Patterns and Skeletons for Parallel and Distributed Computing

Fethi A. Rabhi
School of Information Systems
University of New South Wales, Sydney 2052 (Australia)
Email : `f.rabhi@unsw.edu.au`
and
Sergei Gorlatch
Technical University of Berlin
Sekr, FR 5-6, Franklinstr. 28/29
D-10587 Berlin (Germany)
Email: `gorlatch@cs.tu-berlin.de`

March 1, 2002

Contents

1 Skeleton Realisations from Functional Prototypes **1**

1.1 Functional Prototyping and Parallelism 1
1.2 Prototyping and Transformation 3
1.3 Prototyping Parallel Computer Vision Algorithms and Systems . 4
1.3.1 Overview 4
1.3.2 Skeleton-based Vision System 7
1.4 Towards Skeleton-based Compilers 9
1.4.1 Overview 9
1.4.2 PUFF 10
1.4.3 SkelML 11
1.4.4 EKTRAN 12
1.5 PMLS Compiler 13
1.5.1 Overview 13
1.5.2 HOF Synthesis 16
1.6 Case Study: Matrix Multiplication 17
1.7 Conclusions 22

Chapter 1

Skeleton Realisations from Functional Prototypes

Greg Michaelson
Department of Computing and Electrical Engineering, Heriot-Watt University, Riccarton, EH14 4AS
greg@cee.hw.ac.uk

Norman Scaife
Japanese Advanced Institute of Science and Technology, Asahidai 1-1, Tatsunokuchi, Nomigun, Ishikawa, 923-1211 Japan
norman@jaist.ac.jp

1.1 Functional Prototyping and Parallelism

Prototyping is an attractive general basis for experimentation with system design, where the design is refined in the light of prototype behaviour, leading in turn to prototype refinement. A prototype should be considerably cheaper to construct than the final system, and avoid commitment to fine implementation detail, while displaying the same gross input/output characteristics. Once the design has stabilised, the prototype provides a standard for the final system.

Prototyping is particularly valuable for parallel system development. Parallel programming is complex compared with sequential programming, and parallel resources tend to be expensive and scarce. It is often not practical to tie up a parallel facility during system development. In principle, a sequential prototype may provide both accurate indications of eventual parallel behaviour and a framework for parallel system development, where sequential prototype constructs are systematically replaced with parallel equivalents.

Functional languages are a strong basis for prototyping. The high degree of abstraction enables the construction of succinct prototypes which elide considerable implementation detail. Functional prototyping is well suited for parallel system development. The close relationships between functional and parallel constructs facilitate the derivation of parallel systems from prototypes. There

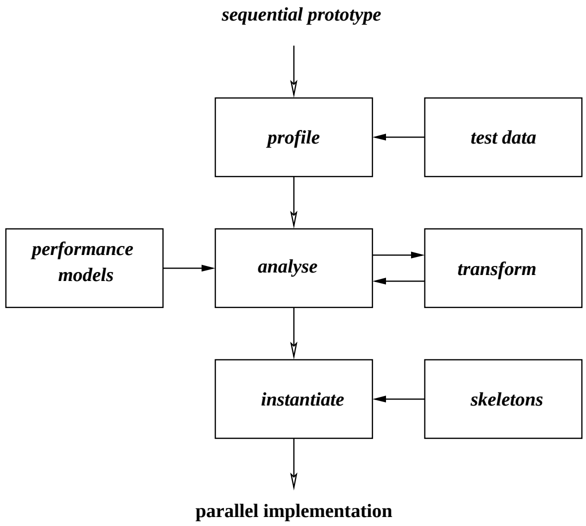

Figure 1.1: Purist skeletons methodology.

are rigorous formalisms for transforming functional constructs, with comparable effects on their parallel equivalents.

A central advantage of the use of a functional language is that, in principle, programs may be transformed without the need for re-profiling. In the absence of side effects and with appropriate granularity profiling information, overall costs should be derivable from transformed components of known costs.

We have focused on the correspondences between higher order functions (HOFs) and algorithmic skeletons. In principle, we propose a purist version of the skeletons credo, illustrated in Figure 1.1, utilising the following techniques to realise parallelism from prototypes:

- A sequential prototype should be analysed to determine processing and communication costs in HOF use;
- Costs should be fed through models for the equivalent algorithmic skeletons to determine if the HOFs may be realised efficiently as skeletons;
- Where no useful parallelism is found, the prototype should be transformed to try and optimise the processing/communication cost balance in the models;
- Where useful parallelism is predicted the HOFs should be realised as, possibly nested, algorithmic skeleton instantiations.

In practice, such purism is almost unrealisable. Analytic cost modeling for arbitrary recursive functions is undecidable and building realistic cost models for other than simple skeletons is hard. Furthermore, the transformation search space grows very rapidly, both with the number of different HOF transformation rules and with the depth of HOF nesting and composition in programs.

Programming constructs may be restricted to make analytic techniques feasible. In particular, data structures may be constrained to be shapely or HOF arguments may be limited to other HOFs and non-recursive functions. However, rather than compromising expressive power in the source language as a whole, we prefer to exploit parallelism in a general-purpose language but through a small number of HOFs, and to assess exploitable parallelism through empirical prototype profiling and instrumentation.

These choices are also problematic. It is very hard to decide what sequential profiling information is relevant to determining potential parallelism, or to provide convincing evidence for the general reliability of predictions based on sequential profiling. Profiling results are inevitably biased towards the chosen data sets. Generating and evaluating with data that represents a wide range of cases becomes as infeasible as the use of general heuristics for static program analysis.

Nonetheless, we have made good progress applying this approach to problems in parallel computer vision. These experiences closely informed the development of four parallelising compilers, where the above techniques are embodied to varying degrees in fully automated components.

The following section discusses our choice of prototyping language and basic transformations. We then provide a background to computer vision, and discuss the application of our approach in developing parallel vision algorithms and systems. Subsequent sections present the embodiment of our approach in parallelising compilers, focusing on the portable PMLS compiler for Standard ML.

1.2 Prototyping and Transformation

We use Standard ML (SML) as our prototyping language. SML is a mature functional language with a stable formal definition and a rich set of tools and libraries. SML is an imperative language with side-effecting assignable references. However, it has a clean, pure functional subset which we employ.

For prototyping, SML brings the well known benefits of contemporary functional languages: parametric polymorphism; pattern matching; first class functions; closeness to formalisms for proof and transformation. For parallel prototyping, furthermore, SML's strictness is better suited than the laziness of Haskell or Miranda as it enables predictable behaviour.

We focus on the `map` and `fold` HOFs, with explicit composition as the basis for identifying parallelism, shown in Figure 1.2. We use the standard infix notation `f o g` for `compose f g`.

Corresponding to our base functions, our transformations involve simple distribution and equivalences.

