MVA '98 IAPR Workshop on Machine Vision Applications, Nov. 17-19, 1998, Makuhari, Chiba, Japan

13—11

Fast Prototyping of Image Processing Applications Using Functional Skeletons on a MIMD-DM Architecture

Dominique GINHAC, Jocelyn SEROT, Jean Pierre DERUTIN
Laboratoire des Sciences et Matériaux pour l'Electronique et d'Automatique
Université Blaise Pascal de Clermont Ferrand
UMR 6602 CNRS
F-63177 Aubière Cedex France

E-mail: (ginhac,jserot,derutin)@lasmea.univ-bpclermont.fr

Abstract

In this paper, we assess the applicability of the *skeleton-based* approach to portable parallel programming within the *vision* application domain. Four algorithmic skeletons for low and intermediate level image processing are proposed. For each skeleton we give an architecture-independent *executable* specification, a parallel implementation template as a process network and a performance model. These skeletons are used to build a programming environment dedicated to the fast prototyping of embedded vision applications, and mostly built from existing software components. Several examples are provided to illustrate the concepts and tools introduced, including a real-time vehicle detection and tracking application.

1 Introduction

Although sequential computers gain more and more power, complex image processing applications still require dedicated parallel architectures in order to cope with real-time constraints.

However, programming such parallel systems remains a non trivial task, leading to long development cycles. As a result fast prototyping and efficiency are often seen as incompatible goals.

We aims at conciliating these two goals by providing a software environment dedicated to the fast prototyping of real time vision algorithms on MIMD-DM platforms. This software is based upon the concept of *algorithmic skeletons*. Skeletons are higher-order program constructs which encapsulate recurring forms of parallel program structures and hide their low-level implementation details.

In this paper, we present the most important features of this environment and demonstrates its usefulness through a real size case study, namely the development of an application performing connected component labelling on digital video streams.

2 Algorithmic Skeletons

Within our application domain — low and intermediate level image processing — a retrospective analysis of existing implementations on MIMD-DM platforms shows that most of parallel applications are actually built upon a limited number of recurring *patterns*. Each of these patterns can be seen as a fixed communication harness embedding a set of user specific sequential functions. This allows for abstracting them into higher-order, reusable parallel constructs the parameters of which are the user sequential functions. The semantics of these constructs — called *algorithmic skeletons* after Cole [2] — can be captured using *two* definitions [8] :

- a *declarative* one, which gives the meaning of a skeleton in an architecture-independent, purely *applicative* manner. This definition is classically written using a functional language like ML[5].
- an *operational* one, which details its actual implementation on a given platform in terms of architecture-specific parallel facilities (message-passing calls, shared-memory access, ...).

Skeletons form the basis of the so-called *restricted* parallel computation models [9] [7] in which the programmer's task is to select and compose instances of pre-defined skeletons, rather than dealing with low-level, error-prone parallel constructs. These models offer a way of conciliating requirements for fast prototyping (the declarative abstract meaning of skeletons promotes instant re-usability) and efficiency (their implementation, written once and remaining hidden, can be carefully tested and optimised).

A key point in any skeletal programming model lies in the choice of a correct set of skeletons. (see for instance [1] for a review of proposals). In our case, this choice was made on the basis of a careful retrospective analysis of existing image processing applications on MIMD-DM architectures (especially the TRANSVISION [4] platforms, for which we had a large corpus of working, hand-coded parallel applications).

Examples of algorithms involved are:

- low-level pre-processing (filtering, edge detection, etc.),
- spatio-temporal operators (Markov field analysis, etc.),
- extraction of geometric primitive and/or perceptual groupings,
- detection and tracking of mobile objects in real scenes.

This retrospective abstraction process soon drew out four broad classes of skeleton-level parallel constructs

- Constructs 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 subimages, each subimage is processed independently with the same function, and the processed subimages are reassembled to form the output image.
- Constructs devoted to the extraction of mid-level features from iconic data. These are also built on a data-parallel, geometric decomposition scheme, but the processing of each subimage produces specific features instead of subimages, hence the need for a specific **merge** function.
- Constructs 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 cannot be predicted.
- Constructs 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[1]. An important subcase of this is when processing of the ith image of the stream depends on results computed on images $i-1, ..., i-k$.

