J. Functional Programming **1** (1): 1–000, January 1993

Prototyping a parallel vision system in Standard ML

Greg Michaelson
Department of Computing and Electrical Engineering,
Heriot-Watt University,Edinburgh, EH14 4AS.
Phone: +44 31 452 3423
Email: greg@cee.hw.ac.uk

Norman Scaife
Department of Computing and Electrical Engineering,
Heriot-Watt University,Edinburgh, EH14 4AS.
Phone: +44 31 452 4177
Email: norman@cee.hw.ac.uk

Abstract

The construction of a parallel vision system from Standard ML prototypes is presented. The system recognises 3D objects from 2D scenes through edge detection, grouping of edges into straight lines and line junction based model matching. Functional prototyping for parallelism is illustrated through the development of the straight line detection component. The assemblage of the whole system from prototyped components is then considered and its performance is discussed.

1 Introduction

1.1 Overview

We have been investigating the use of functional prototyping in the development of parallel systems. In this paper we discuss the construction of a simple but complete parallel vision system which will recognise 3D objects from 2D intensity images. The system was prototyped in a pure functional subset of Standard ML and implemented in Occam2 on a Meiko Computing Surface.

An important objective of this work is to give us experience in building a large scale project from discrete modules. In particular, we want to investigate how parallel performance is affected when existing prototyped system components are linked together to give a complete system. Subsidiary issues are the problems associated with connecting together such modules and the programmer effort involved in the connection process. To investigate this we take our existing components, combine them at the prototype and implementation level and then compare the optimal performance predicted by the prototype with a hand-optimised implementation.

Thus, our main focus is on the development of individual parallel components from functional prototypes, the subsequent integration of these components and the resulting parallel performance. We also consider various issues in the translation from SML to Occam2, in particular the use of skeletons in the identification of useful parallelism.

We choose to prototype systems rather than implementing them directly for a number of reasons. First of all, during system development, as opposed to design, it is common to identify changes and improvements which have major implications for the design and would best be met by rebuilding the system. We view prototyping as a bridge between design and implementation where the design drives the prototype but the prototype can have implications for the design. Thus, major design changes are made at the prototyping stage: thereafter the implementation is based on a frozen design and is subsequently straightforward. Secondly, prototyping enables a focus on functionality as opposed to efficiency. A prototype demonstrates practically that particular algorithmic approaches will work. Once the algorithms are chosen the implementation work can concentrate on efficiency issues. Finally, a prototype forms a practical standard for evaluating the implementation. That is, while the performance of the prototype and implementation may differ markedly, the I/O behaviour on common test data should be the same.

Note that our choice of SML and Occam2 are effectively pre-given as we already have substantial experience in using both for computer vision. SML was developed at the University of Edinburgh in the late 1970's and so we benefited from proximity to an active local community and excellent support for mature, robust implementations. SML has been taught at Heriot-Watt University since the early 1980's and was a natural choice when we began to look at functional prototyping. Similarly, when we started investigating parallel computer vision, we had access to the transputer based Meiko Computing Surface at the Edinburgh Parallel Computer Centre. At that time, programming transputers using Occam2 was significantly faster than in other languages, for example Parallel C. Hence our initial decision to use Occam2 as the target for parallel implementations.

The rest of the paper is organised as follows. The next two sub-sections give brief overviews of functional prototyping for parallelism and computer vision. Section 2 considers in detail the prototyping of a Hough transform for straight line detection from edge detected images. Section 3 discusses the translation from SML to Occam2, illustrated with fragments of the Hough transform prototype and parallel implementation. Section 4 describes the integration of the Hough transform with existing functionally prototyped components for edge detection and vertex based model matching. Section 5 presents the performance of the combined parallel system. Finally, in Section 6 conclusions and future avenues for research are considered.

1.2 Functional Prototyping for Parallelism

Parallelism is a Holy Grail for Computing, bearing a seductive promise of vast increases in performance and functionality. Alas, despite intensive research, general methodologies for parallel system development remain elusive. Much parallel system construction is ad hoc and performance is often disappointing.

Functional programming appears attractive as a basis for parallelism. Referential transparency means that functional components cannot interact through changes to shared objects. Thus, programs are evaluation order independent and contain a richness of potential parallelism.

There have been three main approaches to exploiting functional parallelism. Implicit parallelism seeks to map functional programs directly onto multi-processor hardware. For example the ALICE (Cripps *et al.*, 1987) and GRIP (Peyton Jones *et al.*, 1987) systems are based on special hardware for graph reduction of functional programs after compilation to combinatory forms. While this approach has been immensely influential, both for functional language implementation and for parallel hardware design, the parallelism tends to be too fine grain for direct efficient exploitation. For explicit parallelism, functional languages are augmented with special constructs or libraries for parallelism, for example Caliban (Kelly, 1987) and paraML (Bailey and Newey, 1994). This has the disadvantage of placing the onus on the programmer to identify where parallelism is appropriate, requiring detailed knowledge of the underlying architecture and the performance of programs upon it, analogous to machine code programming. Skeleton parallelism (Cole, 1989) is based on efficient generic parallel harnesses for common functional constructs, typically higher order functions. Programmers may either choose skeletons based on expert understanding of the program and architecture, or skeletons may be identified automatically from programs (Darlington *et al.*, 1993). The harnesses are then instantiated with code equivalent to the function parameters. Skeletons may be thought of as lying between implicit and explicit parallel programming. The use of higher order functions explicitly identifies sites of potential parallelism but does not necessarily imply that such parallelism will be exploited.

We have been using skeleton parallelism for the parallel implementation of vision algorithms from functional prototypes (Wallace *et al.*, 1992). A prototype is constructed from common or special purpose higher order functions for which there are parallel harnesses. The program is then instrumented on characteristic and pathological data sets to find data flows and processing costs. These costs enable the identification of sites of useful parallelism. Programs may be transformed to try and optimise parallelism. For sites of useful parallelism, the corresponding harnesses are instantiated. Otherwise sequential code is produced.

1.3 Computer Vision

Computer vision has been a major research area for over 40 years. While general visual abilities comparable to even the humblest animal remain a distant prospect, there have been striking successes in restricted domains, for example manufacturing inspection, and the application of computer based vision systems is growing.

Computer vision is generally divided into low, intermediate and high level tasks. Low level vision is concerned with converting one iconic image into another, usually convolving or transforming the source to the target image. For example, edge detection is based on convolution with filters to enhance discontinuities. Intermediate level vision involves grouping low level details with common properties. For example, feature space transformation enables the detection of features such as vertices, lines and surfaces from convolved images. High level vision enables object recognition, typically by matching known models to parts of scenes, where the models and scenes are described in terms of features. Fig. 1 shows a 3D object recognition system based on intensity images where a low level Canny edge detector (Canny, 1986) feeds an intermediate level Hough transform (Leavers, 1993) which in turn feeds a high level perspective inversion algorithm (McAndrew and Wallace, 1989) for model matching.

Vision algorithms are data intensive and hence good candidates for benefiting from parallelism. Most work has been on SIMD implementations of low level tasks (Kittler and Duff, 1985) where ease of problem decomposition enables good performance from simple algorithms. Intermediate vision algorithms have been implemented on both SIMD (Rosenfeld *et al.*, 1988) and MIMD (Austin *et al.*, 1991) architectures. As with low level activities, pixel based data eases partitioning. Parallel high-level vision is relatively recent (Bhanu and Nuttall, 1989; Amini *et al.*, 1989). Here, the main requirement is to constrain the search space which grows with both scene detail and the number of models.

We are starting to build a multi-source (eg intensity, depth, surface data) parallel vision system which will dynamically reallocate resources to the most promising source. We already have parallel implementations of a low level Canny edge detector (Koutsakis, 1993) and a high level model matcher (Waugh *et al.*, 1990), both developed through functional prototyping. We decided to use these components to construct an experimental complete parallel vision system to recognise 3D objects in 2D scenes from intensity data.

The missing component for a complete vision system was the straight line detector. The Hough transform was selected as a well-studied and reasonably reliable method and was subjected to our familiar method of prototyping in SML and implementing in Occam2. It turns out, however, that the Hough transform represents a tricky problem for parallel systems and so has been discussed at length in the next section.

2 Prototyping the Hough Transform

2.1 The Hough Transform

The Hough transform is a general technique for solving sets of underdetermined equations where the number of solution classes is unknown. The solutions are arrived at by forming clusters of potential solutions in the solution space and attributing dense clusters of possible solutions to the most likely solutions. The technique was originally developed for recognising particle tracks in bubble chambers (Hough, 1962) but has widespread uses in vision related tasks.

An example of one of these tasks is locating 2D lines in an intensity image (Dudani

