Prototyping an Intensity-Based Vision Recognition System

RM/94/3

N.R. Scaife
Dept. of Computing and Electrical Engineering
Heriot-Watt University
Edinburgh EH14 4AS

February 7, 1995

Abstract

Previous work at this department has been directed towards a method of producing efficient parallel implementations from functional prototypes using algorithms from the field of computer vision as applications. There is existing code in both a functional language (SML) and in an intrinsically parallel language (Occam2) for a Canny based edge detector [Kou93] and a perspective inversion based model matching programme [WMM90]. This report presents a similar analysis of a Hough transform based segmentation algorithm, prototyped in SML and then implemented in Occam2.

The process of prototyping a single algorithm and converting this prototype into a parallel implementation has been studied in detail [Bus93, Bra92]. There has been little experience, however, in combining existing implementations into an overall system. The three individual components (edge detection, feature segmentation and model matching) were combined into a complete intensity based scene interpretation programme. This was carried out, firstly in SML and then in Occam2, in such a way that the efficiency of the overall system was maintained with respect to the efficiency of the individual components.

1 Introduction

1.1 Functional Languages

Functional languages have many advantages over traditional imperative languages. When writing complex software it is important that the code be well-structured which facilitates the development and maintenance processes. Functional languages have a number of features which allow a high degree of modularity to be built into functional programmes. One of these is higher-order functions which has been suggested as one of the main reasons for the importance of functional languages [Hug89].

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

fun reduce ff [] = raise ReduceError
  | reduce ff [h] = h
  | reduce ff (h::t) = ff h (reduce ff t)
```

Figure 1: The `map` and `reduce` functions in SML

An important corollary of the modularisation of functional languages is that higher-order code can be instantiated with different parameters thus allowing a high degree of code reuse. For example, consider the higher-order functions `reduce` and `map`, defined in **SML** in Figure 1. These functions conceptually apply a function `ff` to each element of a list and replace the commas in a list with the (associative) operator `ff`, respectively:-

$$map\ ff\ [x_1, x_2, ..., x_n] \ = \ [ff(x_1), ff(x_2), ..., ff(x_n)] \tag{1}$$

$$reduce\ \oplus\ [x_1, x_2, ..., x_n] \ = \ x_1 \ \oplus \ x_2 \ \oplus \ ... \ \oplus \ x_n \tag{2}$$

where $\oplus$ is an infix operator equivalent to `ff`. These higher-order functions have many uses, for instance the functions $\sum_{i=0}^{n} f(i)$ and $\prod_{i=0}^{n} g(i)$ can be implemented as in Figure 2.

```
fun add x y = x + y : real
fun mul x y = x * y : real
fun iota' 0 = []
  | iota' n = n::(iota' (n - 1))
val iota = rev o iota'
fun sum f n = ((reduce add) o (map f)) (iota n)
fun prd g n = ((reduce mul) o (map g)) (iota n)
```

Figure 2: Combining two higher-order functions in SML

This example illustrates how different modules can be combined, using function composition, to produce other higher-order functions. In fact, the combination of `reduce` and `map` has important implications, notably in the application of list *homomorphisms* to the design of algorithms [Bir87].

1.2 Functional Languages and Parallel Programming

Another important advantage of functional languages is their relevance to the design of parallel programmes [Jon89]. In the design of a parallel system, there are a number of activities required in addition to those for a sequential system solving the same problem.

Firstly, the problem has to be decomposed in such a way that different parts of the problem can be solved concurrently. Some problems have *embarassing* amounts of intrinsic parallelism, others seem to be inherently sequential in nature although these can sometimes be transformed into problems in which significant degrees of parallelism can be exploited.

Once the problem has been decomposed, there has to be a mapping from concurrent operations to the physical hardware of the system. This may require additional considerations such as the tradeoff between replication of data on individual processing elements and the access complications of shared data.

Even when the mapping from the problem solution to the hardware has been deduced there are a number of low-level activities that have to be considered such as communication and synchronisation issues and the scheduling policy. These are important in the validation of the resulting software where phenomena such as *deadlock*, *livelock* and *fairness* all have to be taken into account in a proof of the correctness of the system.

Functional languages can be used to aid this process in a number of ways. Note, firstly that it is much easier to deduce implicit parallelism from functional languages than imperative languages.

Imperative languages often require programmer annotations to guide the compiler in generating parallel constructs. While this is also possible for functional languages [Hud86] a much neater approach is to make use of the implicit parallelism which arises from the property of referential transparency. The problem then becomes one of controlling this parallelism in the most effective way.

With referential transparency, data shared between concurrent tasks are far less likely to become involved in problems of conflict. In fact, if the parallel implementation is functional in nature then no explicit methods for resolving conflict are required at all. A corollary of this is that problems associated with the communication and synchronisation models such as deadlock are also much less likely to occur.

When validating a system or proving its correctness it is far easier to reason about a functional language, whether parallel or sequential, than about an explicitly parallel imperative system. Of course, in a full verification the compiler that converts the functional code into a parallel implementation should also be validated. This only need be carried out once, however, and the full power of the functional language such as algebraic transformations [Bac78] or induction proofs [MNV73] becomes applicable to parallel implementations. One exception to this, however, is Occam in which an imperative language has been restricted to an extent whereby formal programme transformation becomes a viable proposition [Bur88].

1.3 Algorithmic Skeletons

The above arguments indicate how functional languages might be advantageous in the development of parallel systems. Much work remains to be done, however, in the automatic conversion from a functional language to a parallel implementation. There have been several attempts to implement functional languages in parallel, for instance the GRIP (graph reduction in parallel) project [JCSH87, HJ92] which has dedicated hardware designed to exploit the graph reduction model. Another example is the ALICE system [DR81, CDF$^+$87] which uses the dataflow model [AC86] for functional languages.

These methods are entirely general and attempt to exploit the parallelism implicit in the evaluation (ie. the reduction to normal forms) of a functional language. The main problems with such implementations seem to be peripheral activities associated with evaluation and constraining the parallelism in such a way that the application maps naturally on to the chosen architecture. To implement a functional language on a von Neumann machine requires a great deal of bookkeeping, for instance garbage collectors which have to cope with a parallel architecture [Deb86] and other memory management issues. Similarly, constraining parallelism requires advanced task manipulation methods with issues such as efficient task creation and deletion and the minimisation of redundant tasks, see [Jon89] for a general overview.