The *map/compose* transformation is used to group argument functions for composed `map`s where there is an unfavourable processing/communication bal-

```
fun map f [] = [] |
    map f (h::t) = f h::map f t;

fun foldr f b [] = b |
    foldr f b (h::t) = f h (foldr f b t);

fun foldl f b [] = b |
    foldl f b (h::t) = foldl f (f h b) t

fun compose f g x = f (g x);
```

Figure 1.2: Basic higher order functions.

ance for either of them:

(1) *map/compose* (map f) o (map g) $\Leftrightarrow$ map (f o g)

The *foldr/foldr-map* transformation is used where the `foldr` argument function is not associative, preventing parallelisation. Here, some parallelisation through `map` may still be possible:

(2) *foldr/foldr-map* foldr (fn h => fn t => f (g h) t) b l $\Leftrightarrow$
foldr f b (map g l)

The *map/foldr* transformation is a special case of *foldr/foldr-map* and is useful for comparing the behaviours of skeletons for `map` and `foldr`:

(3) *map/foldr* map f $\Leftrightarrow$
foldr (fn h => fn t => f h::t) []

The *partial application* transformation is a simple first attempt at lifting out common arguments to higher order functions for pre-computation before distribution:

(4) *partial application* hof (f x) $\Leftrightarrow$
let val x' = f x in hof x' end

1.3 Prototyping Parallel Computer Vision Algorithms and Systems

1.3.1 Overview

Computer vision is concerned with the extraction of information from representations of visual scenes. It is useful to consider computer vision as a three stage process.

Low-level processing usually manipulates images consisting of arrays of pixels. Typically, each pixel includes a measure of light intensity or of distance from a viewpoint. The image may be analysed to remove extraneous noise, for example where a pixel's intensity falls outside some upper and lower thresholds, and to select pixels that meet some adjacency criteria, for example being close to or at some angle to other pixels. Low-level processing is often based on convolutions where a window or mask is applied at each pixel location. Convolutions are regular data parallel but the processing at each pixel is usually light weight, requiring careful determination of appropriate granularity. In particular, apparent gains from nested parallelism in row/column processing may not be realisable and it may be more effective to base parallel processing on large sub-images rather than on individual pixels.

Intermediate level processing seeks groupings of pixels in images, for example lines or planes. Points are tracked across the image and intermediate representations are constructed. Intermediate-level processing is also often data parallel. For example, lines may be identified by edge tracking, and planes by region growing, where a new point is accepted if it's relationship to an already accepted point falls within some tolerance. All points may be initial candidates for line or plane identification but data parallelism is irregular as the extent of each line or plane is unknown. Dynamic task allocation and balancing become important here to ensure efficient use of parallel resources.

Finally, *high-level* vision attempts to match configurations of groupings from the original scene, for example lines meeting at corners or planes meeting at edges, with equivalent features in known objects, for example CAD models. High-level processing involves, at worst, an exponential search of combinations of features in the scene and in the model, seeking consistent correspondences. It may be necessary to transform either the scene or the model to give them consistent orientations and scales. Processing tends to be intensive but highly irregular, and may also require substantial movements of complex data structures.

The following sections discuss the parallelisation of a range of algorithms at all levels using functional prototyping, and the construction of larger vision systems from prototyped components. In all cases, the algorithms and systems were prototyped in Standard ML for parallel implementation in occam2 on a T800-based Meiko Computing Surface.

Canny Edge Detector

The Canny edge detector [8] is a low-level window-based operator that selects pixels in an image according to strength and adjacency criteria. Koutsakis' prototypes [18] was based on nested `maps`. Sequential profiling suggested a transformation to a single specialised `map` for processing whole rows of images to increase granularity. This was implemented as a linear process farm achieving a peak speedup of 6 on 7 processors on a 256×256 image.

HK Segmentation
HK segmentation [1] is an intermediate-level algorithm that seeks to fit a surface to an image in two stages. An initial coarse segmentation of the image is conducted based on the signs of the mean and the Gaussian surface curvatures. Seed regions are then chosen from which to grow surfaces satisfying a root mean squared error in fitting a bivariate polynomial to that region.

Clarkson's prototype [10] was rich in HOFs. Attention was focused on the central variance (`fold o fold`), smoothing (`map o map`) and surface labeling (`map o map`) modules. These were instrumented sequentially, showing that in all cases processing was considerably more costly than communication. The HOFs were grouped using `map`/`fold`/compose transformations and implemented through a generic linear farm skeleton, achieving good speedups:

function	farm size	speedup
variance	14	13.1
smooth	8	7.3
surface label	14	13.6

Hough Transform
The intermediate-level Hough transform [19] involves mapping features in an image into points in N-dimensional space. Scaife's Hough transform for straight lines [24] was prototyped as a pipeline of three `map`s. Sequential analysis on a range of test images revealed that of the three elements in the pipeline, one phase dominated the others by factors varying from 3.5-25:1 so the parallel implementation was based upon a farm of pipelines rather than a pipeline of farms. The resulting implementation gave a peak speedup of 8 on 12 processors with a maximum efficiency of 0.71 on 4 processors.

Perspective Inversion Algorithm
The Perspective Inversion Algorithm (PIA) [20] is another high-level technique for matching 3D models to 2D scenes. A model is projected into the same plane as the scene, and then rotated and translated to try and align its features with those in the scene. Such alignments are scored and the best is chosen.

Waugh's parallelisation of the PIA [32, 33, 30] was again based on `map` and compose. Sequential profiling suggested a poor balance in sub-components of a `map`ed composition which was transformed to a composition of `map`s. This enabled balance through the allocation of more processors to the slower sub-component. The use of partial application in the prototype to factor out common terms, corresponding to initial data distribution in the implementation, also helped optimise the processing/communication balance. Parallel behaviour was accurately predicted from the sequential profiling, giving speedup of 9 on 11 processors, with consistent scalability. Waugh also devised an $O(N^2M)$ pipeline balancing algorithm [32], improving on Bokhari's $O(N^3M)$ algorithm.

1.3.2 Skeleton-based Vision System

Prompted by the existence of components for low-level Canny edge detection, based on Koutsakis' work, and high-level model matching, based on Waugh's PIA, we constructed an entire intensity-based computer vision system [24]. With the development of the intermediate-level Hough component discussed above, these could be combined into a complete object recognition system for 2D scenes. The resulting system was a pipeline with three components; Canny, Hough and PIA. Processor allocation in the pipeline was based upon the timings for the individual prototypes and subsequently verified by experiment.

The performance of the resulting system was somewhat disappointing, with a peak speedup of 10.5 on 26 processors and a maximum efficiency of 0.35, as the individual farms were operating sub-optimally within the optimal configuration. Improving this would, however, require re-instrumenting the combined prototype. However, the development time for the prototype was impressively quick and the development time of the parallel implementation directly benefited from prototyping.

A much more ambitious system was subsequently developed using combined depth and intensity processing [28]. Depth and intensity information from the same image were explored independently but there was cooperation between these "routes" so each could benefit from relevant information established by the other to avoid duplicated or unnecessary computation. Here the skeletal prototyping method was applied to the low- and intermediate-level feature segmentation components.

One of the most successful aspects of the intensity-based system was the short development time of the prototype. In order to maximise this, as much algorithmic commonality between the depth and intensity routes as possible was used and this influenced the selection of suitable algorithms.