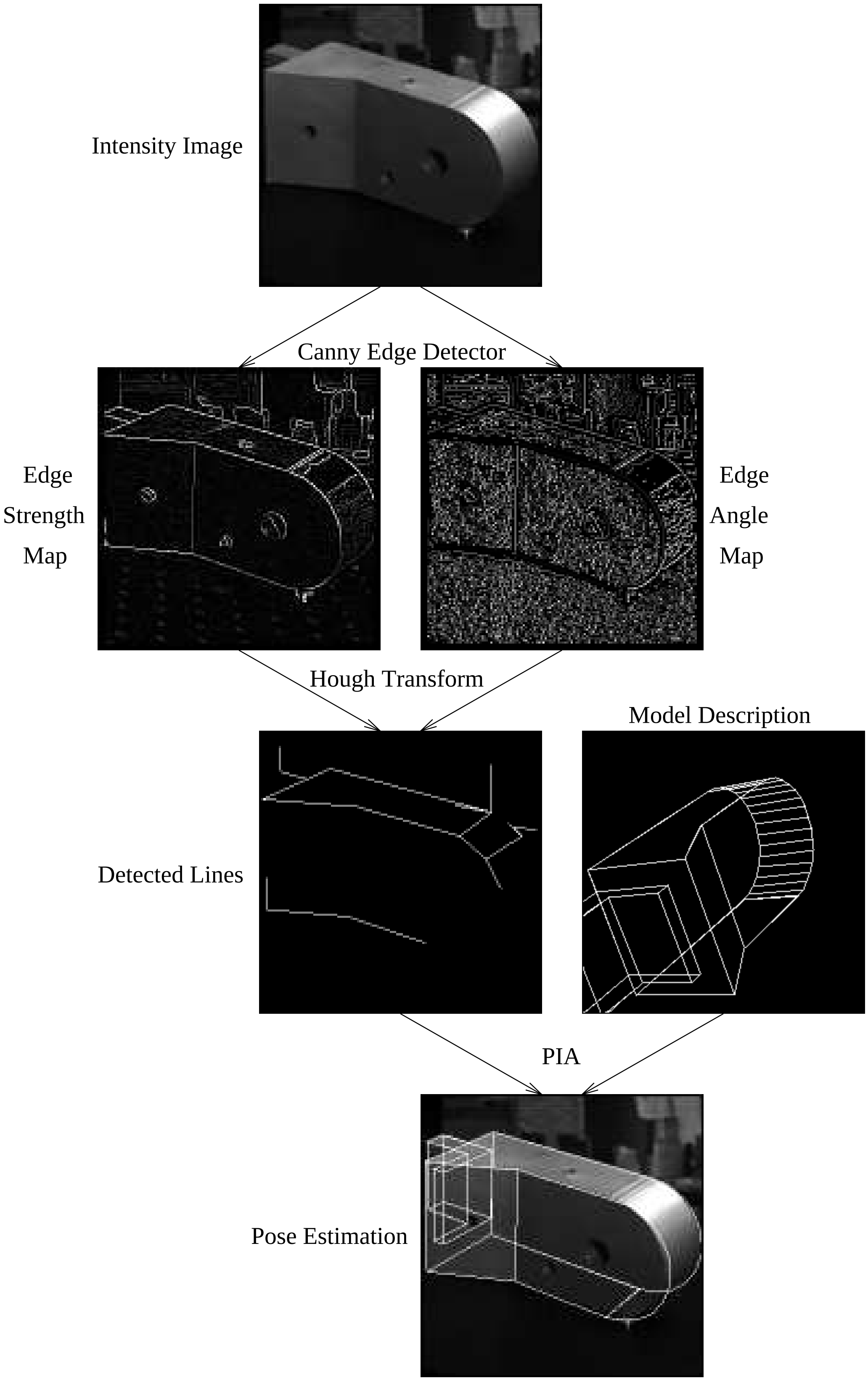


Fig. 1. Object Recognition

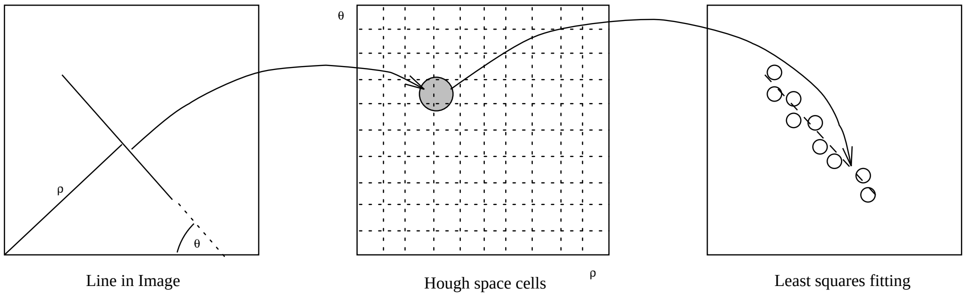


Fig. 2. Schematic of the Hough Transform Process

and Luk, 1978). In this case the system of equations is the normal form for the equation of a straight line:

$$\rho \;=\; x\;\cos\theta \;+\; y\;\sin\theta \tag{1}$$

The potential solutions are plotted in solution space (Hough space), which in this case consists of (ρ, θ) space. For each point in the image (x, y), a sinusoid is plotted in Hough space by scanning θ over a suitable range of values and calculating ρ using Eqn. 1. Points through which large numbers of these sinusoids pass represent the likely (ρ, θ) value of straight lines in the image. Practically, this can be achieved by quantising (ρ, θ) and using a voting system to determine the most likely values of ρ and θ, see Fig. 2.

Formulated in this way, the Hough transform technique is a reasonably robust method in that it has good resistance to noise and can work with high levels of occlusion (Illingworth and Kittler, 1988). This is because the method uses global information concentrated at single possible solutions. It is also a very general method, being applicable to any curves for which parametric forms are available and can be generalised (at the expense of computational complexity) to any representable object (Ballard, 1981).

Note, however, that there are a large number of variations on this basic theme (Leavers, 1993). The major drawbacks of the Hough transform are the size of the transformation space and the computational complexity of the accumulation process and there have been a number of methods developed to reduce one or both of these resource problems.

One such trick for a straight line Hough transform is that if angular information is available from an edge detector then it can be used directly in Eqn. 1 to give a one to one mapping which reduces the complexity of the accumulation strategy.

There have been numerous attempts to implement the Hough transform in parallel, for instance (Austin *et al.*, 1991). The Hough transform represents a tricky problem for parallel systems since the accumulation stage implies some kind of global communications.

Note that by *global communication* we refer to any inter-processor communication during the parallel portion of the algorithm (as opposed to the initialisation phase). This means both point-to-point communications such as between individual processing elements as well as global updates of data from one processor to all processors.

Broadly speaking, parallel Hough transforms can be classified into the three categories of input partitioning, output partitioning and mixed input/output partitioning (Leavers, 1993).

Input partitioning requires the starting image to be divided up between the processing elements. Each element then transforms its portion and transmits the resulting histogram to a central element for combining into a single Hough space. This process is highly inefficient and is only feasible when the number of points accumulated for each image point is small.

Output partitioning requires Hough space to be divided up between the processing elements. Each element transforms the entire image, accumulating only those points that lie within its designated portion of Hough space. This has greater potential for exploiting parallelism, particularly if the postprocessing in Hough space is extensive, for instance modelling noise by Gaussian filtering or performing maximal likelihood estimation in the form of linear regression.

There is still an element of global computation, however, since peaks in Hough space can be split between neighbouring Hough cells which may be on distinct processing elements. This requires global processing to merge similar features although this need not be significant for sparse images. It would be possible to overcome this locally by providing direct communications between processing elements, for instance in a two dimensional array. We use a simple parallel processing model in which there is no direct communication between slave processors: introducing such refinements would complicate our analysis. The algorithmic skeletons approach, however, is not restricted in this way and could decide between such implementations based on suitable performance models.

Mixed input/output partitioning requires a relatively sophisticated communications model but has been implemented in practice on shared memory architectures (Leavers, 1993).

2.2 Prototyping the Hough Transform in SML

Although previous work has reported acceptable results with the input partitioned algorithm (Lotufo *et al.*, 1989), in our case the output partition was chosen on the basis of the degree of post-processing required in the implementation. Mixed partitioning was rejected to allow the implementation to use the simple algorithmic skeleton implementations that have been well-studied in the past. Fig. 3 is a block diagram of the implementation.

An important point is that, under the algorithmic skeletons methodology, the top-level problem decomposition is set by the choice of higher-order functions (skeletons) made by the programmer. This suggests to the system (the implementer in this case, or more ideally, a parallelising compiler) how the problem is structured, but not how to go about mapping this structure onto the chosen hardware. These

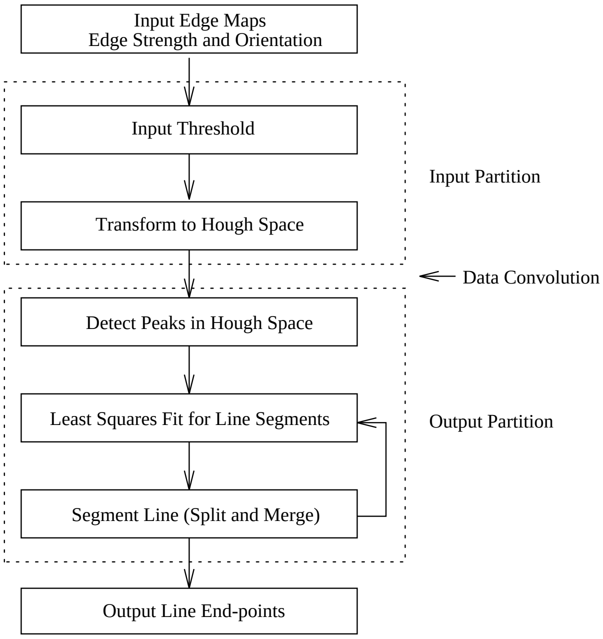


Fig. 3. Block Diagram of the Hough Implementation

architectural decisions, such as the geometric arrangement of processor farms† and processor pipelines, can be made with the aid of performance data from the prototype, which can be used to make direct predictions about the performance of a given arrangement.

There are three top-level candidates for parallelism in the Hough transform described above; the transformation from the image to Hough space, the accumulation of Hough space points into a histogram and the post-processing of Hough space.

The post-processing operation can additionally be subdivided into peak detection, least squares fitting (where the pixels accumulated in a Hough cell are fitted to a straight line) and line segmentation (the process of dividing up pixels on an infinite straight line into contiguous, finite subsegments). These, however, were considered as a single entity since there was not, in the images studied, a sufficient number of peaks in Hough space or contiguous subsegments on detected lines to consider implementing these operations in parallel.

In modelling the parallel aspects of the system it has been found from previous work (Wallace *et al.*, 1992; Bratvold, 1993) that a small number of high-level programming constructs suffice to model the coarse grain MIMD parallelism available to the project. Summarising this work, the constructs considered were *function composition* which models processing elements in a pipeline, the *higher-order function*

† We use the terminology *processor farm* first introduced by May and Shepherd at Inmos in 1987 (May and Shepherd, 1987). Under this scheme there is a master processor (*valve or farmer processor*) meting out work allocations to worker processors on demand. It is also sometimes called the *demand-driven* model.

`map` which can be implemented efficiently in a number of ways on parallel systems, most notably processor farms and simple geometric decomposition, *partial application* which corresponds to preloading of data onto processing elements during an initialisation phase and the *higher-order function* `flatten` which can be viewed in terms of worker processors communicating their results to a single processor.

Note, however, that although these represent equivalent constructs between the functional language and the parallel implementation, these issues would not normally be taken into account by a programmer developing a functional language prototype under algorithmic skeletons method of parallel programming.

The basis of our prototype is the application of `map` to a list of Hough space portion definitions with the input image and program parameters built into the function that `map` applies. While it is disappointing that `map` is the only algorithmic skeleton in the top level of this application, it is encouraging that a complex system such as a Hough transform can be built using only a small subset of the skeleton parallelism available. There has been some criticism of the algorithmic skeletons idea in that it lacks the generality of other methods (Hammond, 1994). This application, however, illustrates that complete generality is not necessary in the skeleton component of an application. The host languages provide the generality and the skeleton parallelism controls the parallel component of the system.

Provided a sufficient number of skeletons and their equivalent implementations are available, any parallel architecture can be treated in this way. What constitutes a minimal set of skeletons, however, is open to debate. It may be that only a small number are required since some skeletons can be expressed in terms of others (Bratvold, 1994).

`map` is the only skeleton required because at all stages of the computation the data is represented by lists of homogeneous data. The edge map and Hough space are both represented using nested lists and the histogram is built using indexed lists. Post-processing is carried out on lists of (x,y) co-ordinates representing detected lines. Below the top level, however, there are other skeletons in use, for instance `fold` and `filter` play a significant role in the post-processing function.

2.2.1 The SML Prototype

Fig. 4 shows the top level code for one possible SML implementation.

The image is supplied in HIPS (Cohen *et al.*, 1982) format in the guise of a HIPS abstype. The dimensions of the image are extracted (`rows` and `cols`) along with the edge image `eimg` and the angular information `aimg`. The definitions of the rectangular areas in Hough space to be processed by each worker are generated by `split_hough`, although strictly speaking this is not necessary in predicting the performance of the parallel implementation and has been done this way merely to indicate one possible method of parallelisation. The parameters `hpx`, `hpy` and `hpo` define the number of divisions in the x and y dimensions and the percentage of overlap of the portions.