A more recent approach is that of algorithmic skeletons [Col89]. Instead of trying to use the full generality of mapping an arbitrary functional language implementation onto parallel hardware, the programmer is restricted to a small number of programming constructs for which there are functional language implementations and efficient parallel constructs realisable on the chosen architecture [Bus93, DFH+93]. This approach has been studied at this department and has been implemented in the form of an ML compiler [Bra93] and is a subject of considerable interest elsewhere [DMPV91, Pel93, DFH+93].

The idea behind this approach is that the skeletons act like higher-order functions for parallel implementations. The advantages of higher-order functions in functional languages are well documented [Bac78, Hug89] but providing such a facility for an explicitly parallel imperative language is not easy, for instance Occam has no facilities at all for code or data abstraction. A functional language is therefore a suitable vehicle for providing such abstractions given that a predictable and robust mapping from functional language HOFs to parallel implementations can be provided. This has turned out to be a practical proposition [Bra91] and, moreover, has been used in practical applications [WMM90, WMM+92, Kou93].

There is a compromise, however, between level of abstraction and performance of the final implementation. Programmes at a low level of abstraction, say assembly language programming, tend to be fast but difficult to maintain. Programmes written at a high level of abstraction tend to be easier to manage but are usually inefficient. One of the reasons for this is that by creating an abstraction some of the information needed to perform a global optimisation becomes *hidden* [BHSI87].

In the case of algorithmic skeletons this can be ameliorated by using accurate performance models for the skeletons in combination with a set of transformation rules for the functional language. The efficiency of the final implementation is therefore *possibly* improved by selecting the best skeleton, or set of skeletons, from a library of such skeletons with performance models for typical data sets. Other methods of improving the efficiency of abstractions have been tried such as multi-level data structures [BHSI87], but these are more difficult to implement: selecting the best skeleton is a method that follows naturally from the use of skeletons in the first place.

As a simple example of a skeleton, consider the following (fictitous) fragment of ML code:-

```
val HoughSpace = foldl accumulatefn (map transformfn (split xsize ysize dx dy))
```

This creates a histogram space from an image space by dividing the image up into equal sized rectangular portions (`split`) transforming each image portion into a list of transformed co-ordinates (`transformfn`) and then combines the results into a histogram by accumulating duplicate co-ordinates (`accumulatefn`). In terms of a functional programme this contains certain inefficiencies such as the unnecessary `split` operation but if the `map` function is implemented as a parallel construct this allows direct user control over the granularity of the parallel implementation.

The `map` function is one of the most common higher-order functions and has parallel implementations on virtually any parallel hardware. On MIMD machines it could be implemented as a processor farm on message passing systems [Bra93] or, as a BSP (Broadcasting Sequential Processes) algorithm [Geh84], on custom hardware [BHSI87]. Other implementations are presumably possible on shared memory systems and SIMD machines. This gives the skeletons approach relevance in the search for architecturally independent parallel programming [Ski90].

Although the `map` function is extremely simple, much more complex skeletons are possible such as divide and conquer [dGHM93], task queues [Col89], reduction operators [Bra94], reduce and map pairs [DFH+93]. An algorithmic skeleton can be as complicated as required but if a skeleton becomes very complex then many of the traditional problems associated with parallel programming become applicable to the construction of the skeleton.

Backus pointed out in his much cited paper: "(there are some) advantages to be drawn from the severely restricted FP systems, as compared with the much more powerful classical systems" [Bac78]. The crux of his argument is that the simpler system can be used directly to formally prove properties of programmes by personnel who do not have an understanding of advanced mathematical techniques such as properties of complete partially ordered sets or least fixed points of functionals. Backus was comparing his FP language (somewhat conceptually similar to the skeleton approach) with classical λ-calculus but his comments could equally apply to the relationship between the skeleton function approach and, say, the parallel graph reduction approach.

1.4 Prototyping Methods

Finally, note that the traditional role of prototyping in a software environment is more to do with clearing up misunderstandings between system developers and end-users [Som92]. A prototype can be used to prove a specification in terms of correctness or completeness and has incidental advantages in terms of training users or running back-to-back tests against the final implementation.

The use of prototypes as exemplified by [WMM90] is rather specialised, whereby the idea is to select the correct parallel implementation from a number of (possibly quite different) functional prototypes. What makes this worthwhile is the difficulty of directly proving and modifying parallel code. Additional benefits will acrue in future when a more complete library is available which should allow a better choice of skeletons to be compared with one another. Note also that, at present, the tools for automatic selection of skeletons and programme transformation are still at an early stage so manual prototyping is currently used to achieve the same effect. As the tools improve, comparisons between hand coded prototypes and parallel implementations with these systems should be worthwhile in terms of evaluating both the prototyping methodology and the skeletonising compilers.

2 Prototyping the Hough Transform

It was decided to use prototyping methodologies to implement the Hough transform to complete a logical set of existing image processing modules. There are currently low-level processing (Canny edge detector) and high-level processing module implementations (perspective inversion model matching). The Hough transform, along with other segmentation algorithms, is usually described as an intermediate-level vision task. Different problems are encountered in implementations at different levels of processing in vision systems and it is important to study all of these if general methods are to be developed for such systems.

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 [Hou62] but the basic theory dates back much further to a method known as the Radon transform [Dea81]. This technique has widespread uses in vision related tasks, for instance, curve detection in intensity [DH72] and depth images [Jon91, NZ88].

An example of one of these tasks is locating 2D lines in an intensity image [DL78]. In this case the system of equations is the normal form for the equation of a straight line:-

$$\rho \;=\; x\;\cos\theta \;+\; y\;\sin\theta \tag{3}$$

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 and 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) space using a suitable range of ρ and θ values and using a voting system to determine the most likely values of ρ and θ, see Figure 3.

