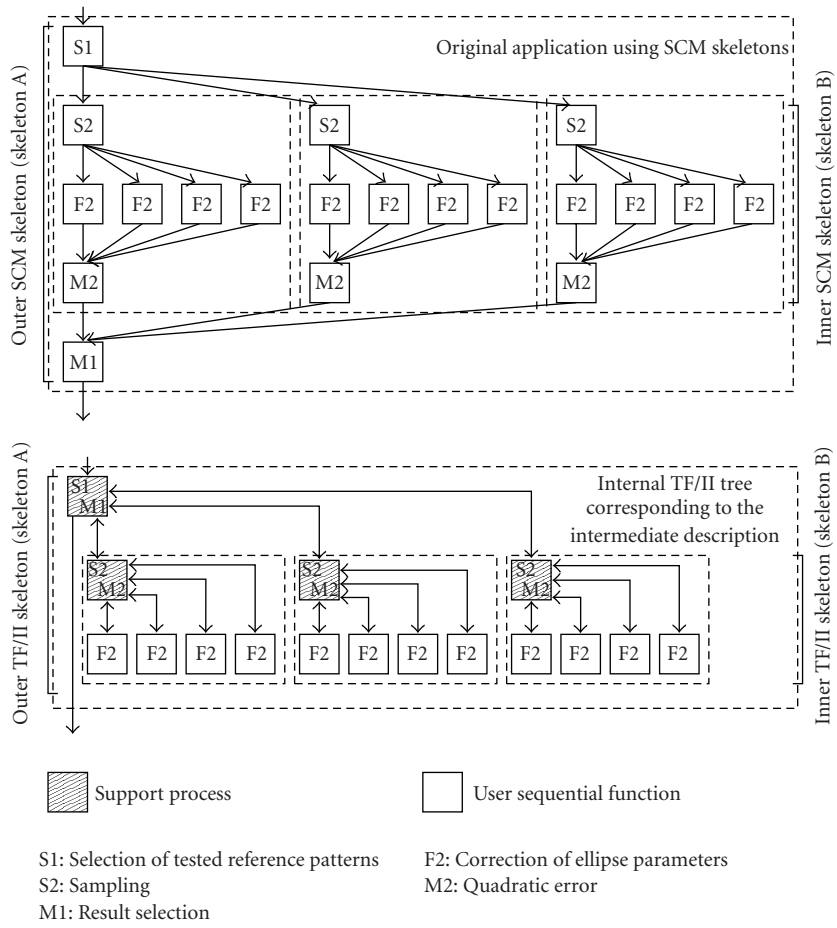


Figure 23: Encoding the parallel structure of the tracker application. Graphical representation.

(iv) the last column gives the ID of the destination skeleton.

The second step in using SKiPPER-II is providing the application-specific sequential functions. These functions—to be used as arguments to the specified skeletons—are written in C and must be "pure" functions (no side-effect, no reference to global variables or shared data). All nonatomic arguments[7] must be passed by address and all results must be returned by address. The prototypes of the sequential functions for the tracker application are given in Algorithm 5.

In the current implementation of SKiPPER-II, the programmer is also required to write a few lines of stub code to allow the application-specific sequential functions to be linked with the kernel code. The main role of this stub code is to alleviate the lack of support for data polymorphism in the C language (at the kernel level, all application-specific functions must have a uniform interface, in which all arguments and results are passed/returned as untyped buffers). This stub code is very systematic and repetitive (it essentially consists in packing/unpacking application-level data structures into/from kernel-level (char $\star$) arrays) and could be automated.

5. RESULTS AND DISCUSSION

The benchmark was performed on Intel Celeron Beowulf machine (32×533 MHz nodes, 100 Mbps switched Ethernet network). Figures 24 and 25, respectively, show the completion time of the algorithm and the relative speedup obtained in increasing the number of nodes for two quantities of sampled points in the elliptic area (170 and 373 sample points).

It must be noticed that using more than 170 sample is not giving better results in terms of tracking capabilities (using 170 points already provides a sufficiently robust tracking). This number was increased in order to further assess the performances of SKiPPER-II kernel, since it directly influences the computation/communication ratio.

The curves in Figures 24 and 25 exhibit three phases which can be related to the behaviour of the SKiPPER-II kernel.