The functions modelling the discrete processes (transformation, accumulation and post-processing) are then generated by partial application (`tfn`, `acfn` and `ppfn`,

```
fun hough (hips:HIPS)
          (low:int) (rhoinc:int) (angleinc:int)
          (hpx:int) (hpy:int) (hpo:int)
          (peakth:int) (clustdist:real)
          (mergedist:real) (mergeangle:real) (minlinelen:real) =

    let

        (* Get data from image abstype *)
        val rows = get_rows hips
        val cols = get_cols hips
        val eimg = nth(get_frames hips,0)
        val aimg = nth(get_frames hips,1)

        (* Split hough space into portions *)
        val hough_portions = flatten (split_hough cols rows hpx hpy hpo)

        (* Transform image into parametric form *)
        val tfn = transform2 low rhoinc angleinc rows eimg aimg

        (* Generate histogram in hough space and threshold peaks *)
        val acfn = accumulate2 peakth

        (* Perform post-processing on data *)
        val ppfn = map (post_process2 clustdist)

        (* Run farm/pipeline emulation *)
        val lines = map (ppfn o acfn o tfn) hough_portions

        (* Merge resulting lines *)
        val fflines = flatten (flatten lines)
        val mrglines = map t1f4 (merge_lines mergedist mergeangle fflines)

    in

        (* Filter out short lines *)
        filter (fn l => rline_length l > minlinelen) mrglines

    end
```

Fig. 4. Top level SML Hough Implementation

```
(* Transform partial images into list of partial Hough spaces *)
val hough_space_list = map transform image_portions

(* Accumulate Hough space from partial Hough spaces *)
val hough_space = fold accumulate null_space hough_space_list

(* Perform post-processing on Hough space *)
val lines = map post_process hough_space
```

Fig. 5. Hypothetical Prototype for Input Partitioning

respectively), corresponding to pre-distribution of the program parameters, including the initial image. The three components (`tfn`, `acfn` and `ppfn`) are composed and mapped over the Hough space portion definitions (`hough_portions`), corresponding to farming pipelines of three processors.

Finally, the global merge process is carried out by flattening the result list, if processor farming is used as the parallel implementation then flattening a list that has been distributed among separate processing elements is conceptually similar to the worker processors returning their results to a central point (the farmer processor). Some essential global post-processing is then carried out by the `merge_lines` function.

2.2.2 Input vs Output Partitioning

Although we have stated our preference for the output partitioned version, in fact both input and output partitioned versions were written initially and the two were used as the basis for the final version.

The problem with input partitioning (Fig. 5) is that partial Hough spaces have to be redistributed in the middle of the computation entailing an extra global communication of iconic data compared to the output partition method. On some parallel system architectures, for instance shared memory systems, this might actually be more efficient but for linear pipelines of transputer systems it is likely to be expensive.

It became obvious comparing the two that there was substantially more work required on Hough space (peak detection, least squares fitting and line segmentation) than in converting the edge map into Hough space (thresholding and applying Eqn. 1) and so output partitioning is more likely to benefit from parallelism. An interesting fact is that it only takes about a couple of hours to recode from one to the other in SML, although it should be said that the conversion was more complex than the simple program transformations used elsewhere in this paper and would be difficult to automate.

Theoretically, it is possible to deduce both partitioning methods (and even mixed partitioning) from a purely declarative definition of the problem. The number of transformations required and the high complexity of costing each possible alternative (either running each version on test data or predicting performance from a performance model) means that arriving at two radically different solutions will

```
fun foldr ff def nil    = def
  | foldr ff def (h::t) = ff h (foldr ff def t)

fun foldrll ff def nil    = def
  | foldrll ff def (h::t) = foldr ff (foldrll ff def t) h
```

Fig. 6. The `foldr` Function Generalised to Nested Lists

require prohibitive calculation times. Good heuristics in the search process may, however, make such a proposition viable so automatic program transformation is still a very promising line of enquiry.

2.2.3 Handling Iconic Data

One of the problems in prototyping vision algorithms in functional languages is that most images are two dimensional in nature. One possible means of handling this data is to represent a 2D array of pixels with nested lists:

```
type HIPS_IMAGE = int list list
```

The complication here is that not all the basic operations on lists can be nested, only those that can be partially or fully applied to give a value of the same type as one of their arguments, such as `map` or `flatten`:

```
- map (map I) [[1,2],[3,4]];
val it = [[1,2],[3,4]] : int list list
- flatten (flatten [[[1],[2]],[[3],[4]]]);
val it = [1,2,3,4] : int list
```

Most of the other list operations have to be built from the non-nested list equivalent, for instance the `foldr` function can be generalised to nested lists, see Fig. 6, although in this case there is an equivalent expression using the standard list operations (`foldr ff def (flatten ll)`).

To write generic routines which are polymorphic in terms of nesting would require a higher level of polymorphism than provided by SML. The vision algorithms discussed in this paper have all been implemented using either explicit decomposition of iconic data or a small set of higher-order functions over nested lists, the most salient being `mapll`, `map2ll`, `foldlll`, `foldrll`, `filterll` and `unzipll`‡.

The only slight complication for the Hough transform is that the (x,y) coordinates are required in the application of Eqn. 1 by the transformation function. This is easily handled by the introduction of counters as shown in Fig. 7.

The function `map2xyll` forms the basis for the transformation into Hough space. The transformation function is passed the tuple (`x,y,edge_strength,edge_angle`) and returns a tuple (`x,y,`ρ`,`θ`,edge_strength`). Note that ρ and θ are suitably quantised. This results in a 2D transformed space which is then reduced into the

‡ We generally use the suffix `-ll` to mean the equivalent of a higher-order function on lists generalised to lists of lists.

```
fun map2xyll _ _ _ _ nil = nil
  | map2xyll _ _ _ nil _ = nil
  | map2xyll ff x y (h1::t1) (h2::t2) =
          (map2xy ff x y h1 h2)::(map2xyll ff x (y + 1) t1 t2)

and map2xy _ _ _ _ nil = nil
  | map2xy _ _ _ nil _ = nil
  | map2xy ff x y (h1::t1) (h2::t2) =
          (ff (x,y,h1,h2))::(map2xy ff (x + 1) y t1 t2)
```

Fig. 7. The `map2ll` Function with Counters

Hough space histogram by **`foldrll`**. The histogram itself is a list of complex tuples of the form: **`((`ρ`,`θ`),accum,point_list)`**, where (ρ, θ) is the index of the Hough cell, **`accum`** is the histogram accumulator (using the well-known heuristic of summing the **`edge_strength`** values) and **`point_list`** is the list of (x,y) points accumulated in the (ρ, θ) Hough cell. From this point onwards the normal list processing functions suffice.

For this particular application, developing the prototype involves no more effort than developing normal functional code, once the extensions to the normal list processing functions are available. There is a leap of intuition in realising that, for the type of manipulations required here, 2D data can be treated in exactly the same way as 1D data. Issues relating to the construction of 2D operations from 1D operations are discussed in (Kozato, 1994). It should be pointed out, however, that this will not be the case for all 2D data operations, for instance, had we implemented an edge tracking algorithm instead of the Hough transform none of the foregoing discussion would apply and the code would have been very convoluted. It would also have been much more difficult to maintain correspondence between the prototype and parallel implementation. The key issue is that our decomposition of the iconic data was both regular and not dependent on any particular order of access (row-column versus column-row, for instance).

Handling a method such as edge-tracking really requires unit time random access to the data, which implies some kind of array representation. Although we have overcome this requirement in our application by structuring the algorithm in a specific way there is no doubt that an even more efficient version could be written using New Jersey SML's **`struct Array2`**. We did not investigate arrays because New Jersey SML's implementation is somewhat unwieldy in syntax and is also not referentially transparent. While these could have been overcome with careful coding it is probably not a good idea to let automatic program transformations loose among non-referentially transparent constructs. The prototype would also lose its aesthetically pleasing purely functional nature.

2.2.4 Program Transformation

The two main possibilities for the parallel architecture are a farm of pipelines which corresponds to the following SML skeleton:

```
map (ppfn o acfn o tfn) hough_portions
```

or as a pipeline of farms:

```
(map ppfn) o (map acfn) o (map tfn) hough_portions
```

These architectures represent two extremes of a range of possible implementations but the intermediate possibilities are most likely intermediate in terms of performance and so were not considered. The reason for this is that the same transformation rule can be used to transform one form into any other:

$map\ (f\ o\ g)\ \Leftrightarrow\ (map\ f)\ o\ (map\ g)$

thus:

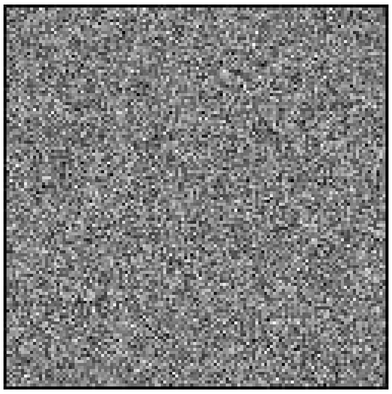
$$
\begin{array}{ll}
 & map\ (ppfn\ o\ acfn\ o\ tfn)\ hough\_portions \\
\equiv & map\ ((ppfn\ o\ acfn)\ o\ tfn)\ hough\_portions \\
\Leftrightarrow & (map\ (ppfn\ o\ acfn))\ o\ (map\ tfn)\ hough\_portions \\
\Leftrightarrow & ((map\ ppfn)\ o\ (map\ acfn))\ o\ (map\ tfn)\ hough\_portions \\
\equiv & (map\ ppfn)\ o\ (map\ acfn)\ o\ (map\ tfn)\ hough\_portions
\end{array}
$$

To decide which construct represents the best solution requires a combination of timing results with static data transfer analysis for each of the components individually. Timing information for sets of representative data gives *relative* computational requirements for each of the components. Static data flow analysis gives an approximation to the communications overhead. Note that, due to the influence of the garbage collector in the SML implementation, the timing results need to be averaged over a significant number of runs. The data flows can be calculated from the type signature in the SML functions. Yet again, we emphasise that these issues would not normally be considered by the prototype implementer since, ideally, they could be automated by a parallelising compiler such as SkelML (Bratvold, 1994).

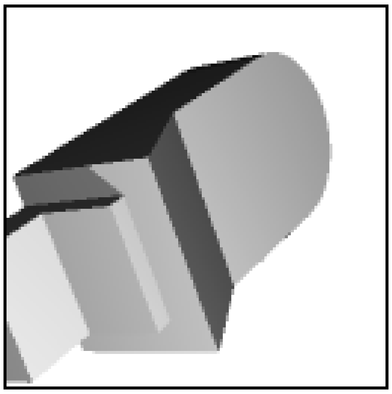
2.2.5 Test Data

The set of test images of dimensions 256 by 256 pixels, which is the maximum image size that can be handled by the SML program is illustrated in Fig. 8. The reason the prototype can only handle 256 by 256 images is due to the size of the intermediate representations indicated in Section 2.2.3. As well as both edge strength (twice) and angle, the (x,y) co-ordinate, the histogram accumulator and the (ρ,θ) Hough cell co-ordinate all have to be simultaneously stored as pixel data. For a 512 by 512 image this is over 12Mb of data which is about the maximum heap size available on our Sparc machines.

Note, however, that the space efficiency of the prototype is not a design criterion. Provided the routines developed on the prototype are scalable, which they are for all the code developed here, then it is acceptable to test the prototype on smaller sets of data. This could be for convenience or because of a system limitation as in this instance. The point is that the test data has to be representative of the data to be processed by the implementation. An image of 256 by 256 pixels is large enough to contain virtually all the information found in, say a 1024 by 1024 image. Reducing an image to much less than 256 by 256, however, would probably result



1) Random



2) Widget

3) Phwidg

4) Postvan

1. A randomly generated image in which each pixel has a normal distribution of intensity, in effect an image consisting solely of noise.
2. A synthetic image of an object comprising mostly of rectangular faces but with one large radius cylinder. This represents the best quality image the system would be presented with in practice.
3. A real (photographic) image of a single object. This image exhibits extremely poor contrast (the image seen here has been enhanced) and has background features that coincide with edges on the object. This is about the poorest quality image of a simple scene with which the system is likely to be presented.
4. A much more complex image with curved surfaces, specular reflections and marked surfaces. This is an unrealistic image for our simple algorithms and in fact even a simple box-shaped model could not be reliably matched with this image.