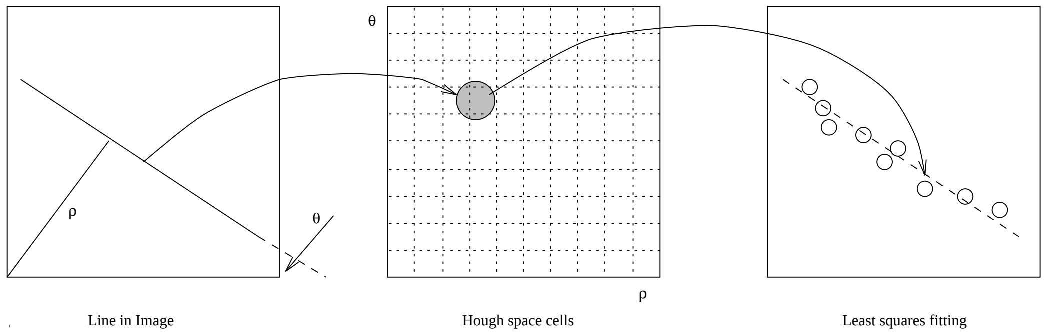

Figure 3: Schematic of the Hough Transform Process

Formulated in this way, the Hough transform technique is a partially robust method with good resistance to noise and occlusion [IK88]. 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 [Bal81].

There are, however, a range of problems associated with the technique. The main limitation is the amount of memory and calculation required. For a system of equations with n variables, each quantised to l levels, the size of the accumulator array is l^n values. For instance, for an ellipse with 5 parameters, if each parameter is accumulated to 200 levels, each of which is a four byte real number, the amount of memory required is 1250 MBytes. A number of methods of overcoming this problem have been developed, for instance the Adaptive Hough Transform [IK87].

Another major problem is finding the clusters (or peaks when accumulators are used) in Hough space. Quite often, in Hough space the peaks themselves have a definite structure, for instance the *four-leaf clover* patterns produced by an ellipse detector [Dav90] and the *butterfly* shapes produced by some line detectors [LB87]. Special filters can be developed which facilitate the efficient detection of peaks if their structure is known. Exploiting such structure with matched filters seems to be quite rare, however.

It is also possible for optical effects such as shadows to result in multiple peaks where a single object is more desirable. Neighbouring peaks tend to distort one another and are difficult to separate. Relatively sophisticated statistical methods are required to ensure robust detection of peaks in Hough space, rejection of false peaks and separation of neighbouring peaks, for example [Dav92].

Another issue is the quantisation of Hough space. Coarsely quantised Hough space results in poor accuracy of the resulting features and a higher chance of neighbouring peaks being taken as a single peak. Finely quantised Hough space gives greater accuracy in feature determination but exacerbates the problem of quantisation errors [VG81]. Angular data as produced by some types of edge detectors can be used to reduce the discretisation errors.

2.2 Parallel Implementations of the Hough Transform

There have been numerous attempts to implement the Hough transform in parallel [LDM+89, YA85, AWF91, LYT93]. The Hough transform represents a tricky problem for parallel systems since the accumulation stage implies some kind of global communications. Broadly speaking, parallel Hough transforms can be classified into three categories; input partitioning, output partitioning and mixed input/output partitioning [Lea93].

Input partitioning requires the starting image to be divided up between the processing elements. Each element then transforms its portion and transmits the result 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 of 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 [*ibid;*].

Note that for both input and output partitioning the transform operation can be combined with an edge detector or local window operation which is likely to precede the Hough transform [LDM+89].

Mixed input/output partitioning requires a relatively sophisticated communications model but has been implemented in practice on shared memory architectures [Lea93].

Aside from these considerations, each method can be control-driven or demand-driven. In the control-driven system, the image is sent to the processing elements which communicate among themselves, returning the result back to a central element. The demand-driven system requires a central processor to coordinate the work processors by supplying new tasks as workers become free This is known as processor farming.

The demand-driven method has been shown to be superior to the control-driven method [LDM+89] under most circumstances due to the implicit dynamic load balancing. Additionally, if the communication time is small with respect to the processing time then a linear pipeline of processors (which is the simplest topology to implement) performs almost as well as more complex arrangements such as n-ary trees.

2.3 The Hough Implementation

Due to time constraints it was only possible to implement a straight line Hough. One advantage of this is that the angular information available from the edge detector can be used directly in Eqn. 3 to give a one to one mapping which simplifies the accumulation strategy.

Taking into account the comments above, the only strategy considered was demand-driven control of an output partitioned algorithm. The design effort was then be focussed on the design of an efficient processor farm whereby the image is pre-loaded onto worker processors which accumulate and peak detect rectangular areas of Hough space. Line definitions are then reported to the farmer processor which performs a merge operation on contiguous or coincidental lines.

The accumulation and peak detection stages are all standard operations but the end points of the lines are detected by a simplex process of least-squares fitting followed by mapping of the accumulated points onto the fitted line. There is a single global threshold for minimum line length and minimal distance between neighbouring segments on the fitted line. It is actually the end points that are communicated to the worker processor.

Additionally, there is a primitive outlier detection implemented such that points more than three standard deviations from the fitted line are rejected. This depends on a good initial least squares fit, however, and can only cope with small percentages of outliers. This is a reasonable assumption since the points included in a given Hough cell should be approximately co-linear to begin with: this method alleviates problems of quantisation rather than rejecting wildly inaccurate points.

The global merge process is also implemented with global thresholds. Lines whose minimum euclidean distance are within the threshold of each other and whose angular discrepancy is within an angular threshold are merged. The merge process is carried out by combining the list of points for the two lines and updating the least squares fit. This means communicating lists of points from workers to the farmer but this is not a problem for sparse images.

Table 1 gives the full list of parameters for the Hough transform and Figure 4 is a block diagram of the implementation.