The first phase is observed for 1 to 2 nodes (373 sampled points) or 1 to 3 nodes (170 sampled points): here the

[7] By *atomic* arguments, we mean those having nonstructured data types, such as int, float, and so forth.

```
S1(
  int          pattern_number,              /*I   */
  Ellipse      current_ellipse,             /*I/O*/
  int  **      tracker_number_to_test       /*  O*/
);
S2(
  Ellipse      current_ellipse,             /*I/O*/
  int          tracker_number_to_test,      /*I/O*/
  int      *   gray_level_vector_size,      /*  O*/
  float    **  gray_level_difference-vector,/*  O*/
  int      **  matrix_line_number,          /*  O*/
  float  ***   matrix                       /*  O*/
);
F2(
  Ellipse      current_ellipse,             /*I/O*/
  int          tracker_number_to_test,      /*I/O*/
  int          gray_level_vector_size,      /*I   */
  float    *   gray_level_difference-vector,/*I   */
  int          matrix_line_number,          /*I/O*/
  float    *   matrix,                      /*I   */
  float    *   correction                   /*  O*/
);
M2(
  Ellipse      current_ellipse,             /*I   */
  int          tracker_number_to_test,      /*I/O*/
  int          matrix_line_number,          /*I   */
  float        correction                   /*I   */
  float    *   quadratic_error,             /*  O*/
  Ellipse  *   corrected_ellipse,           /*  O*/
);
M1(
  Ellipse      current_ellipse,             /*I   */
  int          tracker_number_to_test,      /*I   */
  float        quadratic_error,             /*I   */
  Ellipse  *   final_current_ellipse,       /*  O*/
  int      *   final_pattern_number         /*  O*/
);
```

Algorithm 5: Signature of the application-specific sequential functions for the tracker application.

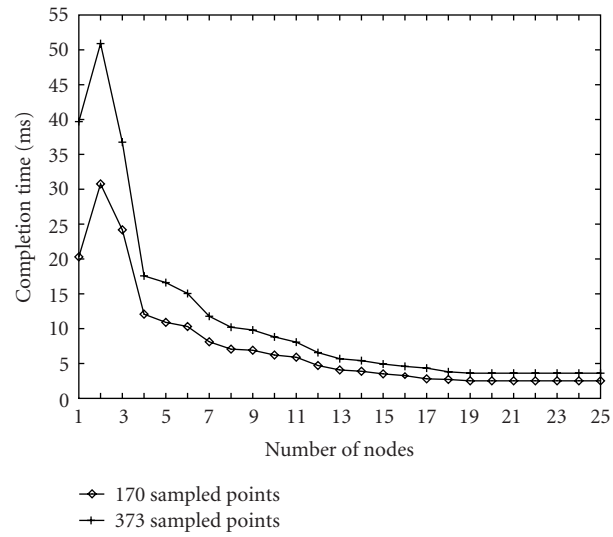


Figure 24: Completion time for 170 and 373 sampled points for 3D face-tracking algorithm parallelisation on an Intel Celeron Beowulf machine.

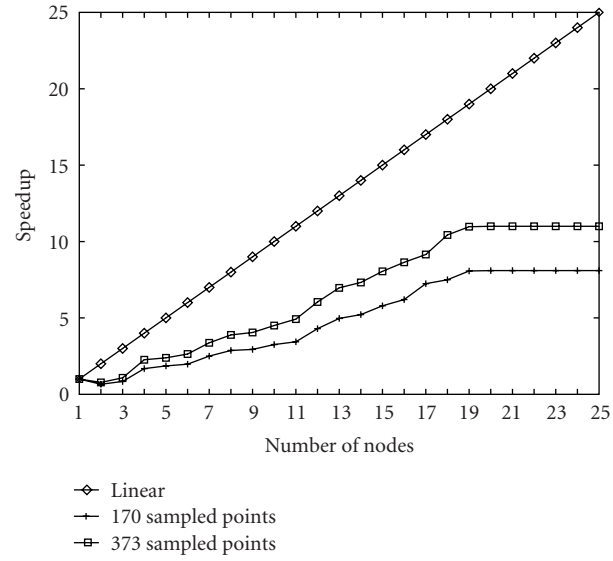


Figure 25: Speedup for 170 and 373 sampled points for 3D face-tracking algorithm parallelisation on an Intel Celeron Beowulf machine.

completion time is higher in the multiprocessor case. This negative speedup can be explained by the way the outer SCM skeleton is deployed on the network. In fact, with the runtime mechanism presented in Section 2.4, using two nodes does not provide more computing power but only creates communications. This is because one of the nodes is used as a dispatching process and is not performing useful computation at all. When the computation versus communication is small (as in the 170 points case), this effect can even be observed with 3 nodes because the time to communicate data to the two nodes doing computations destroys the potential gains of performing theses computations is parallel.