Note that the reason for this particular set of images is that image 1 represents a null case with important properties for the algorithms we use, images 2 and 3 bracket the useful range of image types that we would expect our methods to handle and image 4 is a pathological case.

Fig. 8. Test Images

Table 1. *SML timing summary*

Mean SML processing time [a] in seconds [b]								
Phase	random		widget		phwidg		postvan	
transform	68.5	(117.2)	29.60	(49.20)	29.90	(50.40)	35.20	(61.50)
accumulate	118.9	(40.2)	1.16	(2.24)	4.89	(16.60)	10.75	(10.70)
post-process	15.7	(17.2)	0.46	(0.39)	0.97	(2.72)	2.32	(3.15)
global merge	345.0	(208.8)	1.62	(1.89)	9.72	(43.80)	17.90	(23.10)

[a] This is the user time for the given phase averaged over a wide range of program parameters (mostly the thresholds), each set of parameters is itself is averaged over ten identical runs

[b] Values in brackets are standard deviations

in undesirable effects, features would become unrecognisable and make higher level processing such as model matching difficult or unreliable.

2.2.6 Prototype Time Profiling

The main problem which became apparent when the SML implementation was run on real data was that the processing time is *very* highly data dependent.

Those portions of Hough space with no peaks require very little processing at all. Other portions with peaks to process have only a small number of lines to process and a few portions of Hough space have a large number of lines, each with possibly several subsegments to process. In practice, with geometrically regular objects virtually all the processing may be concentrated in a small number of Hough space portions.

One symptom is that the timing results are highly dependent upon the program parameters such as threshold values, with the added complication that some parameters affect different parts of the computation than others, for instance the peak threshold has no effect on the accumulation step at all whereas the input threshold can affect any part of the computation. The timing runs were carried out over as wide a variety of parameter combinations as possible and over several different types of image.

The results are summarised in Table 1. Based on a very rough average for this set of test images the transformation stage dominates the others, with a ratio of approximately 30:5:1 for transformation to accumulation to post-processing computation times. Note, however, that the results for the random image (which is by far the most dense image of the set) indicate that this relationship will not always hold.

Table 2. *Static Data Flow Analysis*

Communications loads (typical values)			
Phase	Initialisation Bytes	Input bytes	Output bytes
Transformation	512k	256	128k
Accumulation	4	128k	12k
Post-processing	32	12k	480

```
fun transform2 low rinc tinc ymax eimg aimg hough_portion =
  let
    val tfn = transformfn low rinc tinc ymax hough_portion
    val tdata = map2xyll tfn 0 0 eimg aimg
    fun ffn (x,y,t,r,e) = t <> NODATA andalso r <> NODATA
  in
    flatten (filterll ffn tdata)
  end :
      int -> int -> int -> int ->
        int list list -> int list list ->
          (int * int) * (int * int) ->
            (int * int * int * int * int) list
```

Fig. 9. Hough Transformation Function and Type Signature

2.2.7 Static Data Flow Analysis

The data flow calculations are complicated by the data dependency of the algorithms. Assuming averages based on the images used for the test runs (10% of input pixels thresholded, 16 portions in Hough space, 50 points per detected line and 30 lines per image) gives estimates for the communications load as indicated in Table 2.

Consider the implementation of the transformation function shown in Fig. 9 which has been shown with its type signature. The program works by partially applying `transform2` from the `low` argument up to `aimg`, these parameters are therefore initialisation data. Considering the images are 256 by 256 `int` images this gives a total of $4 \times (4 + (2 \times (256 \times 256))) = 524304$ bytes. In fact the parallel implementation optimises this by encoding byte images. Since this is initialisation data, however, it does not affect the parallel performance.

The partially applied function is then mapped over the list of Hough space definitions. The total amount of data communicated to the worker processors by the farmer processor is thus the number of Hough portions times the size of a Hough portion definition $16 \times (4 \times 4) = 256$ bytes.

The output from the function is a list of tuples of 5 `int`s. The length of this list is one element for each pixel in the Hough portion, less those elements that have been filtered out. The total amount of output from this phase, however, has to be multiplied by the number of Hough portions, ie. over the entire image. Assuming

10% of pixels survive thresholding, this gives $0.1 \times (256 \times 256) \times (4 \times 5) = 131072$ bytes.

Note that once the centre of parallelism has been decided (mapping the partially applied `transform2` over the image) there is nothing in the above calculation that cannot be automated by normal profiling of representative data sets. This is in fact carried out by SkelML.

2.2.8 Parallel Implementation Selection

For the linear pipeline processor farms we use to implement the `map` skeleton a rough rule of thumb is that if the time taken to communicate the data to and from the workers is of the same order as the time taken to process the data then the speedup will very quickly run out of steam as the length of the pipeline increases. For the transformation phase analysis above, the total communications load of 128k can be processed by a transputer link, running at 10Mbits/sec in about 0.1 seconds, ignoring latencies and other communications overheads. This is much less than the 30 seconds that it takes to process the data.

This means that processor farming is viable for the transformation phase. Although we do not prove it formally, there is not enough computation in the accumulation and post-processing phases, relative to the transformation phase, to warrant separate farms or to implement them in a pipeline (which would be difficult to balance). A way of visualising this is to consider different computations competing with each other for limited processing resources.

Given the additional programmer workload associated with hand coding separate farms as opposed to a single farm, the best candidate for our application, is a single farm with each worker processing a complete pipeline.

2.3 The Occam2 Implementation

The work on the SML prototype suggests that the best implementation for the parallel system is a farm of single processor pipelines. The input image should be pre-distributed to the workers along with the program parameters. Definitions of Hough space portions are then sent asynchronously to the workers which can be arranged in a linear pipeline giving a simple communications setup but with primitive inherent load balancing.

The Occam2 implementation was strongly influenced by the SML code. Large sections of SML code were converted into Occam2 by hand giving a close correspondence between the two programs. SML can be compiled into Occam2 (Busvine, 1991) but in this instance the conversion process was carried out by hand to ensure that the prototype and implementation have as similar characteristics as possible to aid the validity of the prototyping method and to appreciate the problems associated with the conversion process.

The single processor farm was implemented using a simple scheduling strategy, without buffering, whereby the transputer channel **in** characteristics are used to

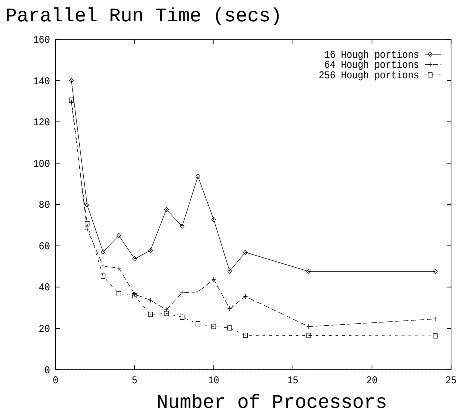


Fig. 10. Parallel Execution Time for the Occam2 Hough Implementation

create a queue of tasks down the pipeline (this actually requires an emulated **out** guard (Burns, 1988)).

The Occam2 implementation was run using values for the parameters that were indicated by the SML prototype. The processor farm was timed using from 1 up to 24 processors and the speedup and efficiency were calculated at each processor count. These are defined as follows:

$$Speedup = \frac{sequential\ run\ time}{parallel\ run\ time} \tag{2}$$

$$Efficiency = \frac{speedup}{number\ of\ workers} \tag{3}$$

Note that our definition of sequential run time is the actual sequential calculation time for the algorithm, not the parallel run time for one processor. This means that, in general, speedup will be less than one for one parallel processor due to the influence of communications overheads.

The parallel run times for tests on the **phwidg** image are presented in Fig. 10, speedup and efficiency values are presented in Fig. 11, The same experiments were carried out using variable numbers of Hough portions (16, 64 and 256) for the farm task descriptions: it became apparent when the Occam2 code was run that the data was very unevenly distributed throughout Hough space, as shown in the uneven plots of speedup and efficiency.

The primitive load balancing provided by the farm implementation is not sufficient to even out the workload for the data being analysed. A more sophisticated load distribution is needed to take into account the amount of data that is distributed with each task rather than naively subdividing Hough space into equal portions.

A simple and easily implemented improvement is to divide Hough space up more finely; however, there is a tradeoff involved. The more finely Hough space is divided between processors, the more likely two portions of the same line are to end up on separate processors which places an added burden on the global merge process.

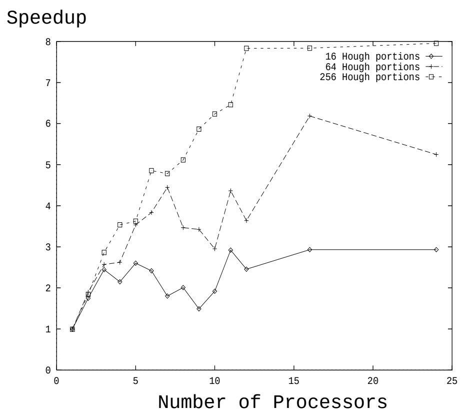


Fig. 11. Performance of the Occam2 Hough Implementation

The assumption is made in the parallel implementation that this phase is much less computationally intensive than the parallel components but experiments with more dense images would suggest that this assumption can break down.

For the values studied (up to 256 portions of Hough space) there was no significant increase in the global component and the results were stable with respect to the program output (ie. lines detected). Given the advantages of speedup and efficiency that are gained this is a worthwhile consideration.

Note, however, that the problems of load balancing were not highlighted by prototyping. The prototype, as implemented here, deals with simple averages and does not pinpoint problems with unusual distributions of data. Perhaps a clue is the high standard deviations for the user timings over a wide range of test runs.

3 Translating SML to Occam2

3.1 Source Language Translation

Programming languages are characterised by their data and control base constructs and abstraction mechanisms. In converting from one language to another, correspondences must be established between source and target language characteristics. Within a paradigm, this is relatively straightforward as there are often direct equivalences. Across paradigms, however, the target language must be used to model explicitly source language characteristics.

Ideally, to translate from one language to another, semantics for both should be defined, the translation process should be formalised and it should be proved that translation preserves semantics, for example as in Stepney's approach to high integrity compilation (Stepney, 1993) or Hammond's DACTL implementation of SML (Hammond, 1991). However, this is a substantial undertaking even for small languages from the same paradigm. Formalisation of translation from a full functional language to a parallel imperative language is a major research activity and beyond the time and resources of our project. While our work is strongly influenced by such formalisation, in particular (Busvine, 1993), here we present an informal overview of our SML to Occam2 conversion.