Hough Transform Parameters		
Name	Type	Description
hips	HIPS	Input edge maps, strength and orientation, also image size
low	int	Input threshold, pixels with strength ¡ low are ignored
rhoinc	int	Quantisation of the ρ dimension of Hough space
angleinc	int	Quantisation of the θ dimension of Hough space
hpx	int	Number of Hough space portions in x direction
hpy	int	Number of Hough space portions in y direction
hpo	int	Percentage of overlap between Hough portions
peakth	int	The threshold for the peak detector in Hough space
clustdist	real	The minimum distance between segments on a detected line
mergedist	real	Euclidean distance threshold for line merge
mergeangle	real	Angular threshold for line merging
minlinelen	real	Shortest acceptable line segment

Table 1: The Parameters for the Hough Transform Implementation

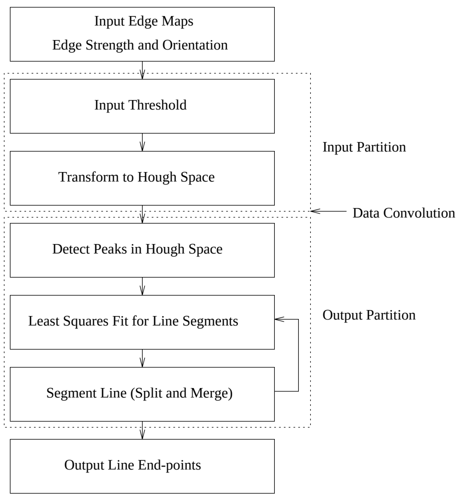

Figure 4: Block Diagram of the Hough Implementation

2.3.1 The SML Implementation

Since the overall strategy has been decided in advance the main effort was focussed on the correct implementation of the processor farm. The SML programme was structured in such a way that the communication and processing parameters could be deduced to within a reasonable approximation.

There are three identifiable operations in the Hough implementation 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 and line segmentation but these were considered as a single entity. There was not, in general, a sufficient number of peaks in Hough space or of segments on detected lines to consider implementing these operations in parallel.

Figure 5 shows the top level code for the SML implementation. The image is supplied in HIPS 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`. 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 from uncurried functions by partial application. Note that in terms of modelling a parallel system this process corresponds to predistribution of the programme parameters. The parallel component of the system is modelled by composing the three components (`tfn`, `acfn` and `ppfn`) and mapping over the Hough space portion definitions (`hough_portions`). Finally, the global merge process is carried out by flattening the result list (equivalent to the workers communicating their results to the farmer processor in the modelling analogy) and applying the `merge_lines` function.

2.3.2 Prototyping Information

The two main possibilities for the parallel architecture are a farm of pipelines which corresonds 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 [Bus90]:-

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

```
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
```

Figure 5: Top level SML Hough Implementation

thus:-

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

In terms of an SML programme, these two constructs are identical. To decide which 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 that bracket the type of data likely to be found in practice 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.

The set of test images of dimensions 256 by 256 pixels (which is the maximum image size that can be handled by the SML programme) comprised:-

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 and several circular holes (Figure 8). This represents the best quality image the system would be presented with in practice.

3. A real (photographic) image of the object in Figure 8. This image exhibits extremely poor contrast and has background features that coincide with edges on the object (Figure 9). This is about the poorest quality image of a simple scene the system is likely to be presented with.

4. A much more complex image with curved surfaces, specular reflections and marked surfaces (Figure 10).

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 segments 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 programme 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 raw results are presented in Tables 5, 6, 7 and 8 and are summarised in Table 9. The main point about these results is the dominance of the transformation stage over the others, typically about 30:5:1 for the ratio of 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) would indicate that this relationship will not always hold.

The static data flow analysis is complicated by the data dependency of the algorithms. Assuming sensible 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 the following estimates for the communications load:-

Communications loads (typical values)			
Phase	Initialisation Bytes	Input bytes	Output bytes
Transformation	512k	256	640k
Accumulation	4	640k	12k
Post-processing	32	12k	480

Table 2: Static Type Analysis

Briefly, there is not enough computation in sparse images for the accumulation and post-processing phases to warrant separate farms, although the communications overhead is moderate enough to consider either implementation. Bearing in mind other considerations such a the difficulty of implementing multiple farms as opposed to a single farm, the best candidate from this information is a farm of pipelines.

2.3.3 The Occam2 Implementation

The work on the SML prototype suggests that the best implementation for the parallel system is a farm of pipelines. The input image should be pre-distributed to the workers along with the programme 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 generated directly from the SML code. Large sections of SML code were converted into Occam2 by hand giving a close correspondence between the two programmes. The fact that SML code can be converted into Occam2 has already been establised [Bus91] but in this instance the conversion process was used to try and ensure that the two implementations have as similar characteristics as possible to aid the validity of the prototyping method.

The single processor farm was implemented using a simple scheduling strategy, without buffering, whereby the transputer channel **in** characteristics are used to create a queue of tasks down the pipeline (this actually requires an emulated **out** guard [Bur88]).

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:-

$$Speedup = \frac{sequential\ run\ time}{parallel\ run\ time} \tag{4}$$

$$Efficiency = \frac{speedup}{number\ of\ workers} \tag{5}$$

The results of these tests are presented in Tables 13 and 14. The same experiment was 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. What is really needed is a more sophisticated load distribution that takes 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. The assumption is made in the parallel implementation that this phase is much less computationally intensive than the parallel components but from Table 5 it is apparent that this assumption can break down when the global merge is given significantly more data.

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 programme output (ie. lines detected). Given the advantages of speedup and efficiency that are gained this is a worthwhile consideration.

Note, however, that this was not detected by the SML prototype. 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; a more sophisticated statistical model might conceivably be able to detect and compensate for effects such as the one presented here. Note that the high degree of data dependency means that it is always going to be difficult to tune the system to be efficient for a wide range of images.

3 Combining Several Prototypes into a Complete System

3.1 The Existing Software Components

The three existing components are the Canny edge detector, the Hough line segmentation process already described and the perspective inversion (PIA) model matching programme. 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 Figure 6.