It became clear that the single, simple skeleton for `map`, used in the entire intensity-based system, would not be sufficiently expressive for the new system, where cooperation was required between components and widely different data sources. Instead, *specialised* skeletons would need to be developed during the course of building the system.

It was not known in advance how many or what kind of skeletons would be required by the final system. These were developed using two guiding principles; firstly, the minimum number of skeletons required to develop the system would be used and secondly, if a new skeleton was required to express an algorithm, adapting an existing skeleton was preferable to creating a new one from scratch. This created a tension between desirable methods of expressing parallelism and optimising the prototyping methodology. Developing a specialised skeleton for each application would lead to optimal parallel performance but we wished to reuse our skeletons as much as possible to minimise development time.

Upon completion of the system, we were left with four skeletons [29, 31]:

- Geometric Decomposition (GD). This is a decomposition over regular data, the data is divided equally among the available processors and the partial results are then combined. This was used extensively over low-level data. The prototype HOF is a functional `map`.

- Data Farming (DF). This is the familiar functional `map` implemented as a process farm and is useful in the presence of irregular data or computation.
- Task Farm (TF). A set of seed tasks are distributed among the workers. Processing each task can potentially engender further tasks which are then redistributed among free workers. This can be viewed in the prototype as a functional `map` within an *iterative* construct.
- GD with Interworker Communications (GDIW). This is a purely imperative implementation of the GD skeleton but incorporating a mechanism for direct communication between worker processors. The message passing can be simulated in the prototype.

The overall system used skeletons as follows:

Module	Skeletons
Canny edge detector	GD
Edge tracking	DF $\rightarrow$ DF
HK calculation	GD
HK segmentation	GDIW
Region growing	DF $\rightarrow$ TF

However, the completeness of this set of skeletons with respect to general computer vision is uncertain.

The performance of the overall system was again rather disappointing, giving a speedup of about 11 on 26 processors. As with the first system discussed above, there proved to be a considerable imbalance between the various phases which made it difficult to balance the computation on a 32 processor machine.

Coincidentally with skeleton development, a method of performance prediction was developed to guide future prototype development. This involved measurement of coarse parameters such as data sizes and execution times followed by fitting this data using numerical analysis techniques. Although useful predictions could be achieved for a limited range of parameters and for the skeletons executing in the steady state, they were not used in any systematic way during the development of the second system. They guided, however, the design of the performance modeling component of the PMLS parallelising compiler for SML, discussed below.

Overall, the second system highlighted the classic tension between specificity and genericity of components. Here, a family of domain specific skeletons proved more useful than a single general-purpose skeleton.

1.4 Towards Skeleton-based Compilers

1.4.1 Overview

Our experiences with parallel implementation by hand from functional prototypes highlight a number of major bottlenecks in our approach:

- Unsupported translation of a SML prototype to occam2 is hard and time consuming, analogous to hand translating a high-level program to assembly language;
- While hand transformation of SML programs is relatively straightforward, the rearrangement of the equivalent occam2 is also hard and time consuming;
- Skeletons tend to be cumbersome and hard to link into sequential programs by hand;
- In the absence of support tools, it seems easier to build new skeletons for each application rather than to reuse existing ones. Programmers certainly have studied predilections for their own code.

Finally, problem specific skeletons seem to offer maximal performance. However, the derivation of such skeletons is a major research area in its own right.

Nonetheless, our successes in basing sequential imperative implementations on sequential functional prototypes, in particular through relatively simple prototype instrumentation and transformation, suggested exploration of automatic parallelisation. From our idealised approach outlined in Figure 1.1, we sought to identify automatable components for a parallelising compiler based on prototyping, shown in Figure 1.3.

The *front end* should parse and type check the prototype, and construct an internal representation. That representation would probably be based on the abstract syntax tree (AST), and form the common standard for all other stages. The *profiler* should then run the prototype on *test data* sets to gather information about processing and communication costs. The *analyser* should identify HOFs and instantiate the corresponding skeleton *performance models* with profile information to determine whether or not parallelism is exploitable. Next, the *transformer* should rearrange the prototype, perhaps guided by the *analyser* to try and optimise sites of parallelism. Finally, the *back end* should generate the parallel implementation as an imperative program invoking suitably instantiated *skeletons*.

Based on our approach, we constructed three small experimental compilers which enabled exploration of the problems our hand prototyping had identified:

- the PUFF compiler introduced the translation of SML to occam2;
- the SkelML compiler employed automatic transformation guided by prototyping;

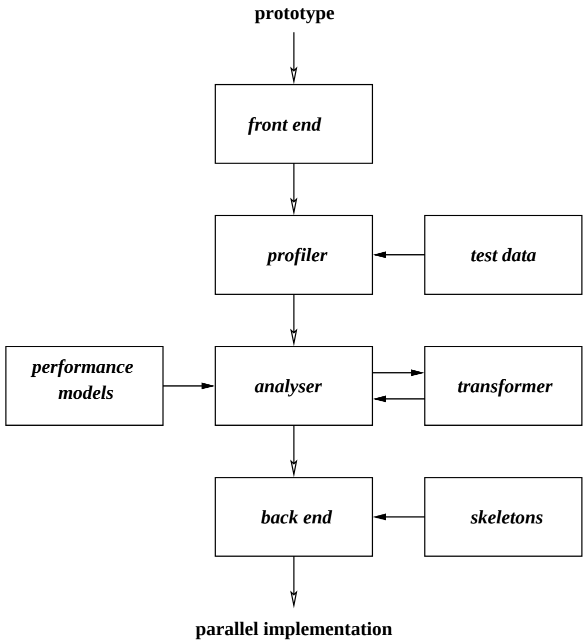

Figure 1.3: Parallelising compiler structure.

- the EKTRAN compiler enabled arbitrary nesting of skeletons.

These are described in the following sections prior to more detailed discussion of our PMLS compiler for a full pure functional subset of SML.

1.4.2 PUFF

Busvine's PUFF (Parallelisation using Farmed Functions) generates occam2 from a pure functional subset of SML [6, 7]. PUFF exploits parallelism in linear recursion of the form:

```
f x = if Call x
      then Base x
      else Comb (f (Upd x),Work x)
```

Here, `Work` is applied to successive values of `x` generated by `Upd` and the results combined by `Comb`. If `Call` determines that processing has terminated, `Base` is applied to `x`. This form is realised as a linear processor farm with control expressions directly analogous to the functions in the body of `f`. Each worker runs a `Work` process, and the farmer uses the other functions to coordinate the overall activity.

PUFF uses a sequential interpreter to profile prototypes. Costs of processing in terms of numbers of sequential operations performed are combined with the actual times of the equivalent operations on a T800-based Meiko, found through measurement. Similarly, data sizes are combined with measured T800 communication times. Models for farm processing and communication costs are instantiated with profiling data.

On examples including naive Fibonacci, Euler totient functions, Winograd matrix multiplication, N Queens, Kelly's ray tracer and Waugh's PIA, PUFF offered better than 80% predictive accuracy.