It is well known that the central issue in constructing imperative implementations of functional languages is the efficient management of name/value associations. In the graph reduction approach (Peyton Jones, 1987), such associations are elided through compilation techniques which identify directly in program graphs the abstraction points requiring specialisation with values. This leads to overheads in graph copying as abstraction points may be specialised with different values in different contexts. Cunning compilation and optimisation techniques are required to minimise such copying. While graph reduction is the natural model for functional language implementation, it corresponds poorly to classic von Neumann architectures and hence to imperative languages. In contrast, the SECD machine approach (Landin, 1964) is based on von Neumann architectures. Here, name/value associations are held explicitly in environments with attendant problems of access and storage management, particularly for closures. However, the SECD machine's close correspondence to the von Neumann architecture suits it well as a basis for the imperative implementation of functional languages, particularly for strict languages. Consequently, we use an SECD like approach in translating SML to Occam2 in that name/value associations are held explicitly in Occam2 variables.

SML is well described elsewhere (Milner *et al.*, 1990): here we will only refer to salient features. We are using a pure functional subset of SML. That is, we do not use any imperative constructs such as assignment, sequencing or iteration, nor do we use SML arrays. However, we do use imperative I/O to interface our programs to the host system. Consequently, we are primarily concerned with the conversion of the boolean, integer, real and string base types, tuples, lists and concrete datatypes to Occam2 equivalents. Similarly, we must find Occam2 equivalences for pattern matching, case structuring, composition and recursion. Finally, abstractions are introduced through global and local definitions, and bound variables in function values.

Occam2 is also well described elsewhere (Inmos Ltd., 1988): once again we will only refer to salient features here. Occam2 provides base types for bytes (ie characters), 16, 32 and 64 bit integers, 32 and 64 bit reals, and booleans. Arrays of base types are the sole data structuring construct. Programs are structured through command sequences, deterministic and non deterministic conditional commands, bounded and unbounded iteration, and parallelism. Abstractions are introduced through global definitions, formal parameters to procedure and function processes, and local definitions at the top level of process bodies. Processes may be composed through explicit calling sequences or coordinated through channel communication. Hence, a sequence of values on a channel is a form of data structuring without explicitly storing the entire sequence.

We translate SML base types to their Occam2 equivalents. Individual tuples are held as separate elements in Occam2 variables. Lists of base types are held as arrays, lists of tuples are held as multiple arrays with one array for each element and lists of lists are held as multi-dimensional arrays. Elements of datatype values are also held as separate variables and lists of datatypes as arrays.

```
let
  val ri = real rinc
in
  let
    val theta = real angleval * 360.0 / 255.0
  in
    let
      val beta = (90.0 - theta) * pi / 180.0
    in
      let
        val rho = real x * cos beta + real yy * sin beta
      in
        let
          val thetastep = floor(theta * real tinc / 360.0 + 0.5)
        in
          let
            val rhostep = floor(rho * ri / 720.0 + ri / 2.0 + 0.5)
          in
            (x,yy,thetastep,rhostep,angleval)
          end
        end
      end
    end
  end
end
```

Fig. 12. SML: Hough Transformation Code

SML pattern matching and case structuring are translated to explicit Occam2 selection and testing in deterministic conditional commands. Function composition is translated to either explicit nested process calling through parameter passing or inter-process communication through channels, depending on whether or not the granularity of the composition is appropriate for single or multiple processor implementation. Recursion outside identifiable skeletons is translated to iteration with an explicit stack to hold intermediate values where necessary. Skeleton recursion is used to instantiate the equivalent Occam2 harnesses if prototype instrumentation suggests useful parallelism is present. Otherwise, iteration with stacks is used. We avoid functional constructs that require closure manipulation.

As suggested above, where function composition is translated to channel based inter-process communication, lists may be transmitted element by element between processes without being held locally in their entirety. However, where a process deals with list elements asynchronously, for example a process farm for map, and where the order of elements is important then the process may need to store substantial portions of the list for output in the required order. Alternatively, list elements may be tagged with order information for asynchronous transmission.

3.2 Conversion of the Hough Transform Prototype

Linear, sequential sections of code can be translated at the syntactic level. Consider the section of SML code in Fig. 12 which can be directly converted into the Occam2 code shown in Fig. 13.

There are a few pitfalls in this process, however. This sequence of `let` constructs

```
INT thetastep, rhostep:
REAL32 theta, beta, rho, ri:
SEQ
  ri := REAL32 ROUND RhoInc
  theta := ((REAL32 ROUND frame[1][y][x]) * 360.0(REAL32)) / 255.0(REAL32)
  beta := ((90.0(REAL32) - theta) * pi) / 180.0(REAL32)
  rho := ((REAL32 ROUND x) * COS(beta)) + ((REAL32 ROUND yy) * SIN(beta))
  thetastep := INT ROUND ((theta * (REAL32 ROUND ThetaInc)) / 360.0(REAL32)
  rhostep := INT ROUND (((rho * ri) / 720.0(REAL32)) + (ri / 2.0(REAL32)))
```

Fig. 13. Occam2: Hough Transformation Code

in SML can be transformed into a single expression by a process of substitution but it is important to maintain the same pattern of single/multiple evaluation in the translation process. For instance, if the `ri` expression were substituted into the `rhostep` expression this would result in a double evaluation which would have to be duplicated between the prototype and implementation.

Another problem is that there are two real number representations in Occam2 whereas only one is provided by New Jersey SML. In terms of precision, `REAL64` is identical to the New Jersey SML reals although in terms of processing time, `REAL32` runs (very approximately) 25-30% faster than `REAL64` for arithmetic operations on T800 transputers. Due to the influence of iteration using residuals as convergents the transcendental functions run about twice as fast for `REAL32` (although this is highly data dependent).

This represents a thorny problem since we would like to use the space efficiency of the `REAL32` representation (vision algorithms generally do not need the precision of 64-bit real numbers) but are constrained by the 64-bit representation in New Jersey SML. We adopt the 32-bit representation in the Hough transform due to the large amount of intermediate data which limits the size of image we can process. We estimate a discrepancy of 49% in the time to process the core of the Hough transformation in 32-bit as opposed to 64-bit real numbers and make the assumption that this does not affect the validity of our prototype.

Minor differences between the prototype and implementation are tolerable, however, since it is the ratio of computation between different parts of the program that is important, not absolute processing times. Provided both prototype and implementation are consistent within themselves these ratios should be preserved to within acceptable limits.

This is an important observation since there will inevitably be differences of one kind or another between two equivalent programs in different paradigms, for instance the `angleval` value in the SML version is provided as a function argument by mapping over a list whereas the Occam2 version has to explicitly index an array within an iterative loop. The code implementing the map function is the logical equivalent of array indexing but they occur at different points in the computation.

Note, however, that it is often the case that the amount of computation in the differences is outweighed by the processing in equivalent code sections, for instance, in the above code, the time to index an array is far smaller than the time to process the floating point operations.

```
fun rshdst ((l1 as (l1p1,l1p2)):rline) ((l2 as (l2p1,l2p2)):rline) =
    let
        fun shproj
            ((l1 as (l1p1 as (l1x1,l1y1),l1p2 as (l1x2,l1y2))):rline)
            ((l2 as (l2p1 as (l2x1,l2y1),l2p2 as (l2x2,l2y2))):rline) =
            let
                fun within t = t >= 0.0 andalso t <= 1.0
                val omc1 = romc l1
                val pp1 = proj_pt_to_line omc1 l2p1
                val pp2 = proj_pt_to_line omc1 l2p2
                val wi1 = rptwithinln pp1 l1
                val wi2 = rptwithinln pp2 l1
            in
                if within wi1
                    then
                        let
                            val dp1 = rdistfn pp1 l2p1
                        in
                            if within wi2
                                then rmin dp1 (rdistfn pp2 l2p2)
                                else dp1
                        end
                    else
                        if within wi2
                            then (rdistfn pp2 l2p2)
                            else mmd
            end (* shproj *)
        val md1 = rmin (rdistfn l1p1 l2p1) (rdistfn l1p1 l2p2)
        val md2 = rmin (rdistfn l1p2 l2p1) (rdistfn l1p2 l2p2)
        val mmd = rmin md1 md2
    in
        rmin (shproj l1 l2) (shproj l2 l1)
    end (* rshdst *)
```

Fig. 14. SML: Shortest Distance Function

The direct equivalence between sequential code segments can be extended by preserving function (`PROC`) boundaries. This does not apply to recursive functions in SML which translate into iteration over suitable variables, but programmers will generally use function abstraction as a mechanism for structuring code and this can be carried across directly from the prototype to the implementation. A modest example is illustrated in SML in Fig. 14 and its Occam2 conversion in Fig. 15. In this example, the `rshdst` function contains a nested function `shproj`, which in turn contains `within`.

The external function calls are not important, this code is given to exhibit how structure is preserved in translation. Here, tuples are translated into separate variables, resulting in unwieldy syntax in the implementation and conditionals are translated across directly, due to the identical semantics of conditionals in each language.

The conversion from recursion to iteration is again illustrated with a simple example. Figs. 16 and 17 show how a summation over a set of 2D coordinates (part of a least squares routine) can be implemented by `foldl` with a suitable function in SML and using iteration in Occam2. Although this is a trivial example the conversion is quite general in that complex nested recursive functions are converted

```
{{{ PROC rshdst(l1x1,l1y1,l1x2,l1y2,l2x1,l2y1,l2x2,l2y2,dist)
PROC rshdst(VAL REAL32 l1x1,l1y1,l1x2,l1y2,l2x1,l2y1,l2x2,l2y2,
            REAL32 dist)

  REAL32 shl1l2,shl2l1,md1,md2,mmd:
  {{{ PROC shproj(l1x1,l1y1,l1x2,l1y2,l2x1,l2y1,l2x2,l2y2,shdst)
  PROC shproj(VAL REAL32 l1x1,l1y1,l1x2,l1y2,l2x1,l2y1,l2x2,l2y2,
              REAL32 shdst)

    REAL32 grad1,inter1,pp1x,pp1y,pp2x,pp2y,wi1,wi2,dp1:
    BOOL HV1:
    {{{ BOOL FUNCTION within(t)
    BOOL FUNCTION within(VAL REAL32 t)

      VALOF
        SKIP
          RESULT (t >= 0.0(REAL32)) AND (t <= 1.0(REAL32))

    :
    }}}
    SEQ
      romc(l1x1,l1y1,l1x2,l1y2,HV1,grad1,inter1)
      ProjPtToLine(l2x1,l2y1,HV1,grad1,inter1,pp1x,pp1y)
      ProjPtToLine(l2x2,l2y2,HV1,grad1,inter1,pp2x,pp2y)
      wi1 := rptwithinln(pp1x,pp1y,l1x1,l1y1,l1x2,l1y2)
      wi2 := rptwithinln(pp2x,pp2y,l1x1,l1y1,l1x2,l1y2)
      IF
        within(wi1)
          SEQ
            dp1 := rdist(pp1x,pp1y,l2x1,l2y1)
            IF
              within(wi2)
                shdst := rmin(dp1,rdist(pp2x,pp2y,l2x2,l2y2))
              TRUE
                shdst := dp1
        TRUE
          IF
            within(wi2)
              shdst := rdist(pp2x,pp2y,l2x2,l2y2)
            TRUE
              shdst := mmd

  :
  }}}
  SEQ
    md1 := rmin(rdist(l1x1,l1y1,l2x1,l2y1),rdist(l1x1,l1y1,l2x2,l2y2))
    md2 := rmin(rdist(l1x2,l1y2,l2x1,l2y1),rdist(l1x2,l1y2,l2x2,l2y2))
    mmd := rmin(md1,md2)
    shproj(l1x1,l1y1,l1x2,l1y2,l2x1,l2y1,l2x2,l2y2,shl1l2)
    shproj(l2x1,l2y1,l2x2,l2y2,l1x1,l1y1,l1x2,l1y2,shl2l1)
    dist := rmin(shl1l2,shl2l1)

:
}}}
```