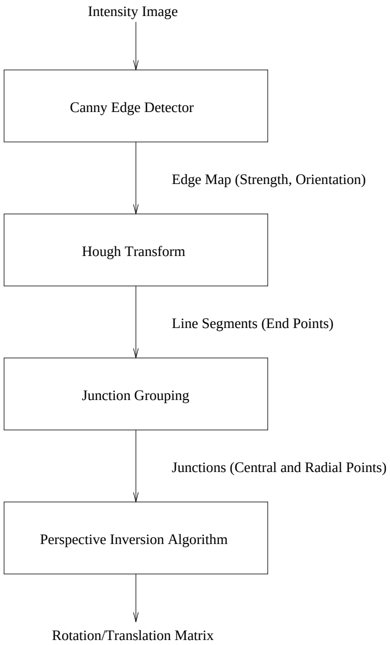

Figure 6: Block Diagram of the Complete System

The Canny edge detector was studied in comparison with the more primitive Laplacian and Sobel edge detectors [Kou93]. The Canny method [Can86] has almost become an industry standard, however, since it gives the best results in the presence of noise and has other desirable properties for later processing of images.

The convolution operations that are the core of all the edge detectors studied were themselves subjected to rigorous analysis. These involve 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.

In a similar situation to the Hough implementation in this document there are several phases; smoothing, edge detection and non-maximal suppression. There is a complication in this instance, however, 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 this work 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 studied in detail on sequential systems [McA90] but was subsequently implemented in a simpler form as an SML prototype and Occam2 implementation [WMM90].

The SML prototype identified four major components suitable for implementation on processor farms; generation of point triples (`generate.triples`), calculation of the rotation/translation matrix (`RT.calc`), calculating a score for the match defined by a given transformation matrix (`score`) and comparing the scores for all the generated triples to get the best match (`data.capture`).

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.

3.2 Combining the Three Components

Taking different components that have been developed using the SML prototyping and Occam2 implementation method and merging them has not previously been attempted. There are a number of questions raised by this idea that could be addressed by building a complete system and analysing its performance relative to the individual components.

Ultimately there should be a higher level abstraction mechanism provided by a skeletonising compiler to allow modularisation of code. This would then place additional demands upon the compiler implementation to ensure that the overall efficiency is maintained when modules are combined. This causes problems in that performing a global optimisation may require breaking open the abstraction boundary to gain the necessary information.

Unfortunately, for the implementation at hand the difficulty of modifying parallel code did not allow for this type of analysis. There are a number of parameters, however, that entertain a limited amount of optimisation and can be easily modified.

For instance, consider two programme modules implemented as processor farms passing data from one to the other. A global optimisation would require dataflow analysis from each of the two pipelines to make the decision as to whether to implement the farms as a single farm of pipelines or as a pipeline of separate farms. This requires type analysis from within the abstraction boundary, ie. the representation type becomes exported which is a violation of data integrity. A simpler approach is to treat the abstractions as *black boxes*, assuming a pipeline of farms and optimising the performance by re-allocating processing elements.

One other approach is to assume that the data transferred between the modules exactly reflects the behaviour of the two farms' communications with one another. This is a rather big assumption as, quite often (as in the case for the Hough implementation in Chapter 2), there are global processes situated between the output from one farm and the input to the other.

The modularisation of parallel code is highly desirable since code maintenance and reuse is one of the strengths of modularisation and is also one of the great weaknesses of parallel systems at the present time. The value of abstraction as a mechanism in software development has been recognised for a long time [Gut77, GH78] but due to the fact that parallel code has to be targetted on a wide range of different architectures it has proved much more difficult to provide a satisfactory abstraction mechanism for parallel software development.

Once such a mechanism is available, however, all the effort that has gone into making sequential programming systems reliable and easy to construct and maintain becomes applicable to equivalent parallel implementations. This is the motivation for methods such as the algorithmic skeletons approach [Col89] and the current activity in the search for architectural independence in parallel programming [Ski90] which is effectively an abstraction over the underlying hardware.

The biggest problem with combining these particular modules is that they were written completely independently without benefit of any structured programming methodology. This is particularly evident in the farm implementations where, despite that fact that all three are implemented on the same farm configuration, each farm's code differs radically in its methods of scheduling and control.

The other major restriction is that to modify the code at such a low level (for instance merging the three farms into a single pipeline) would require considerable programmer effort and might defeat any advantages gained by having the existing code in the first place. This means that the only realistic architecture is to create a pipeline of processors, each of which is a farmer for one of the component modules. This rather limits the performance since at any time during the computation only one farm is actually doing any work. The performance would be much better in system where images were being cascaded through the pipeline but this is a slightly different optimisation problem and would require extra control within the pipeline to ensure optimisation.

A spin-off from this is that the only means available for optimisation is processor placement. A simple procedure might be to perform an analysis of the individual farms to give a reasonably optimised system and then refine this allocation of processors by incremental adjustment.

Another point in combining these programmes is that there is actually a missing component see Figure 6. The Hough programme outputs end point specified line segments whereas the PIA programme 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 [McA90] which is designed to give less weight to co-linear lines. The conversion to point triples has already been written as part of the perspective inversion programme [WMM90] to allow input from model description files.

It is interesting to note that this missing phase could be spotted by considering the type information from the programme modules. The output type from the Hough segmentation algorithm did not match the input type of the perspective inversion programme. 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 programme under the highly contrived acronym of "(C)anny, (H)ough, (I)ntensity, (P)ia (S)ystem". This process is not of any great utility in terms of the parallel version of the complete system, however. The architecture was predefined by coding restrictions engendered by the limited time available. The information needed to perform the (limited) optimisation intended for the final parallel version was already available from the individual analyses of the discrete components. The combined SML code would be of much greater value in an environment where such code can be directly converted into parallel implementations.

The one useful point about the SML system was that the new code for the junction grouping was prototyped which aided the construction of the same code for the Occam2 version. This module was not implemented in parallel, however, since there was not enough computation involved compared with the other modules in the system.

The Occam2 code took considerably longer to combine but was aided by the previous work carried out during the original prototyping of the individual components. The architecture of the Occam2 code is presented in Figure 7. Each of the major components implemented in parallel are serviced by a single, dedicated farmer processor. The farmers each control a linear pipeline of processors (or two pipelines for the PIA farmer) and are themselves connected in a short 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 data from disc files (the raw image and the model description), writing the final results to a disc file and transmitting any initialisation data to the farmers.