From 4 to 19 nodes, performance increases with the number of nodes (though not linearly). This phase corresponds to the deployment of the inner SCM skeleton, which performs the vector-matrix multiplications in parallel.

The last phase starts with 19 nodes. Here, increasing the number of computing nodes does not improve performance. This can be explained by the fact that the each SCM skeleton encapsulates a fixed data-parallel strategy. The maximum efficiency is reached when the number of available processing

nodes matches this fixed parallelism degree.[8] When the number of available nodes is higher (19–32), no further parallelism can be exploited and hence efficiency decreases. When the number is smaller (4–18), the kernel sequentialises some processings on some nodes (thus providing a form of "virtualisation" mechanism).

The relatively poor results in terms of efficiency can be more generally explained by the relatively small computation versus communication ratio of the parallel version. As a matter of fact, the sequential version of the algorithm was already very efficient because of very few intensive computing stages. This is especially true for the inner parallelisation level because the matrix multiplication is only a $(p \times N)$ matrix multiplied by the gray-level vector of size N.

6. CONCLUSION

This paper presents a skeleton-based parallel programming environment supporting skeleton nesting and the parallelisation of a realistic image processing application using this latest capability. As far as we know, the work described here is one of the few reported experiments showing the application of skeleton nesting facilities for the parallelisation of a realistic application, especially in the area of image processing. Indeed the 3D face-tracking algorithm cannot be entirely parallelised without this kind of skeleton combination. This is due to the fact that the different parallelisation levels cannot be merged into a single one and have to be handled by separate nested skeletons.

However, this work also shows that the run-time mechanism used in SKiPPER-II entails a significant performance penalty, especially when the computation versus communication ratio is low. In this case, skeleton nesting is not a transparent operation in terms of efficiency.

As for methodological aspects, we noticed that the parallelisation of the tracking algorithm only required three to four working days. This time was mainly dedicated to select the right parallelisation structure (which skeletons and how are they connected), and subsequently to split the original algorithm into computing functions to plug into the skeletons (the original user's functions have to be splited and their interface rewritten). Concerning parallelisation choices for the 3D face-tracking algorithm, we think that after this experiment, it could be interesting to parallelise the first stage of the algorithm, although it is normally an "off-line" stage. The reason is that decreasing the completion time will bring the opportunity to use all of the processing stages in an "on-line" way in order to use the tracking algorithm for multitarget tracking purposes, like multivehicle tracking [20]. In this case, speeding up this stage could allow the application to learn reference patterns of vehicles on the fly and hence allow it to adapt itself to the road environment without relying on a large prebuilt database of patterns.

[8] For the outer SCM skeleton, the optimal number of nodes is 4 (3 for computing, one for dispatching); for the inner one, this number is 5 (4 + 1).

ACKNOWLEDGMENTS

The authors would like to acknowledge the support of the European Commission through Grant number HPRI-1999-CT-00026 (the TRACS Programme at EPCC, Scotland) and to thank the staff members of the Department of Computing and Electrical Engineering, the Heriot-Watt University, Edinburgh, Scotland. The authors would also like to thank Santosh Anand, a Graduate from IIT Kampur, India, for his careful review of the paper's English.

REFERENCES

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

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

[3] J. Sérot, D. Ginhac, and J.-P. Dérutin, "Skipper: a skeleton-based parallel programming environment for real-time image processing applications," in *5th International Conference on Parallel Computing Technologies (PACT '99)*, V. Malyshkin, Ed., vol. 1662 of *Lecture Notes in Computer Science*, pp. 296–305, Springer-Verlag, Petersburg, Russia, September 1999.

[4] J. Sérot, D. Ginhac, R. Chapuis, and J.-P. Dérutin, "Fast prototyping of parallel-vision applications using functional skeletons," *Machine Vision and Applications*, vol. 12, no. 6, pp. 271–290, 2001.

[5] R. Coudarcher, J. Sérot, and J.-P. Dérutin, "Implementation of a skeleton-based parallel programming environment supporting arbitrary nesting," in *6th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'01)*, F. Mueller, Ed., vol. 2026 of *Lecture Notes in Computer Science*, pp. 71–85, Springer-Verlag, San Francisco, Calif, USA, April 2001.