Fig. 15. Occam2: Shortest Distance Function

```
fun xysums (Sx,Sy,Sxy,Sxx,Syy) ((x,y):rpnt) =
    (Sx + x,Sy + y,Sxy + (x * y),Sxx + (x * x),Syy + (y * y))

val (Sx,Sy,Sxy,Sxx,Syy) =
    foldl xysums (0.0,0.0,0.0,0.0,0.0) coords
```

Fig. 16. SML: Summations

```
{{{ PROC RCalcXYsums(X,Y,N,Sx,Sy,Sxx,Sxy,Syy)
PROC RCalcXYsums(VAL []REAL32 X,Y,VAL INT N,REAL32 Sx,Sy,Sxx,Sxy,Syy)

  REAL32 x,y:
  SEQ
    SEQ n = 0 FOR N
      SEQ
        x,y := X[n],Y[n]
        Sx,Sy := Sx + x,Sy + y
        Sxx,Sxy,Syy := Sxx + (x * x),Sxy + (x * y),Syy + (y * y)

:
}}}

Sx,Sy := 0.0(REAL32),0.0(REAL32)
Sxx,Sxy,Syy := 0.0(REAL32),0.0(REAL32),0.0(REAL32)
RCalcXYsums(xpts,ypts,pts,Sx,Sy,Sxx,Sxy,Syy)
```

Fig. 17. Occam2: Summations

into similarly nested iteration. The only complication, as shown in the example, is that default values for recursion have to be replaced by explicit initialisation of variables. Mutually recursive functions are more problematical, however, but could be handled by additional control variables in the iteration.

The conversion of higher order functions into parallel constructs has been explained in more detail elsewhere (Wallace *et al.*, 1992). Summarising this work, we convert the `map` function into a processor farm with one master processor and a linear pipeline of worker processors. This construct is simple to implement and has primitive load balancing but there is a practical limit to the length of the pipeline that can be used which is much smaller than the limitations of more dense arrangements of processors such as 2D grids or toroids.

The `map` function is translated into this construct by implementing the `map` function argument on the workers and decomposing the list over these workers. There are a number of design decisions that have to be taken at this point, however.

For low-level operations such as convolving a filter with an image, where communication dominates over computation, geometric decomposition generally works better than processor farming. For higher level operations, such as scanning a list of features and calculating transforms between pairs of features, computation dominates over communication and the load balancing inherent in processor farming becomes more effective relative to naive geometric decomposition.

Another issue is preloading of data. The list to be decomposed can be built into the work definition sent from the farmer to the workers or it can be distributed among the workers prior to beginning the computation. If communications times are a substantial fraction of the overall processing time, preloading of the list data may prove effective. This is partly because saturated communications can cause remote workers to be starved of work by workers physically nearer the farmer and also because optimisations can be made during initialisation, such as image compression, which is more effective when applied to a complete image.

Note also that preloading of data can be controlled at the source code level by program transformation, thus the following SML construct:

```
let
  f1 = f x
in
  map f1 l
end
```

corresponds to preloading, whereas the following equivalent section of code:

```
map (f x) l
```

corresponds to distribution of program data during computation. The idea here is that if the function's argument is applied in the skeleton instantiation `(f x)` then the implication is that the argument should be distributed with the list data whereas in the first version the function and data are bound together `f1 = f x` implying both should be present on the worker processor when `map` is evaluated.

In the absence of an accurate performance model for the map skeleton we base our implementations on the predicted ratio of communications to computation. Very roughly, if the predicted communications time is anywhere near the order of magnitude of the processing time then geometric decomposition is used in conjunction with preloading of data.

The extensions to `map` such as `map2` and `mapll` are currently implemented using exactly the same parallel code since multiple argument lists can be `zip`ed up into a list of tuples and nested lists can be `flatten`ed into a single indexed list. There are opportunities, however, for the additional structure in these functions to be exploited in the parallel code, for instance `mapll` could be implemented as a farm of farms on a 2D grid of processors.

We instantiate our central Hough transform implementation, ie. mapping the `transform2` function shown in Section 2.2.7 over the list of Hough portion definitions, with the Occam2, illustrated in pseudo-code, in Fig. 18. This also illustrates partial application and preloading of data.

3.3 Validity of the Conversion Process

We have shown by example that SML can be converted into Occam2 in such a way that the overall characteristics of the prototype are retained in the implementation. Note, however, that this process is not perfect and that, although the previous examples flatter the conversion process, the original SML and the resulting Occam2 do have significant differences, for instance modelling lists with arrays presents some problems due to the different time complexities of some of the basic operations.

An exact correspondence between the two is not necessary, however. Provided the

```
PAR

  -- Farmer processor
  SEQ
    ... Broadcast program parameters and images to workers
    ... Split Hough space up into rectangular portions
    PAR
      SEQ c = 0 FOR ColPortions
        SEQ r = 0 FOR RowPortions
          ... Send work request for Hough portion [c][r]
      WHILE PortionsProcessed < NumberHoughPortions
        SEQ
          ... Read message back from workers
          IF
            ... Detected line returned: add to global line list
            ... Portion processed: increment PortionsProcessed

  -- Worker processor
  SEQ
    ... Read message from farmer processor
    IF
      ... Program parameters and images: update local copy
      ... Work request: return lines then portion processed message
```

Fig. 18. Occam2: Processor farm implementation

major components perform the same calculations in the same order there is considerable latitude in the details of the computation. In fact, during hand conversion of code it is difficult to resist performing the obvious optimisations that become apparent, for instance folding multiply nested function calls into a single iteration loop. This is perfectly acceptable and, indeed, desirable since in the context of an automated parallelising compiler it is possible that an optimiser could be applied to the output code, although the optimisations would have to be defined so as not to affect the validity of the prototyping.

4 Combining Prototypes into a Complete System

4.1 The Existing Software Components

The three existing components are the Canny edge detector (Koutsakis, 1993), the Hough straight line detection process already described and the perspective inversion (PIA) model matching program (Waugh *et al.*, 1990). These three items together comprise the backbone of a complete visual recognition system, albeit with rather naive and simple algorithms. The block diagram of such a system is presented in Fig. 19.

Originally the Canny edge detector was studied in comparison with the more primitive Laplacian and Sobel edge detectors (Koutsakis, 1993). For our system, the Canny method (Canny, 1986) is the only realistic choice, since it gives the best results in the presence of noise and has other desirable properties for later processing of images: for example non-maximal suppression ensures that there are no multiple responses to single lines in an image. The convolution operations that are the core of all the edge detectors studied were themselves subjected to analysis. They involve

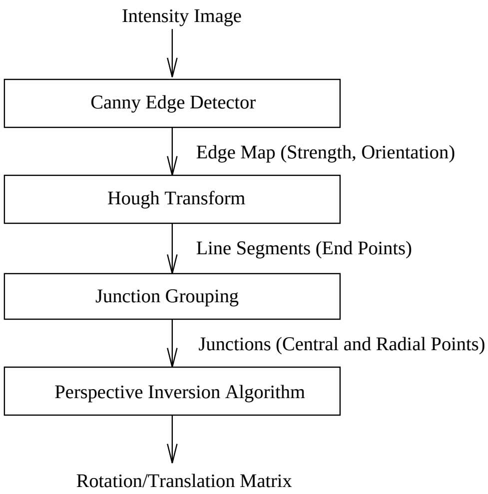


Fig. 19. Block Diagram of the Complete System

passing a square window over the image performing various calculations that lead to a value for a single pixel. Several different ways of decomposing this problem into concurrent tasks were studied.

As with the Hough implementation there are multiple phases; smoothing, edge detection and non-maximal suppression. There is a complication in this instance since these operations are maps over regions rather than pixels. This opens up the possibility of processor *factories*, ie. farming farmer processors, but the best implementation produced by the Canny study uses a single processor farm with preloading of the input image. This is probably due to the dominance of communication in low-level processing.

The perspective inversion algorithm was analysed in detail on sequential systems (McAndrew and Wallace, 1989) but was subsequently implemented in a simpler form as an SML prototype and Occam2 implementation (Waugh *et al.*, 1990). The SML prototype enabled the identification of four major components suitable for implementation on processor farms:

1. `generate.triples`: generation of point triples, the PIA algorithm matches vertices defined by three connected points, taken from the lines detected in the scene and from the model description.
2. `RT.calc`: calculation of the rotation/translation matrix from the model into the scene based on a pair of vertices, one from the scene and one from the model. This is the eponymous perspective transformation.
3. `score`: calculating a score for each match calculated by the `RT.calc` function, based on distances between nearly matching points (within 10 pixels).
4. `data.capture`: comparing the scores for all the generated triples to determine the match with the highest score.

The exact function of each of these phases is not important, they represent a similar decomposition of the overall algorithm to that decribed for the Hough transform in this paper. The `generate.triples` and `data.capture` operations did not contain enough calculation to be worth implementing in parallel. The `RT.calc` and `score` operations were found from the SML prototype (and subsequently the Occam2 implementation) to require processing times in the ratio of about 1:5 for efficient implementation. Combining this with static data type analysis led to an implementation as a pipeline of two farms, one for the `RT.calc` operation and one for the `score` operation with worker processors in the ratio of 1:5.

4.2 Combining the Three Components

Merging different components that have been developed using the SML prototyping and Occam2 implementation method has not previously been attempted. The approach here was from the point of view that each software component represents a black box which could have been developed within an overarching abstraction mechanism, for instance abstract data types. This causes problems in that performing a global optimisation may require breaking open the abstraction boundary to gain the necessary information (Bastani *et al.*, 1987).

The number of architectural choices available is also restricted. Here, the only realistic option is to connect the software components in a pipeline. However, global optimisation is simplified and can be performed by re-allocation of processing elements.

Note that there is actually a missing component (Fig. 19). The Hough program outputs end point specified line subsegments whereas the PIA program requires junctions in the form of point triples. The methods required to perform the conversion are already available and have been well studied. The junctions can be formed by an approach developed by McAndrew (McAndrew, 1990) which is designed to give less weight to co-linear lines. The conversion to point triples had already been written as part of the perspective inversion program (Waugh *et al.*, 1990) to allow input from model description files. The junction grouping phase was implemented as a sequential component in the initialisation of the PIA process and thus appears as a communications overhead in the overall performance of the system.

It is interesting to note that this missing phase could be spotted by considering the type information from the program modules: the output type from the Hough straight line detection algorithm did not match the input type of the perspective inversion program. This is not significant in a simple system with only three components but illustrates the importance of strong typing in the development of larger systems where type analysis can be used as a means of defining the interface between software components.

The SML implementations were very quickly merged into a single program. This process is not of any great utility in terms of providing predictive instrumentation for the parallel version of the complete system: such information was already available from the discrete prototypes. The combined SML code would be of greater value in an environment where it can be directly converted into parallel implementations by automated software tools.