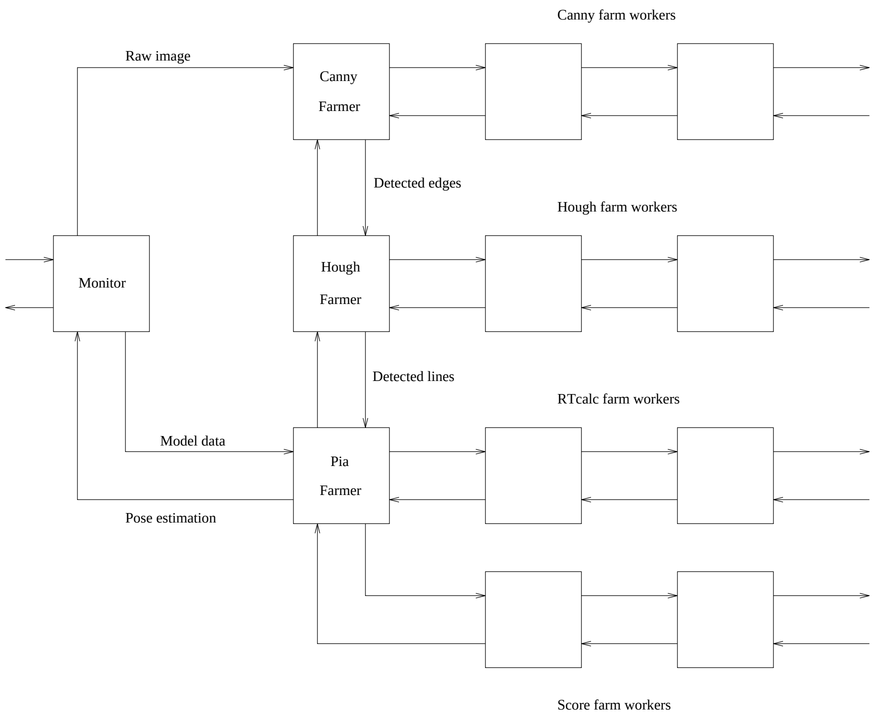

Figure 7: The architecture of the vision recognition system

The main effort required 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 but the intermediate values and timing data were also required. This data was sent back to a single monitor processor that handled the input and output files and collected the intermediate data for debug purposes. Luckily, the highest connectivity required (the PIA farmer) was not greater than the number of channels on a transputer.

3.3 Performance of the Combined System

The speedup and efficiency figures for the three individual farms are presented in Figures 11, 12, 13, 14, 15 and 16. The peaks for speedup and efficiency for each three farms (deduced directly from the graphs) are presented in Table 3 along with the measured sequential processing time for each phase.

	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	1608

Table 3: Optimum speedup and efficiency for individual farms

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 for farm workers this gives 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 [WMM90] as the most efficient balance for this farm.

Canny	Hough	Score	RTcalc	Speedup	Efficiency
2	4	3	17	10.54	0.35
3	3	3	17	9.81	0.33
3	4	2	17	9.42	0.31
3	4	3	16	9.99	0.33
1	5	3	17	9.20	0.31
2	5	2	17	9.94	0.33
2	5	3	16	9.83	0.33
1	4	4	17	9.29	0.31
2	3	4	17	9.65	0.32
2	4	4	16	10.04	0.33
1	4	3	18	9.16	0.30
2	3	3	18	9.51	0.32
2	4	2	18	9.33	0.31

Table 4: Incremental improvement in speedup by processor reallocation

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.

One point to note is that these results were all carried out on the **phwidg** 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. All three predictions, however, are close enough to the optimum to allow quick convergence in the simple heuristic search process applied here (in fact the starting point was the maximum speedup ratio).

More generally, a speedup of 10.5 for a system with 30 processing elements (26 farm workers, 3 farmers and 1 monitor processor) is a very poor figure but the reasons for this have already been pointed out. Of more interest is the fact that the individual farms are operating sub-optimally in the optimal configuration. This is most 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.

4 Conclusions

The work presented in this report has reaffirmed the potential of SML prototypes in the context of vision systems. While the resulting implementations are rather simple and would not be considered reliable enough for a practical vision system the methods used can be extrapolated to more realistic code.

Two main points arose from the Hough transform implementation, a notoriously difficult algorithm to implement efficiently in parallel. Firstly, it was shown that correct load balancing in the parallel implementation was of vital importance, not just for the efficiency of a particular application but also to make the system predictable enough for automatic optimisation. A point of caution was noted since it was discovered that the SML prototype can sometimes *mask* the real situation. This led to the conclusion that sophisticated statistical models may be required for atypical data sets.

The second point is that the SML prototype has to be *very* close in structure to the final implementation for the prototyping process to be valid. This point is not apparent in the foregoing work but during the development of the SML prototype an early version had substantially different characteristics from the proposed parallel implementation. This appears to defeat the point of working in an abstract language, divorced from the nuts and bolts of the intended hardware but the methodology here does not consider the use of the programme transformation methods that are available for languages such as SML. An optimising compiler could in theory transform a declarative problem statement into a solution with intrinsic parallelism [Bir87].

The construction of the complete system has shown that even simple analysis of the individual components can be used to predict an approximately optimal system and that the initial solution can be incrementally improved.

The restrictions of hand coding have prevented a more thorough optimisation of the system and indicate that automated tools are actually *necessary* to make available more complex decisions such as selecting different architectures in the parallel implementation. In the absence of such tools only a very limited set of parameters are available for optimisation but even these can be used to make worthwhile improvements to the system.

It is not certain whether architectural changes are likely to give better improvements in performance than correct tuning of a given hardware topology. It is suspected, however, that the key issue in this question is whether communication or computation is the dominant force. In an application where communication dominates it is likely that the correct physical placement of tasks will be more important than accurate processor allocations. If computation dominates then any physical interconnection of processors will approach optimality but correct allocation of the available processors would be of greater significance.

5 Future Work

This system is a preliminary to a much larger system [AS94] which will use both depth and intensity images along with fused images. The new system will be both more advanced (using robust statistical methods) and better controlled (in that processing resources and processing paths can be altered during the course of computation) than the simple system presented here.