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

The first class can be viewed as a specific instance of the second, in which the **merge** function reduces to a purely geometric composition operation. We also judged that both the *data* and *task farming* control abstractions deserved their own skeleton. Hence the following four "elementary" skeletons that will make the basis of our programming environment: SCM (Split, Compute and Merge), DF (Data Farming), TF (Task Farming), and ITERMEM (Iterate with Memory).

The four basic skeletons are :

- The SCM skeleton (Split, Compute and Merge) encompasses most of regular, data-parallel strategies in which the input data is divided into a fixed number of partitions and each one is processed by a different processor. The final output is obtained by combining the results computed on each partition.
- The DF (Data Farming) skeleton, devoted to irregular data-parallelism, is an abstraction of the processor farm model in which a master dynamically dispatches data packets to a pool of workers and accumulates results until each input data is processed.
- The TF (Task Farming) skeleton can be seen as 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.
- The ITERMEM skeletons is used whenever the stream-based model of computation has to be made explicit, for example when computations on the nth image depends on results computed on the n-1th. Such "looping" patterns are very common in tracking algorithms, based upon system-state prediction

For the sake of brevity, only the SCM skeleton is illustrated in the sequel.

3 The software environment

The different components of our skeletal programming environment are depicted in figure 1. The source program is a functional specification of the algorithm, in which all parallelism is made explicit by means of composing instances of the above-mentioned skeletons, each instance taking as parameters user-specific sequential functions written in C. A custom ML compiler first turns this specification into a data-flow graph, with nodes associated to user computing functions and/or skeleton control processes, and edges indicating communications. This graph of processes has then to be mapped onto the target architecture, which is also

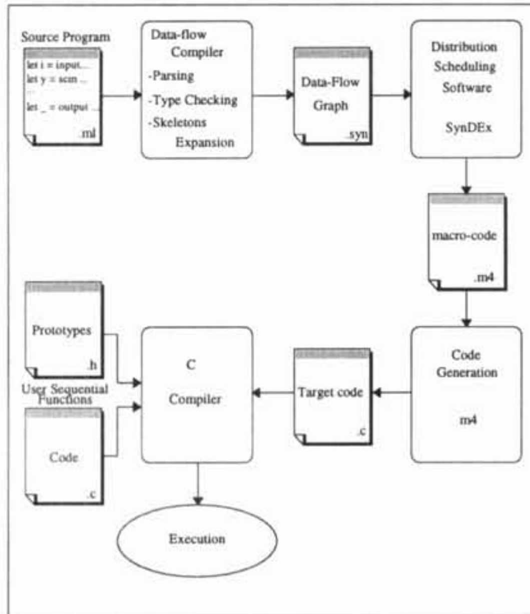

Figure 1: The skeleton-based programming environment

described as a graph, with nodes associated to processors and edges representing communication channels. This task, which involves finding a static distribution of processes onto processors and a mixed static/dynamic scheduling of communications onto channels is handled by a third-party software called SynDEx[10]. SynDEx's output is as set of processor-independent programs (m4 macro-code, one per processor). These macro-codes are finally expanded into compilable code by taking account the actual target language and message-passing facilities (C + read/write instructions on transputer links in our case).

4 First results

The skeletal programming system presented in section 3 has been first validated on a application performing connected component labelling on 25Hz, 256x256 digital video streams.

The connected component labelling (CCL) is a "classic" algorithm for which efficient parallel implementations have been extensively searched, both for SIMD[3] and MIMD[6]. It consists in identifying and labelling contiguous regions within a binary image. It can be used for instance to separate candidate objects after a region extraction was completed. Classical algorithms relies on three steps:

1. local pre-labelling: temporary labels are attributed to pixels using a 2x2 L-shaped mask and forwarded following 4 (or 8) connectivity.
2. labelling conflicts (typically arising from U-shaped patterns) are detected and used to build a list of label equivalences.
3. the image of temporary labels is corrected by means of a lookup table built from the final list of equivalences.

The functional specification of this algorithm, using two SCM skeletons is given fig 2.a. `rb`, `pre`, `toem`, `corr` and `br` are the user sequential functions (written separately in C). The first SCM uses a partition of the input image into n horizontal strips to compute the temporary labels and the final lookup table, taking into account the equivalence conflicts arising at strip boundaries. The second SCM skeleton simply applies this lookup table to produce the result image, using the same geometric partition.