The Occam2 code took considerably longer to assemble but was aided by the work

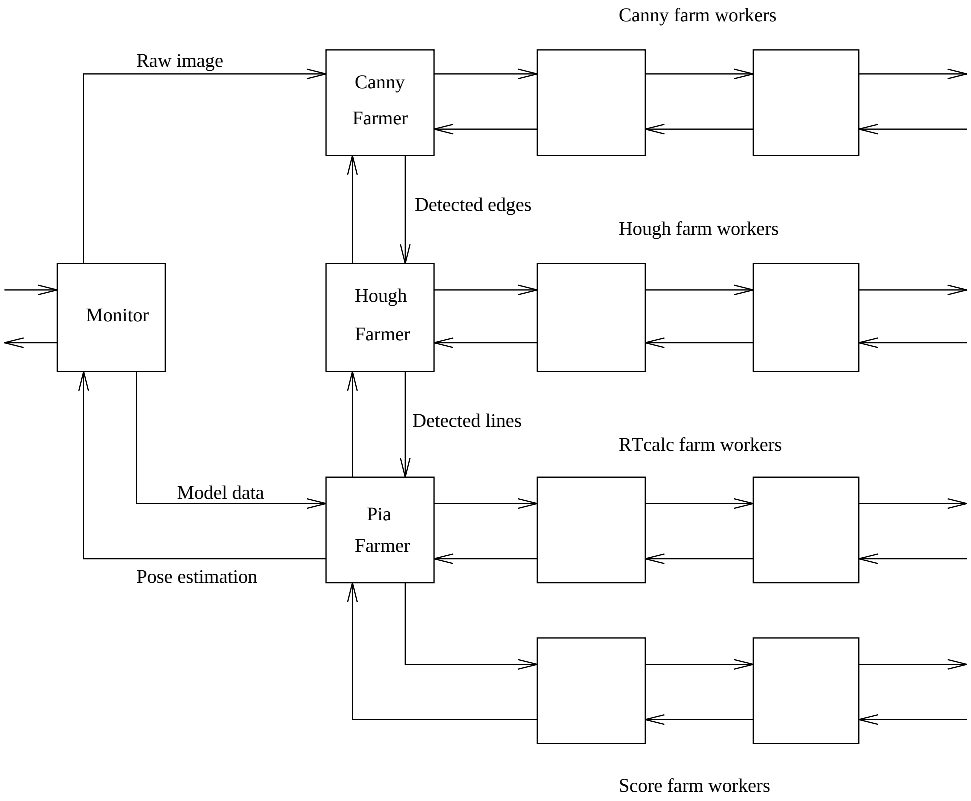


Fig. 20. The architecture of the object recognition system

carried out combining the prototypes. The hardware architecture of the parallel system is presented in Fig. 20. Each of the major components implemented in parallel are serviced by a single, dedicated farmer processor. The farmers each control linear pipelines of processors and are themselves connected in a pipeline. The first farmer in the pipeline reads its data from a central monitor processor and the last farmer reports the final results back to this processor. The monitor processor is also responsible for reading the raw image and model description data from disc files, writing the final results to a disc file and transmitting any initialisation data to the farmers.

The main effort required to combine the Occam2 implementations was in writing the code to monitor and control the farmer processors arranged in a pipeline. This was aided by direct communication between the individual farmers.

5 Performance of the Combined System

5.1 Optimising the Occam2 Implementation

Note that we are not trying to optimise the performance of the system for any particular image. We are interested in studying the effect of combining *existing* implementations, with their architecture intact, on the overall performance of the

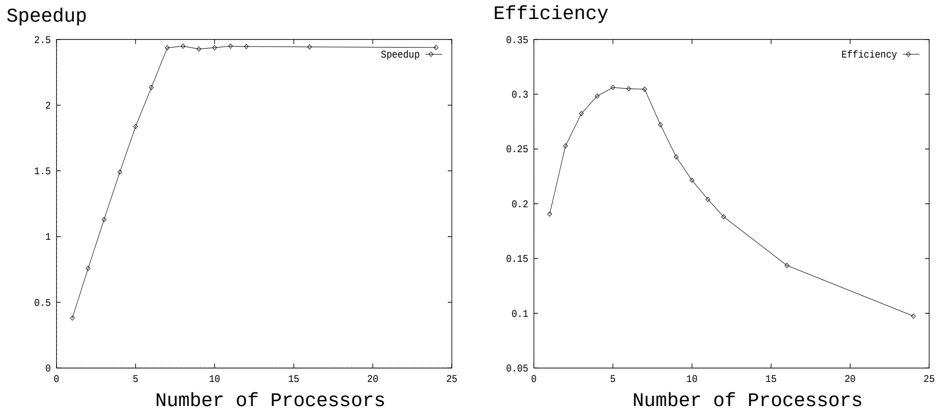


Fig. 21. Performance of the Occam2 Canny Implementation

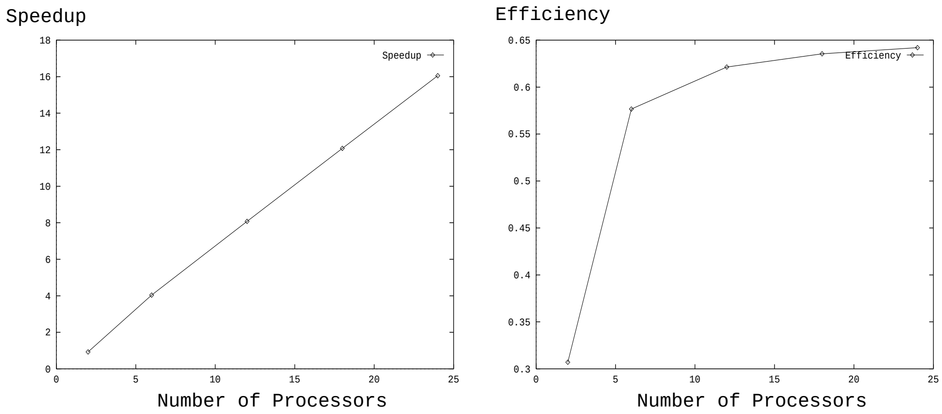


Fig. 22. Performance of the Occam2 Pia Implementation

system. Since the top level of parallelism consists of a pipeline and since we are unable to reallocate processing resources during processing we are thus looking for the best ratio of processors between the components of the pipeline. Since we can only do this as an average over a wide range of images this ratio of processors will therefore merely be an informed guess as to the optimal performance of the system.

The optimisation was attempted for one particular image, the phwidg image, which was chosen as being the most typical image for real applications using the image processing methods employed. The speedup and efficiency figures for the Canny and Pia farms are presented in Fig.s 21 and 22. The results for the Hough farm have already been presented in Fig. 11.

The peaks for speedup and efficiency for each of the three farms (deduced directly from the graphs) are presented in Table 3 along with the measured sequential processing time for each phase.

As an approximation to an optimised processor placement, the processors could be allocated in ratio to the number of processors for optimal speedup (or efficiency) or in proportion to the sequential workload. Assuming 26 processors are available

Table 3. *Optimum speedup and efficiency for individual farms*

	Maximum Speedup			Maximum Efficiency			
Phase	No. procs	Sp	Ef	No. procs	Sp	Ef	Seq. Time (secs)
Canny	5	1.8	0.3	7	2.5	0.3	38
Hough	4	3.6	0.7	12	8	0.6	129
Pia	24	16	0.65	24	16	0.65	2119

Table 4. *Optimum speedup by processor reallocation*

Canny	Hough	Score	RTcalc	Speedup	Efficiency	Parallel Time (s)
2	4	3	17	10.54	0.35	217.0
3	3	3	17	9.81	0.33	177.3
3	4	2	17	9.42	0.31	184.3
3	4	3	16	9.99	0.33	174.1
1	5	3	17	9.20	0.31	189.3
2	5	2	17	9.94	0.33	229.7
2	5	3	16	9.83	0.33	177.0
1	4	4	17	9.29	0.31	187.6
2	3	4	17	9.65	0.32	180.6
2	4	4	16	10.04	0.33	173.5
1	4	3	18	9.16	0.30	190.1
2	3	3	18	9.51	0.32	183.0
2	4	2	18	9.33	0.31	186.1

for farm workers this gives approximate proportions of 4:7:15 (Canny:Hough:Pia) for maximum speedup in the individual farms, 4:3:19 for maximum efficiency and 1:2:23 based on sequential workload. The distribution of workers between the two farms in the PIA phase was fixed as close as possible to 1:5 (RTcalc:Score) since this has already been established (Waugh *et al.*, 1990) as the most efficient balance for these farms.

Table 4 shows that there is a local maximum in speedup with processor allocations of 2:4:3:17 (Canny:Hough:RTcalc:Score) inasmuch as moving one processor from one farm to any other farm lowers the overall speedup. This figure was arrived at by a simple hill-climb search and converged to the maximum in ten iterations. There is no guarantee that this is a global maximum, bearing in mind the instability of the Hough farm's speedup with respect to processor count. The problem of false maxima might be alleviated if the Hough phase had better load balancing. Note that these results were all found from the same image and it is likely that different processor placements would be optimal for different images.

Of the three theoretical values the one based on sequential workload is closest to the actual maximum although there are too few processors in the system for an accurate prediction based on such simple measures. However, all three predictions are close enough to the actual optimum to allow quick convergence in the simple heuristic search applied here, starting from one of the calculated optima.

More generally, a speedup of 10.5 for a system with 30 processing elements (26 farm workers, 3 farmers and 1 monitor processor) is very poor: the individual farms are operating sub-optimally in the optimal configuration. This is probably due to the influence of the global communications and computations phases interspersed between the farms but a more detailed analysis would be required to corroborate this.

5.2 Comparison of the Prototype and Implementation

Very roughly, the SML Hough transform prototype took 15 programmer days to write. This includes basic research plus investigations into existing Hough transforms written in C. The Occam2 Hough implementation was written in about 24 programmer days but this was based strongly upon the SML prototype. Combining the three SML prototypes into a single system took 10 programmer days but, again, this included some preliminary familiarisation with the existing implementations (approximately 2 days), plus writing the missing junction grouping module (approximately 4 days). Combining the Occam2 implementations took 19 programmer days, of which around half was spent modifying the existing top-level code in each implementation to allow meshing together of the components.

Note that the above programmer times all relate to the same programmer, familiar with both SML and Occam2. The author was also well-versed in both functional languages and image processing. No equivalent figures are available for the Canny and PIA work, although both authors can be assumed to be familiar with functional languages and had access to image processing experts throughout program development.

The SML implementation comprises approximately 1700 lines of code, excluding comments and the Occam2 implementation comprises about 3900 lines of code, excluding comments but including configuration data.

Although it takes much less time to develop a system in SML than in Occam2, in the absence of experience in developing equivalent code in native Occam2 without benefit of prototyping, little can be said about whether the combined time to develop both the prototype and the implementation is less than the time required to develop the final system from scratch without prototyping. Bearing in mind that experimentation with different architectures is much easier in SML than in Occam2, where there are no abstraction facilities at all, and from experience on this and similar projects, a rough impression by the authors is that the two take a similar amount of time, to within say 30% of each other.

Development time is not the main advantage of the prototyping method, however, this is more about the reliability of the parallel performance and the potential for ongoing development, for instance to cope with new data sets. In testing a modification on the SML prototype, if it can be shown that there are no performance improvements or other benefits then the Occam2 implementation will be left unchanged. The same process carried out on the implementation would take considerably longer. More generally, the recognised benefits of functional languages are being made available in a parallel context.