An automated tool for converting SML code into parallel implementations is under development [Bra93] and a direct comparison is planned between the hand coded Occam2 implementation developed here and an automated conversion from the SML prototype for the CHIPS system into parallel code.

Investigations are also under way into the role abstract data types (ADTs) might play in the prototyping methodology used in this work. This is looking forward to the day when commercially viable implementations of skeletonising compilers are available and methods of controlling larger quantities of parallel code are required. The combining of three disparate sections of code is a preliminary to this.

Acknowledgements

I would like to thank Greg Michaelson and Andy Wallace for their supervision.

I would also like to thank Bill Austin and Tore Bratvold for their help with the Occam2 code and Jim Clark for the inspiration for the acronym CHIPS.

The work was funded by SERC grant GR/J07884.

A Test images

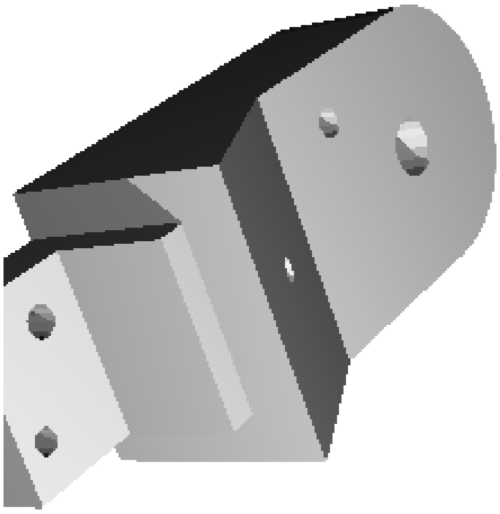

Figure 8: Widget: Single object, synthetic

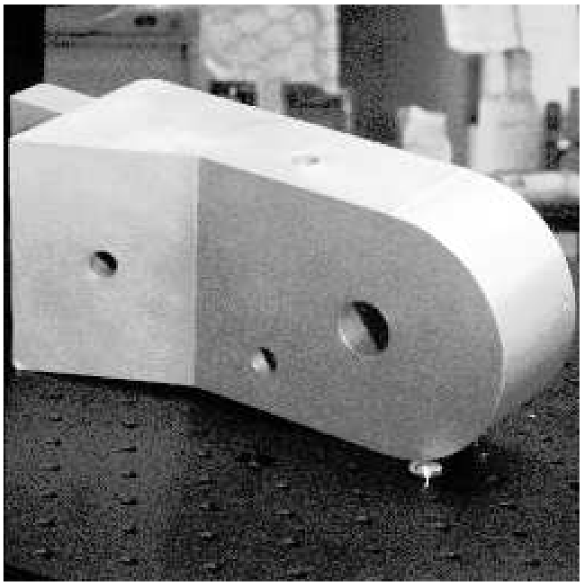

Figure 9: Phwidg: Single object, photographic

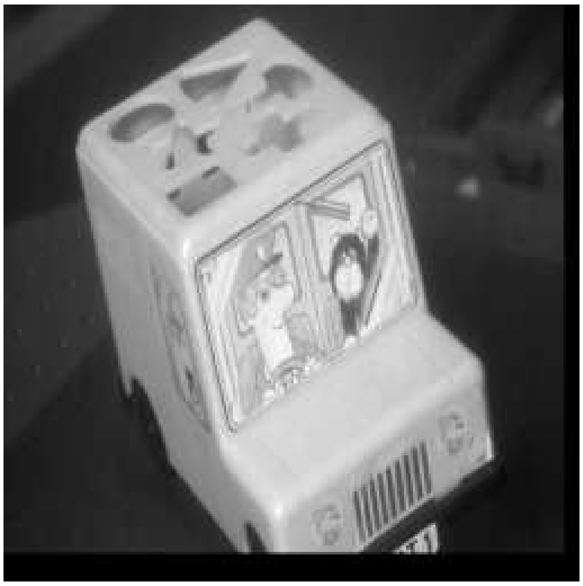

Figure 10: Postvan: Complex, photographic

B SML prototype information

Image	xp	yp	ovr	low	ρ-inc	θ-inc	peakth	cd	md	ma	mll	tfn	acfn	ppfn	merge
random.canny	4	4	10	0	64	64	1	10.0	5.0	0.3	15.0	89.9	134.17	19.85	321.35
random.canny	4	4	10	64	64	64	1	10.0	5.0	0.3	15.0	46.63	131.55	10.09	340.68
random.canny	4	4	10	128	64	64	1	10.0	5.0	0.3	15.0	43.8	125.41	10.43	344.74
random.canny	4	4	10	250	64	64	1	10.0	5.0	0.3	15.0	42.34	117.27	8.2	280.05
random.canny	4	4	10	128	8	8	1	10.0	5.0	0.3	15.0	42.91	4.87	77.39	54.97
random.canny	4	4	10	128	16	16	1	10.0	5.0	0.3	15.0	42.92	12.43	85.28	239.84
random.canny	4	4	10	128	32	32	1	10.0	5.0	0.3	15.0	42.48	46.94	31.28	537.69
random.canny	4	4	10	128	64	64	1	10.0	5.0	0.3	15.0	44.85	131.2	10.67	345.86
random.canny	4	4	10	128	128	128	1	10.0	5.0	0.3	15.0	43.8	173.75	3.69	4.44
random.canny	2	2	10	128	64	64	1	10.0	5.0	0.3	15.0	11.52	127.17	9.33	258.49
random.canny	4	4	10	128	64	64	1	10.0	5.0	0.3	15.0	43.78	134.92	10.48	352.86
random.canny	8	8	10	128	64	64	1	10.0	5.0	0.3	15.0	184.25	55.83	13.46	386.71
random.canny	16	16	10	128	64	64	1	10.0	5.0	0.3	15.0	715.66	22.73	13.67	393.37
random.canny	4	4	0	128	64	64	1	10.0	5.0	0.3	15.0	45.7	125.11	8.18	261.38
random.canny	4	4	10	128	64	64	1	10.0	5.0	0.3	15.0	44.42	126.4	9.83	344.94
random.canny	4	4	20	128	64	64	1	10.0	5.0	0.3	15.0	44.02	129.59	12.93	328.19
random.canny	4	4	30	128	64	64	1	10.0	5.0	0.3	15.0	45.76	148.38	12.85	378.34
random.canny	4	4	40	128	64	64	1	10.0	5.0	0.3	15.0	44.5	161.28	14.74	544.09
random.canny	4	4	10	128	64	64	1	10.0	5.0	0.3	15.0	45.13	133.81	9.01	338.57
random.canny	4	4	10	128	64	64	1000	10.0	5.0	0.3	15.0	44.0	127.48	10.02	336.33
random.canny	4	4	10	128	64	64	2000	10.0	5.0	0.3	15.0	44.65	134.82	12.3	322.2
random.canny	4	4	10	128	64	64	3000	10.0	5.0	0.3	15.0	43.75	130.14	9.56	283.77