PUFF was firmly oriented to the T800, lacked transformation and only supported one, unnested skeleton within a restricted functional language. Nonetheless, PUFF embodied many of the features of our idealised prototyping scheme and clearly demonstrated that automatic parallelisation through prototyping was feasible.

1.4.3 SkelML

Bratvold's SkelML [3, 4, 5] built on the PUFF SML to occam2 translator and provided occam skeletons for `map`, `filter`, `fold` and compose. The `map` and `filter` skeletons were both based on linear farms whereas the `fold` skeleton was tree structured.

Where PUFF profiling used an interpreter for simple Par-expressions, SkelML used a rule counting interpreter based on the SML Structural Operational Semantics (SOS). Weights for SOS rules were acquired by timing occam2 generated by PUFF on a T800-based Meiko. Again, skeleton performance models were instantiated with profile data to predict parallel performance.

Prototypes were restructured automatically through six basic transformation rules:

- `map`/compose distribution:
  `map` f `o map` g $\Leftrightarrow$ `map (`f `o` g`)`
- `map/filter` combination:
  `map` f `o filter` p $\Leftrightarrow$ `mapfilter` f p
- `filter/map` combination:
  `filter` p `o map` f $\Leftrightarrow$ `filtermap` p f
- `fold/map` combination:
  `fold` f a `o map` g $\Leftrightarrow$ `foldmap` f a g
- `filter/filter` combination:
  `filter` $p1$ `o filter` $p2$ $\Leftrightarrow$ `filter (fn x =>` $p1$ `x andalso` $p2$ `x)`
- `filter`/compose distribution:
  `filter` $p1$ `o filter` $p2$ $\Rightarrow$ `filter` $p2$ `o filter` $p1$

using simple cost minimisation. Note the use of the sequential `mapfilter`, `filtermap` and `foldmap` which are not defined here.

A process network was extracted from the final form of the prototype, to guide the allocation of processes to processors. Allocation was top-down, with lower-level processes combined on a single processor. As noted, final occam2 generation used the PUFF translator.

SkelML achieved a predictive accuracy of better than 85% on examples including summing a list, finding primes, insertion sorting, Euler totient function, ray tracing, matching polygons and finding the areas of symmetric polygons.

Like PUFF, SkelML supported a restricted functional language and did not exploit parallelism in nested HOFs. Nonetheless, SkelML represented a full realisation of our prototyping scheme.

1.4.4 EKTRAN

Hamdan's EKTRAN is a vehicle for exploring skeleton nesting. EKTRAN is based on a simple functional language, influenced by FP, for coordinating skeletons through HOFs [15, 16, 14]. EKTRAN supports arbitrary nesting of `map`, `fold` and compose. Nested HOFs are exposed through inlining.

The system can also exploit pipeline parallelism from heterogeneous-type function composition. The usual skeletal formulation of nested composition requires that that the range and domain of all functions be of the same type. In EKTRAN, a sequence of mixed type compositions is identified and converted to a list of functions with suitable wrappers to satisfy type checking. The backend then strips off the wrappers and generates appropriate instances of the compose skeleton.

There are three fold skeletons corresponding to the general case and two special cases. The first special case is based on transformation (2) discussed above, where for `fold f b l` the argument function `f` is not associative but has the form:

```
fn h => fn t => f1 (f2 h) t
```

This can be reformulated as:

```
Fold f1 f2 b l = fold f1 b (map f2 l)
```

where `map` is realised in parallel. This new `Fold` is further refined to a version where `f1` can be transformed to introduce associativity, as for example in:

```
fn h => fn t => f h::t → fn h => fn t => [f h]@t
```

EKTRAN is translated to Caml and the Camlot compiler is used to generate C, which is then linked to skeletons written in C with MPI. The system makes considerable use of MPI groups to realise nesting and dynamic reconfiguration of skeletons. Parallel code generated from EKTRAN programs has been run on the EPCC Cray T3D, the Imperial College AP1000 and AP3000, and local workstation networks.

EKTRAN does not identify useful parallelism and has a highly restricted language basis. However,, it is one of the first compilers to realise nested skeleton parallelism automatically, is portable and offers a high degree of cross platform consistency.

1.5 PMLS Compiler

1.5.1 Overview

Our experience with the PUFF and SkelML compilers, combined with that gained in developing the parallel computer vision systems led to the design of a more general parallelising compiler for SML [21, 26, 23], with the following criteria:

- A far greater proportion of the SML language definitions would be covered than in the previous systems.
- The new compiler would be capable of being installed on a broader range of general-purpose parallel computers.
- The skeletons could be executed within other skeletons but not necessarily treated as first class objects within the language.
- The performance prediction would be updated to the new system, incorporating elements from PUFF/SkelML and the performance modeling used in constructing the parallel computer vision systems.
- A new component would be investigated, that of lifting HOFs from arbitrary code using automated proof synthesis.

The central core of the Parallel ML with Skeletons(PMLS) compiler was constructed from 1997 to 2000, incorporating `map` and `fold` skeletons. Considerable effort was put into selecting host compilers for the project. The ML Kit [2] was used for the front end because it provided an evaluator for SML based closely upon the structural operational semantics. Objective Caml [9] was used for the back end because of its relatively lightweight implementation. Since we were targeting general-purpose machines we chose MPI [13] as the communications layer. Although this was known to have high latencies it was available on a wide range of parallel machines.

The compiler itself was implemented as a series of phases. Essentially, there were two major components: the transformational compiler which generated parallel Objective Caml and the backend compiler which called the Objective Caml and C compilers with appropriate options for linking in the MPI and skeleton libraries. The idea was that the transformational compiler could be run on a separate machine so that the "portable" backend component could be kept small with the minimum dependency upon external software. This proved a good design decision since the compiler was subsequently ported to a significant number of Unix-like parallel machines.

Skeleton Design
The efficient use of skeletons in nested mode was known to be a difficult problem to solve [25], so we restricted ourselves to the development of updated versions of `map` implemented as a process farm and `fold` implemented as a divide-and-conquer tree. Communication of arbitrary data was solved using the Objective Caml `Marshal` module for linearising arbitrary types of data, in conjunction with the MPI `MPI_BYTE` datatype. While the Objective Caml foreign function interface is untyped, our compiler generates code from correctly typed source code using semantics-preserving transformations.

The major differences between the previous implementations of `map` and `fold` and the new versions were in the handling of nested constructs. The use of MPI implied an SPMD approach where execution started as a set of identical programs running on a fixed set of processors. Upon encountering the first skeleton, the familiar *master-worker* model was initiated and this remained so throughout skeleton execution. When a skeleton terminated the master processor could optionally broadcast the final results to the workers which only had partial results at that point.

Processor allocation is carried out at startup according to the data extracted from the program by the compiler. Each processor is allocated to a skeleton in a top-down manner and, upon encountering a skeleton, the worker processors resumes execution in the instance function for the skeleton to which they were allocated once the initialisation data for that instance function becomes available. This also applies to inner nested skeletons giving a relatively efficient method of nested initialisation. A complication, however, is that free-values have to be propagated through nested levels. The case for inner nested skeletons requires a slightly modified analysis compared with the outer level where we could make use of the initial phase of joint sequential execution to minimise the amount of free-value transmission. It also creates the need to distinguish between worker processors which only perform work under the auspices of a master processor, and worker processors which are themselves master processors for inner skeletons.