The corresponding data-flow graph (generated by th compiler) and its distribution/scheduling onto a 4-processor ring architecture (computed by SynDEx) are shown in fig 2b and 2c.

Measured execution times are about 80 ms for a 256x256 image and 4 processors, with a relative speedup of 2.7). The main lesson drawn from that preliminary test-bench, however, was not on raw performances but on the effective prototyping capabilities of the software environment: in that case, the programmer's work boiled down to writing 3 pure sequential C functions: `pre`, `toem` and `corr` (`rb`, `br`, `input` and `disp` are provided by the environment) and the skeletal specification showed in fig 2a. Underlying implementation details (including process placement, communication scheduling, buffer allocation, provision for deadlock avoidance, ...) were all transparently handled by the compiler. The result is that it took less than one day to get a working implementation of the algorithm on the target platform (regardless of the number of processors).

5 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-data-flow compiler and an existing distributing/scheduling tool to turn a high-level specification into executable functional code. Preliminary results of this system — illustrated here with a complete application performing CCL on a multi-transputer real-time vision machine — are encouraging since they show a dramatic

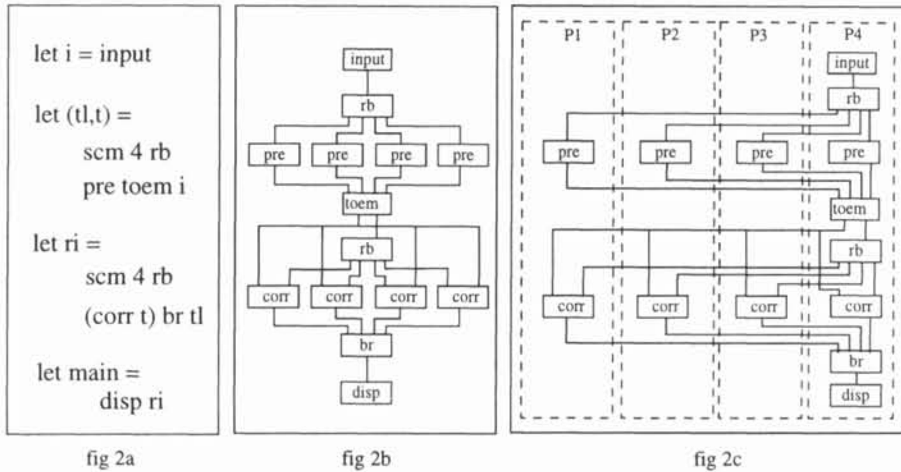

Figure 2: The target implementation

reduction in development time while keeping satisfactory performances.

Work under progress includes further validation of the approach through the implementation of more complex applications (involving several distinct skeletons and inter-skeletons optimisations) and streamlining the system in order to provide a complete integrated solution that could be used by programmers with little or no previous experience in parallelism.

References

[1] Duncan K.G. Campbell. Towards a Classification of Algorithmic Skeletons. Research Report YCS-276, Department of Computer Science, University of York, December 1996.

[2] M. Cole. *Algorithmic skeletons: structured management of parallel computations*. Pitman/MIT Press, 1989.

[3] R. Cypher, J.L.C. Sanz, and L. Snyder. Algorithms for image componed labeling on SIMD mesh connected computers. In *IEEE Trans. Computers*, volume 32, February 1990.

[4] 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 (CAMP 93)*, pages 410–420, New-Orleans, USA, December 93.

[5] R. Milner, Tofte M., and Harper R. *The Definition of Standard ML*. MIT Press, 1984.

[6] H.T. Nguyen, K.K. Jung, and R. Raghavan. Fast parallel algorithms : from images to level sets and labels. In *Parallel Architectures for Image Processing*, volume 1246, pages 162–176, 1990.

[7] S. Pelagatti. *A methodology for the development and the support of massively parallel programs*. PhD thesis, Universita degli studi di Pisa, Dipartimento di informatica, March 1993.

[8] J. Sérot. Embodying parallel functional skeletons : an experimental implementation on top of MPI. In *Proceedings of Europar 97*, pages 629–633, Passau, Germany, August 1997.

[9] D. B. Skillicorn. Architecture-independent parallel computation. *IEEE Computer*, 23(12):38–50, December 1990.

[10] Y. Sorel. Massively parallel systems with real time constraints. The "Algorithm Architecture Adequation" Methodology. In *Proc. Massively Parallel Computing Systems*, Ischia Italy, May 1994.