[6] M. Hamdan, G. Michaelson, and P. King, "A scheme for nesting algorithmic skeletons," in *Proc. 10th International Workshop on Implementation of Functional Languages (IFL '98)*, C. Clack, T. Davie, and K. Hammond, Eds., pp. 195–212, London, UK, September 1998.

[7] G. Michaelson, N. Scaife, P. Bristow, and P. King, "Nested algorithmic skeletons from higher order functions," *Parallel Algorithms and Applications*, vol. 16, no. 2-3, pp. 181–206, 2001, Special issue on high level models and languages for parallel processing.

[8] M. Hamdan, *A Combinational framework for parallel programming using algorithmic skeletons*, Ph.D. thesis, Heriot-Watt University, Department of Computing and Electrical Engineering, Edinburgh, UK, January 2000.

[9] F. Jurie and M. Dhome, "A simple and efficient template matching algorithm," in *Proc. 8th IEEE International Conference on Computer Vision (ICCV '01)*, vol. 2, pp. 544–549, Vancouver, BC, Canada, July 2001.

[10] M. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation*, Pitman/MIT Press, London, UK, 1989.

[11] M. Cole, "Algorithmic skeletons," in *Research Directions in Parallel Functional Programming*, K. Hammond and G. Michaelson, Eds., pp. 289–303, Springer, UK, November 1999.

[12] J. Sérot, "Embodying parallel functional skeletons: an experimental implementation on top of MPI," in *3rd Intl Euro-Par Conference on Parallel Processing*, C. Lengauer, M. Griebl, and S. Gorlatch, Eds., pp. 629–633, Springer, Passau, Germany, August 1999.

[13] N. Scaife, *A dual source parallel architecture for computer vision*, Ph.D. thesis, Heriot-Watt University, Department of Computing and Electrical Engineering, Edinburgh, UK, May 2000.
[14] T. Grandpierre, C. Lavarenne, and Y. Sorel, "Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors," in *Proc. IEEE 7th International Workshop on Hardware/Software Codesign (CODES '99)*, pp. 74–79, Rome, Italy, May 1999.
[15] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel Computing*, vol. 22, no. 6, pp. 789–828, 1996.
[16] R. Coudarcher, *Composition de squelettes algorithmiques: application au prototypage rapide d'applications de vision*, Ph.D. thesis, LASMEA, Blaise-Pascal University, Clermont-Ferrand, France, December 2002.
[17] G. D. Hager and P. N. Belhumeur, "Efficient region tracking with parametric models of geometry and illumination," *IEEE Trans. Pattern Anal. Machine Intell.*, vol. 20, no. 10, pp. 1025–1039, 1998.
[18] F. Dellaert and R. Collins, "Fast image-based tracking by selective pixel integration," in *International Conference on Computer Vision Workshop on Frame-Rate Vision*, Corfu, Greece, September 1999.
[19] J. Sérot, "Camlflow: a caml to data-flow translator," in *Trends in Functional Programming. Volume 2*, S. Gilmore, Ed., pp. 129–141, Intellect Books, Bristol, UK, 2001.
[20] F. Marmoiton, F. Collange, P. Martinet, and J.-P. Dérutin, "A real time car tracker," in *Proc. International Conference on Advances in Vehicle Control and Safety (AVCS '98)*, pp. 282–287, Amiens, France, July 1998.

Rémi Coudarcher is a computer science engineer. He graduated from ISIMA, France, and received his Ph.D. degree in computer science from Blaise-Pascal University, France. He has been a post-doc fellow at INRIA, France. His research interests include parallel image processing, and methodologies for parallel and distributed programming and grid computing.

Florent Duculty is an engineer in electrical science. He graduated from CUST, France, and has a Ph.D. degree in computer science from Blaise-Pascal University, France. His research interests include image processing applied to robotics motion by visual servoing.

Jocelyn Serot is an Associate Professor in electrical engineering at Blaise-Pascal University, France. His research interests are in the development and use of methodologies for parallel programming, especially in the field of image processing.

Frédéric Jurie is a Researcher at CNRS, France. His researches are concerned with image processing, especially movement and object detection, and image recognition from appearance.

Jean-Pierre Derutin is a Full Professor in electrical engineering at Blaise-Pascal University, France. His research interests are in the design of dedicated and embedded parallel computers for real-time image processing.

Michel Dhome is a Director of Research at CNRS and Codirector of the LASMEA Laboratory, France. His researches include image processing for automatic recognition and robotics.