The `pmap` skeleton executes as a processor farm with one master processor and the remaining processors as workers. Since our skeletons are list-based the unit of granularity is a fixed-size block of list elements. The block size is set as a global constant with the value set at run-time but the intention is to use the profiling information to set this automatically. The `pfold` skeleton operates as a binary tree of processors. Here, all non-leaf processors operate as nested masters where a distinction is made between the *root* master for the `pfold` skeleton and a *parent* processor for a node in the processor tree. Parallel nesting is implemented solely from the leaf processors. If a nested instance is encountered from an intermediate node the skeleton is executed sequentially.

One limitation of these skeletons is that the master-worker model tends to cause a bottleneck in communications at the master processor. We generally get good speedups for small numbers of processors but scalability is limited by this bottleneck.

Dynamic profiling
PMLS includes an integrated *dynamic* profiling mechanism. The ML Kit interpreter is used to execute the sequential prototype. This interpreter is a direct implementation of the dynamic semantics of SML so the compiler is able to annotate the syntax tree with information about the number of rules in the semantics which are fired during execution. This leads to sequential performance prediction.

Given a suitable set of "learning" programs it is possible to assign a weight to each rule in the semantics. The test programs cause as many of the different rules to be fired as possible and the actual execution times for the test programs form the dependent variable for weight determination. This system can be expressed in matrix form as:

$$Pw = x \tag{1.1}$$

where P is an $r \times n$ matrix where r is the number of rules and n the number of program executions, w is a vector of weights and x is the vector of execution times. In theory, we can solve this equation for a set of trial programs giving a set of fixed weights and then apply the weights to an unknown profile of rule counts to give a predicted execution time. The main problem is in selecting a suitable set of test programs whereby each rule has a significant representation and no rules dominate the entire data set. This is difficult since some ubiquitous rules, such as accessing an identifier in an environment, occur far more often than other specialised rules such as raising an exception.

Currently, we can use either numerical analysis methods, in particular Singular Value Decomposition fitting, or genetic algorithm techniques [22] to solve the above equation for our test data sets. The numerical methods give more accurate fits but suffer from numerical instability which limits the range of validity of the generated weights. The genetic solution is extremely slow and much less accurate but suffers from less numerical instability. At best this technique only gives a *rule-of-thumb* estimate of the sequential performance of an arbitrary section of code, typically with about a 200% margin. This is sufficient, however, to drive a performance-improving transformation system.

Exemplar Programs
Several significantly-sized exemplar programs have been developed for PMLS. An implementation of the Canny edge detector from the parallel computer vision system gave a speedup of 34 on 129 processors on the Fujitsu AP1000 but gave much faster runtimes and could handle much larger images. A ray tracer for spheres was translated from a Haskell port of one of the tests from the Impala suite [17]. This gave a speedup of 26 on a 31 processor Beowulf class machine. Other examples include; an implementation of matrix multiplication, an implementation of a genetic algorithm support library and several examples using the library, an exact integer linear equation solver, prime filtering and an Euler totient function. All gave modest parallel performance and used `pmap` and `pfold` in a variety of different combinations.

Results	Times/s
$\lambda x.(map\ \lambda y.(map\ \lambda.z * z\ y)\ x)$	52
$\lambda x(foldr\ \lambda y.\lambda z.((map\ \lambda u.u * u\ y) :: z)\ []\ x)$	57
$\lambda x.(map\ \lambda y.(foldr\ \lambda z.\lambda u.z * z :: u)\ []\ y)\ x)$	79
$\lambda x.(foldr\ \lambda y.\lambda z.((foldr\ \lambda u.\lambda v.u * u :: v)\ []\ y) :: z)\ []\ x)$	85
$\lambda x.(map\ \lambda y.(squares\ y)\ x)$	305
$\lambda x.(foldr\ \lambda y.\lambda z.(squares\ y :: z)\ []\ x)$	312
$\lambda x.(squs2d\ x)$	1201

Figure 1.4: HOF lifting in `squs2d`.

In summary, we have created a compiler which can compile the full SML language and provides partial support for the embedding of arbitrary HOFs implemented as parallel skeletons within the code. Our skeletons' implementations were complex and have a number of drawbacks such as the communications overheads and the inability to handle specific types of free functional arguments. However, the skeletons can be arbitrarily nested within one another without programmer intervention. We have tackled successfully, several significant applications and have demonstrated the practicality of the system. We have also characterised system limitations with a view to future improvements such as the development of further skeletons and more complete analyses of some of the compiler phases.

1.5.2 HOF Synthesis

A major criticism of HOF-oriented skeleton programming is that while programmers are not supposed to be aware of underlying parallel concerns, in practice they are restricted to the supported HOFs if parallelism is to be exploited. As part of the PMLS project, Cook investigated the automatic extraction of HOFs from functional prototypes [11, 12].

Core SML is translated to λ-Prolog, and the λ-CLAM proof planner is employed to locate instances of `map`, `fold` and `scan` using higher order unification and middle-out reasoning. Considerable use is made of type information to restrict the search space of potential HOF instantiations.

For example, consider the function to square every element of a list of integer lists:

```
fun squs2d [] = [] |
    squs2d (h::t) = squares h::squs2d t

fun squares [] = [] |
    squares ((h:int)::t) = h*h::squares t
```

The proof plan finds seven solutions shown in Figure 1.4. Note that the plan finally locates the original call to `squs2d` after traversing the entire search

```
fun dist [] [] = []
  | dist (e1::t1) [] = [e1]::dist t1 []
  | dist (e1::t1) (l1::t2) = (e1::l1)::dist t1 t2

fun transpose [] = []
  | transpose (row::rows) = dist row (transpose rows)

fun dotprod [] [] = 0
  | dotprod (e1::t1) (e2::t2) = e1*e2+dotprod t1 t2

fun rowmult row [] = []
  | rowmult row (col::cols) = dotprod row col::rowmult row cols

fun rowsmult [] cols = []
  | rowsmult (row::rows) cols = rowmult row cols::rowsmult rows cols

fun mmult x y = rowsmult x (transpose y)
```

Figure 1.5: Matrix multiplication program.

space.

The intention was that the HOF synthesiser would form part of the PMLS parallelising compiler. To that end, translators were constructed from core SML to λ-Prolog and from λ-Prolog back to core SML. The synthesiser has yet to be fully integrated. It could form a pre-processor, sending the compiler a version of the program containing maximal HOFs so that the transformer and analyser might prune out HOFs where no parallelism could be exploited. Alternatively, the synthesiser could be incorporated into the transformer stage, to generate new versions of a program on demand for analysis.