As a brief check on the validity of the prototype as a predictor for implementation performance, we consider the ratios of sequential processing times between the different phases, in the same language. This ratio for the **phwidg** image, between the Hough and PIA phases is 1:16.3 for both the SML and Occam2 versions. The fact that these two ratios are so similar is a coincidence, however, one would expect and tolerate significant differences in this ratio without questioning the validity of the prototyping method.

The ratio between the Canny and Hough phases is 4.74:1 for the SML version and 0.30:1 for the Occam2 equivalent. This discrepancy is because the edge detector operates extensively upon 2D array structures modelled as lists of lists and the mode of access is not regular enough to iron out the difference in element selection time for lists and arrays. This, perhaps, points out the need for special considerations where prototype language constructs have completely different characteristics from the implementation architecture. More sophisticated modelling methods might compensate for this discrepancy or alternatively, for our application, functional purism could be abandoned and SML arrays used in prototyping.

This problem does not arise for the Hough or PIA implementations. Output partitioning enables representation of Hough space as a sparse linear list of accumulators so different access times become less significant. The PIA operates within the SML to Occam2 conversion outlined in Section 3.2, data is accessed in a way that allows identical processing in SML lists and Occam2 arrays.

Note that the sequential processing times were based on SML user times with garbage collection and system time removed and the Occam2 times have no communications overheads. The SML code was timed on a Sun Sparc 10 system running New Jersey SML Version 0.93 and the Occam2 code was run on a Meiko Computing Surface with T800 transputers.

6 Conclusions

We have successfully constructed a primitive but fully functional object recognition system in Occam2 through SML prototyping. The parallel implementation was based closely on the functional prototype, and the prototypes' behaviour was the primary source of guidance for parallelisation. The system was formed through the integration of existing components: the performance of the combined system is acceptable though not as good as might be expected from a uniformly developed system.

Functional prototyping proved a valuable design approach. In particular, the ease of program transformation enabled experimentation with different but mutually consistent designs. Prototype instrumentation was found to be a strong indicator for the behaviour of the equivalent parallel system. Extant prototyping information for individual modules was reused successfully without the need to instrument the combined prototype.

The employment of a purely functional style was found to be extremely important in making the prototyping process work. The speed and reliability of programming in a declarative style allowed all the algorithmic design decisions to be made at the prototype stage where mistakes could easily be corrected and alternatives explored. This allowed work during the implementation phase to concentrate on optimising the parallel performance.

The main problem in prototyping was caused by the representation of two dimensional images as nested lists. In general there is no simple relationship between nested list and array behaviour for arbitrary algorithms. This was overcome for the Hough transform prototype by organising the algorithm so that normal list processing style functions could be used, giving a good correspondence to the equivalent Occam2 array behaviour.

Combining SML components proved substantially easier than combining the equivalent Occam2 processes. Two SML components may be integrated through function composition provided they are type consistent. However, combining Occam2 processes involves the explicit construction of a linking harness. Although in our case the individual components' instrumentations were used as the basis for the combined system, in general a better optimisation might be achieved by re-instrumenting the combined prototype.

It is difficult to say whether it takes less time to develop both a prototype and implementation than to implement the final system directly in Occam2 but we are convinced that there is no significant development time increase when using the prototyping method. Combining the prototypes is a worthwhile exercise in any case. Apart from the benefits of functional prototyping already described, any additional code to glue the existing components together may be identified at this stage. An example of this is the use of SML's strong typing system in defining the linkages between different modules where type mismatches highlight missing code.

This work is being followed by the development of a much larger system which will identify 3D objects from depth, intensity and fused images. The new system will be more sophisticated, using robust statistical methods for image processing, and dynamically controlled so that resources will be reallocated during the course of processing. It is being developed from a uniform abstract specification (Austin and Scaife, 1994).

In conjunction with the development of the new system, investigations are under way into the use of abstract data types (ADTs) in managing the complexity of large functional prototypes. Instrumentation will help determine the appropriate placement of ADT methods relative to the components that initiate them, in parallel realisations of ADT based prototypes. The construction of the three stage vision system discussed above enabled preliminary experimentation with this approach.

Finally, a skeleton based compiler from SML to parallel Occam2 is under development (Bratvold, 1993). A direct comparison is planned between hand coded Occam2 implementations and the results of automatically converting the corresponding SML prototypes.

The code for both the SML prototype and Occam2 implementations are available by ftp from ftp.cee.hw.ac.uk in the directory pub/vision/jfp.95. The PIA section of the SML has already been used in benchmarking a garbage collector for the Standard ML of New Jersey (Tarditi and Diwan, 1993).

Acknowledgements

This work is supported by EPSRC grant GR/J07884. We would also like to thank our colleagues in the Heriot-Watt Vision Group for advice and suggestions.

References

A.A. Amini, T.E. Weymouth, and D.J. Anderson. A parallel algorithm for determining two dimensional object positions using incomplete information about their boundaries. *Pattern Recognition*, 22(1):21–28, 1989.

W. J. Austin, A. M. Wallace, and V. Fraitot. Parallel Algorithms for Plane Detection using an Adaptive Hough Transform. *Image and Vision Computing*, 9(6):372–384, 1991.

W. J. Austin and N. R. Scaife. Reconfigurable Parallel Vision System: Informal Specification. Technical Report RM/94/4, Dept. of Computing and Electrical Engineering, Heriot-Watt University, Apr 1994.

P. R. Bailey and M. C. Newey. An Extension of ML for Distributed Memory Multicomputers. Technical report, Department of Compuuter Science, Australian National University, 1994.

D. H. Ballard. Generalising the Hough Transform to Detect Arbitrary Shapes. *Pattern Recognition*, 13:111–122, 1981.

F. Bastani, W. Hilal, and S. Sithrama Iyengar. Efficient Abstract Data Type Components for Distributed and Parallel Systems. *IEEE Computer*, pages 33–44, 1987.

B. Bhanu and L.A. Nuttall. Recognition of 3D objects in range images using a butterfly processor. *Pattern Recognition*, 22(1):49–64, 1989.

T. Bratvold. A Skeleton-Based Parallelising Compiler for ML. In R. Plasmeijer and M. van Eekelen, editors, *Proceedings of the Fifth International Workshop on Implementation of Functional Languages, Nijmegen, the Netherlands*, pages 23–33, Sep 1993.

T. Bratvold. *Skeleton-based Parallelisation of Functional Programs.* PhD thesis, Dept. of Computing and Electrical Engineering, Heriot-Watt University, 1994.

A. Burns. *Programming in Occam2.* Addison-Wesley, 1988.

D. Busvine. Translation of SML to Sequential Occam2. Technical Report TR91/7, Dept. of Computing and Electrical Engineering, Heriot-Watt University, Riccarton, Edinburgh, 1991.

D. Busvine. *Detecting Parallel Structures in Functional Programs.* PhD thesis, Heriot-Watt University, Riccarton, Edinburgh, 1993.

J. Canny. A Computational Approach to Edge Detection. *IEEE Trans. Pattern Analysis and Machine Intelligence*, PAMI-8:679–698, 1986.

Y. Cohen, S. Landy, M. Pavel, and G. Sperling. *HIPS: Image Processing Under Unix Software and Applications.* Human Information Processing Laboratory, Department of Psychology, New York University, 1982.

M. I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* Pitman/MIT, 1989.

M. D. Cripps, J. Darlington, A. J. Field, P. G. Harrison, and M. J. Reeve. *The Design and Implementation of ALICE: a Parallel Graph Reduction Machine*, pages 300–326. 1987.

J. Darlington, A. J. Field, P. G. Harrison, P. H. J. Kelly, D. W. N. Sharp, and Q. Wu. Parallel Programming Using Skeleton Functions. In A Bode, M Reeve, and G Wolf, editors, *PARLE '93 Parallel Architectures and Languages Europe, Munich, Germany*, pages 146–160. Springer-Verlag, LNCS 694, 1993.

S. A. Dudani and A. L. Luk. Locating straight-line edge segments on outdoor scenes. *Pattern Recognition*, 10:145–147, 1978.

K. Hammond. *Parallel SML: a Functional Language and its Implementation in Dactl.* Pitman, London, 1991.

K. Hammond. Parallel Functional Programming: An Introduction (invited paper). In *Proceedings of PaSCo94, Linz, Austria.* World Scientific Publishing Company, September 1994.

P. V. C. Hough. Method and Means for Recognising Complex Patterns. U.S. Patent No. 3069654, 1962.

J. Illingworth and J. Kittler. SURVEY: A Survey of the Hough Transform. *CVGIP*, 44:87–116, 1988.

Inmos Ltd. *Occam2 Reference Manual*, 1988.

Simon L Peyton Jones, Chris Clack, Jon Salkild, and Mark Hardie. GRIP - A High-Performance Architecture for Parallel Graph Reduction. In G Kahn, editor, *Functional Programming Languages and Computer Architecture*, pages 98–112. Springer-Verlag, Berlin, September 1987.

P. H. J. Kelly. *Functional Languages for Loosely Coupled Microprocessors.* PhD thesis, Imperial College, Univ. of London, 1987.

J. Kittler and M. J. B. Duff. *Image Processing System Architectures.* Research Studies Press, 1985.

G. Koutsakis. Parallel Low Level Vision from Functional Prototypes. Master's thesis, Dept. of Computing and Electrical Engineering, Heriot-Watt University, Riccarton, Edinburgh, Oct 1993.

Y. Kozato. *Lazy Image Processing: An Investigation into Applications of Lazy Functional Languages in Image Processing.* PhD thesis, University of London, 1994.

P. J. Landin. The Machanical Evaluation of Expressions. *Computer J.*, 6(4):308–320, 1964.

V. F. Leavers. Survey: Which Hough Transform? *CVGIP: Image Understanding*, 58(2):250–264, Sep 1993.

R.A. Lotufo, E.L. Dagless, D.J. Milford, A.D. Morgan, J.F. Morrissey, and B.T. Thomas. Hough transform for transputer arrays. In *Proc. of 3rd Int. Conf. on Image processing and its Applications*, pages 122–130, Warwick, UK, 1989.

M. D. May and R. Shepherd. Communicating Process Computers. Technical Note 22, Inmos Limited, Bristol, 1987.

P. McAndrew. *Recognising and Locating Objects in Two Dimensional Perspective Views.* PhD thesis, Heriot-Watt University, Dept. of Computing and Electrical Engineering,, Sep 1990.

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.

R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

S. Peyton Jones. *The Implementation of Functional Languages.* Prentice-Hall, 1987.

A. Rosenfeld, J. Ornelas, and Y. Hung. Hough transform algorithms for mesh-connected SIMD parallel processors. *Computer Vision, Graphics and Image Processing*, 41:293–305, 1988.

S. Stepney. *High Integrity Compilation: A Case Study.* Prentice-Hall, 1993.

D. Tarditi and A. Diwan. The Full Cost of a Generational Copying Garbage Collection Implementation. Technical report, School of Computer Science, Carnegie Mellon University, Aug 1993.

A M Wallace, G J Michaelson, P McAndrews, K G Waugh, and W J Austin. Dynamic Control and Prototyping of Parallel Algorithms for Intermediate- and High-Level Vision. *IEEE Computer*, 25(2), February 1992.

K. Waugh, P. A. McAndrew, and G.J. Michaelson. Parallel Implementations from Functional Prototypes: A Case Study. Technical Report TR90/4, Heriot-Watt University, Riccarton, Edinburgh, 1990.