The synthesiser has been tested on a variety of exemplars, including a HOF-free version of the matrix multiplication program discussed below. Here, the proof plan found 12 different configurations of `map` and `fold`. However, it took 10899 seconds. While further work is needed to reduce synthesis time, we have demonstrated that HOF synthesis is practical as a basis for fully automatic parallelisation of functional programs.

1.6 Case Study: Matrix Multiplication

To illustrate the intended use of our compiler, we present a case study involving the synthesis of a number of alternative HOF-based implementations of a simple matrix multiplication program, shown in Figure 1.5.

The sequential code is written in a fairly natural, if rather naive, style and contains none of our simple HOFs. The HOF synthesis program generates

12 different programs incorporating `foldr` and `map` in different combinations, shown in Figure 1.6. Note that all the programs have the same basic structure of a sequence of two HOFs nested within an outer HOF. The different versions represent all combinations of replacement of each potential HOF site with either a `map`, `foldr` or a function from the original code. This is expected, however, since the synthesizer is set up to lift HOFs from existing code, not to make structural changes to the code.

A cursory glance at these alternatives suggests that many are practically useless. For example, the `transpose` function does not involve any computation at all but it simply transposes the rows and columns of a matrix. Hence, it is unlikely to lead to any speedup at all for coarse-grain MIMD systems when implemented using the `foldr` HOF.

Note that the synthesizer generates the general `foldr` function, rather than a direct instance of our associative `fold` HOF: associative `fold` has a double recursion which greatly increases the synthesis cost. This causes some problems in that not all instances of `foldr` can be directly implemented in parallel. It is possible in some cases to turn these into a `fold` instance by providing specialized versions of our `fold` HOF at particular type instances. This is possible for all the `foldr` instances in these programs.

The first step in deciding which version to implement is to obtain a parallel performance prediction through profiling. At present, our profiling method is not fully automated. We edit each program to indicate HOF instance functions using a semantically neutral marker and run the resulting programs through the profiler. We also *compile and run* the sequential code. The profiler is able to execute the entire program according to the dynamic semantics of SML and count definition rules as they are fired. Rule counts for marked instance functions are gathered and weights derived from a large number of "learning" runs are applied, giving an estimate of the sequential runtime for an *average* execution of each instance function. The average is over *all* instances of that function, even if there are large variations in runtime between different invocations. We currently lack the ability to predict the communications sizes since this involves predicting the behaviour of the Objective Caml `Marshal` module. This should be possible in theory but in practice we currently compile and run the profiled code and use modified skeletons to actually measure the data sizes generated.

Table 1.1 shows the predictions for each version squaring the 3×3 matrix `[[1,2,3],[4,5,6],[7,8,9]]`. T_p is the sequential instance function execution time averages in μS. D_w is the total number of bytes generated as arguments to the instance function. D_r is the total number of bytes generated as results from instance functions. For comparison, T_m is the measured instance function execution times from the sequential runs. These are very similar between the different versions in terms of data sizes and runtimes, which is to be expected, although in general, the same function implemented using `map` has lower overheads than using `fold`.

Given the data sizes and the sequential execution times we are then able to apply our parallel performance models to get an estimate of the runtime for each HOF *individually* for a range of available processor counts. The parallel

```
fun mmult x y = (* Version 1 *)
  map (fn z => map (fn u => dotprod z u)
                   (foldr (fn u => fn v => dist u v) [] y)) x

fun mmult x y = (* Version 2 *)
  foldr (fn z => fn u =>
         (map (fn v => dotprod z v)
              (foldr (fn v => fn w => dist v w) [] y))::u) [] x

fun mmult x y = (* Version 3 *)
  map (fn z => foldr (fn u => fn v => (dotprod z u)::v) []
                     (foldr (fn u => fn v => dist u v) [] y)) x

fun mmult x y = (* Version 4 *)
  foldr (fn z => fn u =>
         (foldr (fn v => fn w => (dotprod z v)::w) []
                (foldr (fn v => fn w => dist v w) [] y))::u) [] x

fun mmult x y = (* Version 5 *)
  map (fn z => rowmult z (foldr (fn u => fn v => dist u v) [] y)) x

fun mmult x y = (* Version 6 *)
  foldr (fn z => fn u =>
         (rowmult z (foldr (fn v => fn w => dist v w) [] y))::u) [] x

fun mmult x y = (* Version 7 *)
  map (fn z => map (fn u => dotprod z u) (transpose y)) x

fun mmult x y = (* Version 8 *)
  foldr (fn z => fn u =>
         (map (fn v => dotprod z v) (transpose y))::u) [] x

fun mmult x y = (* Version 9 *)
  map (fn z => foldr (fn u => fn v =>
                      (dotprod z u)::v) [] (transpose y)) x

fun mmult x y = (* Version 10 *)
  foldr (fn z => fn u => (foldr (fn v => fn w => (dotprod z v)::w) []
                                (transpose y))::u) [] x

fun mmult x y = (* Version 11 *)
  map (fn z => rowmult z (transpose y)) x

fun mmult x y = (* Version 12 *)
  foldr (fn z => fn u => (rowmult z (transpose y))::u) [] x
```

Figure 1.6: Synthesized matrix multiplication programs.

Table 1.1: Profiler output.

V	Position	HOF	T_p	D_w	D_r	T_m	P_2	P_4	P_8
1	outer 1	map	25.1	81	88	18.7	0.0029	0.0061	0.0126
	inner 2	map	8.3	243	196	4.1	0.0036	0.0068	0.0133
	inner 3	fold	9.4	369	351	4.5	0.0141	0.0213	0.0355
2	outer 1	fold	20.2	128	129	25.5	0.0036	0.0068	0.0133
	inner 2	map	8.3	243	196	4.1	0.0097	0.0169	0.0311
	inner 3	fold	9.4	369	351	4.9	0.0141	0.0213	0.0355
3	outer 1	map	26.0	81	88	19.1	0.0029	0.0061	0.0126
	inner 2	fold	9.2	287	240	3.9	0.0146	0.0217	0.0360
	inner 3	fold	9.4	369	351	4.4	0.0141	0.0213	0.0355
4	outer 1	fold	26.5	128	129	19.4	0.0146	0.0217	0.0360
	inner 2	fold	9.2	287	240	3.9	0.0097	0.0169	0.0311
	inner 3	fold	9.4	369	351	4.4	0.0141	0.0213	0.0355
5	outer 1	map	25.0	81	88	18.7	0.0029	0.0061	0.0126
	inner 3	fold	9.4	369	351	4.6	0.0141	0.0213	0.0355
6	outer 1	fold	25.4	128	129	19.2	0.0097	0.0169	0.0311
	inner 3	fold	9.4	369	351	4.4	0.0141	0.0213	0.0355
7	outer 1	map	19.3	81	88	16.0	0.0029	0.0061	0.0126
	inner 2	map	8.3	243	196	3.8	0.0036	0.0068	0.0133
8	outer 1	fold	19.8	128	129	15.9	0.0036	0.0068	0.0133
	inner 2	map	8.3	243	196	4.0	0.0097	0.0169	0.0311
9	outer 1	map	20.2	81	88	15.3	0.0029	0.0061	0.0126
	inner 2	fold	9.2	287	240	4.0	0.0146	0.0217	0.0360
10	outer 1	fold	20.7	128	129	15.5	0.0146	0.0217	0.0360
	inner 2	fold	9.2	287	240	4.0	0.0097	0.0169	0.0311
11	outer 1	map	19.2	81	88	16.9	0.0029	0.0061	0.0126
12	outer 1	fold	19.7	128	129	14.7	0.0097	0.0169	0.0311

performance models are also empirical and based upon a large number of executions of skeletons over as wide a range of parameters as is practical given the number of parameters and the actual time per execution. This method was developed in [28] and gives good estimates for skeletons in the *steady state* over a *narrow* range of parameters. The parameterisation of the models in this case is set to the values produced by the profiler.

Table 1.1 shows the predictions of the runtime in seconds for each HOF in the synthesized programs at 2, 4 and 8 processors. These suggest that the most likely candidate is Version 11, a single non-nested `map`.

Note that single processor values are fictitious, based on an extrapolation of the parallel execution times to one processor. Thus we cannot get a speedup estimate without a whole program execution estimate from the profiler. This requires the use of fitting weights from different ranges of parameters which

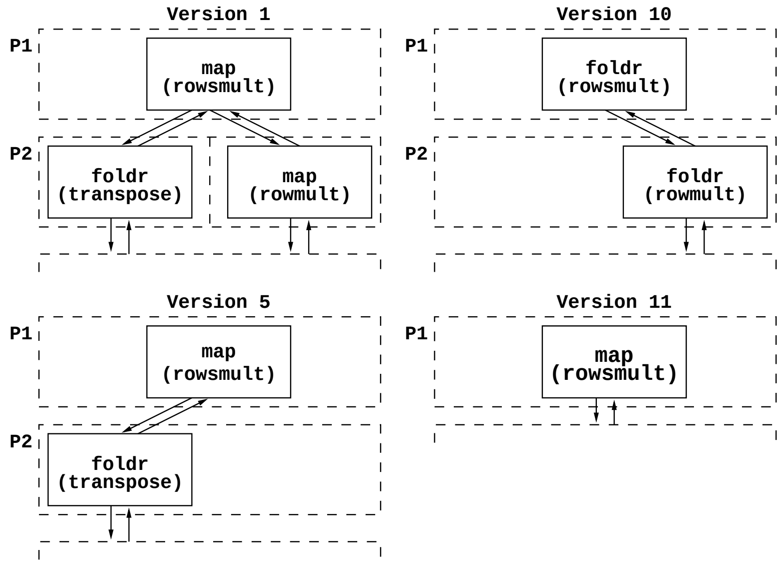

Figure 1.7: Parallel schema.

makes the comparison unreliable.

Note that there is no estimate of the nesting overhead between different skeletons in the same program. The development of a uniform model for each possible combination of skeletons is impractical due to the explosion in the number of parameters. Furthermore, the fitting method is not compositional due to the numerical instability of the models outside the narrow range of validity of the parameters. The best possible with the current state of the system is to use the runtimes for each HOF and to assume a large nesting overhead.

Note that there are apparent inconsistencies in the predictions inasmuch as the same skeleton can give different predictions between different versions, for example "inner 2 map" between Versions 1 and 2. This is a consequence of the implementations of the outer skeletons (`map` or `fold`) which provide the inner skeletons with the data in a different order and in different decompositions. Small differences such as this can give apparently significant differences in the predictions due to the numerical instability in the fits discussed above.

Finally, we have executed each program to get the actual parallel runtime for each version assuming all HOFs execute in parallel according to the number of available processors. The parallel execution schema for some of the versions is depicted in Figure 1.7. The parallel runtimes on 1, 2, 4 and 8 processors are shown in Table 1.2, for a 50×50 matrix. These are typical of the behaviour of our skeletons when instantiated with functions with little or no work relative to communication. The outer `map` skeleton shows lower overheads than a `fold` in an equivalent position as seen by comparing Versions 7 with 8 or 11 with 12. Here nesting is of little use since the overheads are high with respect to the

Table 1.2: Parallel runtimes for nested implementations.

Version	HOFs	R_1	R_2	R_4	R_8
1	MMF	0.483	0.595	15.69	7.954
2	FMF	0.472	0.886	11.51	5.759
3	MFF	0.481	0.600	8.655	4.602
4	FFF	0.473	0.882	8.844	5.108
5	MF	0.455	0.578	1.724	0.535
6	FF	0.449	0.875	2.336	0.651
7	MM	0.167	0.267	1.684	0.738
8	FM	0.185	0.840	4.449	2.035
9	MF	0.164	0.270	1.389	0.699
10	FF	0.167	0.598	1.952	0.799
11	M	0.147	0.310	0.127	0.090
12	F	0.137	0.590	0.167	0.112

sequential runtimes.

For this application the single non-nested `map` gives the best actual performance. This has been borne out by other work on matrix multiplication [27]. Furthermore, these runtimes broadly correspond to the predictions from the profiler.

1.7 Conclusions

Our investigation of computer vision problems and of the construction of automatic parallelising compilers, both based on HOF realisations as algorithmic skeletons, give useful insights into the feasibility and scalability of the broad HOF/skeletons approach. In particular, these various studies have been novel in:

- Investigating the construction of realistic parallel algorithms in a substantive problem domain;
- Seeking to match the parallel topology to the algorithm rather than the algorithm to the topology. Indeed, this approach was the main practical motivation for utilising skeletons;
- Basing sequential profiling on Structural Operation Semantics rule counts rather than absolute or simulated times;
- Closely coupling instrumentation, analysis and transformation in parallelising compilers;
- Seeking to broaden parallel opportunities in prototypes by synthesising higher order function instances;

- Providing a large collection of functional exemplars with equivalent parallel implementation, which forms a basis for evaluating the behaviours of our various parallelising compilers.

However, there is much scope for further work. This includes:

- Exploring stable approaches to predicting performance from profiling;
- Linking profiling and transformation so that transformation is guided by profiling;
- Linking profiling and synthesis so that synthesis is guided by profiling;
- Exploring skeleton based parallelisation in the presence of conditional expressions.

Acknowledgements

This research has been supported by UK SERC/EPSRC grants:

- SERC GR/F01833 (1989/92) "Rapid prototyping and parallel implementation of algorithms for high-level image interpretation"
- SERC GR/G30522 (1990/91) "Image interpretation, graphics, rapid prototyping and performance evaluation"
- SERC GR/J07884 (1993/96) "A reconfigurable parallel visual architecture through prototype instrumentation and transformation"
- EPSRC GR/L42889 (1997/00) "Parallelising compilation of Standard ML through prototype instrumentation and transformation"

We would like to thank everyone who contributed to this work, in particular: Kevin Waugh, David Busvine, Tore Bratvold, George Koutsakis, Michael Clarkson, Paul Bristow, Mohammad Hamdan, Andrew Cook, Andy Wallace (SERC GR/F01833 and GR/G30522, EPSRC GR/J07884), Peter King (SERC GR/30522, EPSRC GR/L42889), Andrew Ireland (EPSRC GR/L42889).

We would like to thank the Edinburgh Parallel Computing Centre, the Imperial College Fujitsu Parallel Computing Research Centre, Quadstone Ltd and Harlequin Ltd for supporting this work.

Bibliography

[1] P. Besl and R. Jain. Invariant surface characteristics for 3d object recognition. *Computer Vision, Graphics and Image Processing*, 33:33–80, 1986.

[2] L. Birkedal, N. Rothwell, M. Tofte, and D. N. Turner. The ml kit Version 1. Technical Report 93/14, Department of Computer Science, University of Copenhagen, 1993.

[3] T. Bratvold. Determing useful parallelism in higher order functions. In *Proceedings of 4th International Workshop on the Parallel Impelentation of Functional Languages, Aachen, Germany*, pages 213–226. Aachener Informatik-Berichte, September 1992.

[4] T. Bratvold. A skeleton based parallelising compiler for standard ML. In *Proceedings of 5th International Workshop on Impelentation of Functional Languages*, pages 23–34, Nijmegen, Holland, September 1993.

[5] T. Bratvold. Parallelising a functional program using a list-homomorphism skeleton. In *Proceedings of PASCO'94*, pages 44–53. World Scientific Publishing Company, 1994.

[6] D. Busvine. Implementing recursive functions as processor farms. *Parallel Computing*, 19:1141–1153, 1993.

[7] D. Busvine. Detecting parallel structures in functional programs. In *PhD thesis*. Dept of Computing & Electrical Engineering, July 1994.

[8] J. Canny. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-8:679–698, 1986.

[9] E. Chailloux, P. Manoury, and B. Pagano. *Développement d'applications avec Objective Caml*. O'Reilly, Paris, April 2000.

[10] M. I. Clarkson. Parallel implementation of a range image segmentation algorithm. Master's thesis, Department of Computing and Electrical Engineering, Heriot-Watt University, Riccarton, Edinburgh, EH14 4AS, 1994.

[11] A. Cook. *Using Proof in Transformation Synthesis for Automatic Parallelisation*. PhD thesis, Dept of Computing and Electrical Engineering, Heriot-Watt University, 2001.

[12] A. Cook, A. Ireland, and G. Michaelson. Higher order function synthesis through proof planning. In *Proceedings of 16th Annual International Conference on Automated Software Engineering ASE 2001*, pages 307–310, San Diego, USA, November 2001. IEEE Computer Society.

[13] M. P. I. Forum. Mpi: A message-passing interface standard. *International Journal of Supercomputer Applications and High Performance Computing*, 8(3/4), 1994.

[14] M. Hamdan. *A Combinational Framework for Parallel Programming Using Algorithmic Skeletons*. PhD thesis, Department of Computing and Electrical Engineering, Heriot-Watt University, 2000.

[15] M. Hamdan, G. Michaelson, and P. King. A scheme for nesting algorithmic skeletons. In C. Clack, T. Davie, and K. Hammond, editors, *Proceedings of 10th International Workshop on Implementation of Functional Languages*, pages 195–212, University College London, September 1998.

[16] M. Hamdan, G. Michaelson, and P. King. A framework for nesting algorithmic skeletons. In *Proceedings of 1999 International Conference on Parallel Computing ParCo'99*, pages 542–549, Delft, Holland, January 2000. Imperial College Press.

[17] Impala. Impala – IMplicitly PArallel LAnguage Application Suite. <URL:`http://www.csg.lcs.mit.edu/impala/`>, July 2001.

[18] G. Koutsakis. Parallel low level vision from functional prototypes. Master's thesis, Dept. of Computing and Electrical Engineering, Heriot-Watt University, Riccarton, Edinburgh, October 1993.

[19] V. F. Leavers. Survey: Which hough transform? *CVGIP: Image Understanding*, 58(2):250–264, September 1993.

[20] P. McAndrew and A. M. Wallace. Rapid invocation and matching of 2d images to 3d models using curvilinear data. In *Proc. of 3rd Int. Conf. on Image Processing and its Applications*, pages 83–87, Warwick, July 1989.

[21] G. Michaelson, A. Ireland, and P. King. Towards a skeleton based parallelising compiler for sml. In C. Clack, T. Davie, and K. Hammond, editors, *Proceedings of 9th International Workshop on Implementation of Functional Languages*, pages 539–546, University of St Andrews, September 1997.

[22] G. Michaelson and N.Scaife. Parallel functional island model genetic algorithms through nested skeletons. In M. Mohnen and P. Koopman, editors, *Proceedings of 12th International Workshop on the Implementation of Functional Languages*, pages 307–313, Aachen, September 2000.

[23] G. Michaelson, N. Scaife, P. Bristow, and P. King. Nested algorithmic skeletons from higher order functions. *Parallel Algorithms and Applications special issue on High Level Models and Languages for Parallel Processing*, 16(2-3):181–206, 2001.

[24] G. J. Michaelson and N. R. Scaife. Prototyping a parallel vision system in standard ml. *Journal of Functional Programming*, 5(3):345–382, July 1995.

[25] R. Rangaswami. *A Cost Analysis for a Higher-Order Parallel Programming Model*. PhD thesis, University of Edinburgh, 1995.

[26] N. Scaife, P. Bristow, G. Michaelson, and P. King. Engineering a parallel compiler for sml. In C. Clack, T. Davie, and K. Hammond, editors, *Proceedings of 10th International Workshop on Implementation of Functional Languages*, pages 213–226, University College London, September 1998.

[27] N. Scaife, G. Michaelson, and S. Horiguchi. Comparative cross-platform performance results from a parallelizing sml compiler. In *Proceedings of 13th International Workshop on the Implementation of Functional Languages*, pages 191–206, Stockholm, September 2001.

[28] N. R. Scaife. *A Dual Source, Parallel Architecture for Computer Vision.* PhD thesis, Department of Computing and Electrical Engineering, Heriot-Watt University, 1996.

[29] N. R. Scaife, G. J. Michaelson, and A. M. Wallace. Four skeletons for computer vision. In *Implementation of Functional Languages '97*, September 1997.

[30] A. M. Wallace, G. J. Michaelson, P. McAndrew, K. Waugh, and W. Austin. Dynamic control and prototyping of parallel algorithms for intermediate and high-level vision. *IEEE Computer*, 25(2):43–53, February 1992.

[31] A. M. Wallace, G. J. Michaelson, N. Scaife, and W. J. Austin. A dual source, parallel architecture for computer vision. *The Journal of Supercomputing*, 12(1/2):37–56, January/February 1998.

[32] K. G. Waugh. Parallel imperative programs from functional prototypes. In *Proceedings of the Workshop on the Parallel Implementation of Functional Languages*, pages 75–88. Dept. Electronics and Computer Science, University of Southampton, June 1991.

[33] K. G. Waugh and G. J. Michaelson. Parallel process placement from functional prototype instrumentation. In *Proceedings TENCON 91, Energy, Computer, Communication and Control Systems*, volume 2, pages 275–282, New Delhi, August 1991. IEEE.