random.canny	4	4	10	128	64	64	1	1.0	5.0	0.3	15.0	45.4	134.33	10.91	0.01
random.canny	4	4	10	128	64	64	1	2.0	5.0	0.3	15.0	44.62	132.47	9.93	0.0
random.canny	4	4	10	128	64	64	1	5.0	5.0	0.3	15.0	44.15	124.82	12.64	5.17
random.canny	4	4	10	128	64	64	1	10.0	5.0	0.3	15.0	44.2	139.43	10.04	348.93
random.canny	4	4	10	128	64	64	1	10.0	1.0	0.3	15.0	44.39	123.7	11.93	804.39
random.canny	4	4	10	128	64	64	1	10.0	2.0	0.3	15.0	44.12	126.91	10.38	504.24
random.canny	4	4	10	128	64	64	1	10.0	5.0	0.3	15.0	43.33	139.14	10.87	351.07
random.canny	4	4	10	128	64	64	1	10.0	10.0	0.3	15.0	44.37	140.28	11.0	144.54
random.canny	4	4	10	128	64	64	1	10.0	5.0	0.01	15.0	44.9	139.26	11.18	753.78
random.canny	4	4	10	128	64	64	1	10.0	5.0	0.05	15.0	43.69	135.05	11.37	820.32
random.canny	4	4	10	128	64	64	1	10.0	5.0	0.1	15.0	42.69	137.25	11.12	658.66
random.canny	4	4	10	128	64	64	1	10.0	5.0	0.3	15.0	46.81	136.11	9.8	340.44

Table 5: Image: Synthetic, randomly generated

Image	xp	yp	ovr	low	ρ-inc	θ-inc	peakth	cd	md	ma	mll	tfn	acfn	ppfn	merge
widget.canny	4	4	10	0	64	64	1	10.0	5.0	0.3	15.0	83.12	13.79	2.25	11.81
widget.canny	4	4	10	64	64	64	1	10.0	5.0	0.3	15.0	19.01	1.03	0.31	1.36
widget.canny	4	4	10	128	64	64	1	10.0	5.0	0.3	15.0	18.21	0.77	0.32	1.33
widget.canny	4	4	10	250	64	64	1	10.0	5.0	0.3	15.0	18.11	0.77	0.27	1.02
widget.canny	4	4	10	128	8	8	1	10.0	5.0	0.3	15.0	17.84	0.17	1.26	0.66
widget.canny	4	4	10	128	16	16	1	10.0	5.0	0.3	15.0	18.52	0.33	0.81	0.8
widget.canny	4	4	10	128	32	32	1	10.0	5.0	0.3	15.0	18.34	0.58	0.36	0.97
widget.canny	4	4	10	128	64	64	1	10.0	5.0	0.3	15.0	18.04	0.85	0.31	1.38
widget.canny	4	4	10	128	128	128	1	10.0	5.0	0.3	15.0	18.15	0.45	0.32	1.62
widget.canny	2	2	10	128	64	64	1	10.0	5.0	0.3	15.0	4.16	0.85	0.3	1.11
widget.canny	4	4	10	128	64	64	1	10.0	5.0	0.3	15.0	17.94	0.88	0.29	1.28
widget.canny	8	8	10	128	64	64	1	10.0	5.0	0.3	15.0	75.43	0.44	0.39	1.63
widget.canny	16	16	10	128	64	64	1	10.0	5.0	0.3	15.0	295.08	0.25	0.35	1.4
widget.canny	4	4	0	128	64	64	1	10.0	5.0	0.3	15.0	17.93	0.89	0.26	1.09
widget.canny	4	4	10	128	64	64	1	10.0	5.0	0.3	15.0	18.23	0.84	0.32	1.43
widget.canny	4	4	20	128	64	64	1	10.0	5.0	0.3	15.0	18.08	0.82	0.32	1.46
widget.canny	4	4	30	128	64	64	1	10.0	5.0	0.3	15.0	17.96	0.92	0.41	2.15
widget.canny	4	4	40	128	64	64	1	10.0	5.0	0.3	15.0	18.98	1.7	0.59	3.36
widget.canny	4	4	10	128	64	64	1	10.0	5.0	0.3	15.0	17.67	0.72	0.32	1.29
widget.canny	4	4	10	128	64	64	1000	10.0	5.0	0.3	15.0	17.62	0.66	0.29	1.3
widget.canny	4	4	10	128	64	64	2000	10.0	5.0	0.3	15.0	17.79	0.76	0.28	0.96
widget.canny	4	4	10	128	64	64	3000	10.0	5.0	0.3	15.0	18.27	0.92	0.21	0.66
widget.canny	4	4	10	128	64	64	1	1.0	5.0	0.3	15.0	19.28	1.08	1.02	0.06
widget.canny	4	4	10	128	64	64	1	2.0	5.0	0.3	15.0	18.64	0.95	0.96	0.18
widget.canny	4	4	10	128	64	64	1	5.0	5.0	0.3	15.0	18.67	0.85	0.56	1.29
widget.canny	4	4	10	128	64	64	1	10.0	5.0	0.3	15.0	18.04	0.79	0.33	1.39
widget.canny	4	4	10	128	64	64	1	10.0	1.0	0.3	15.0	18.97	0.7	0.32	1.37
widget.canny	4	4	10	128	64	64	1	10.0	2.0	0.3	15.0	18.07	0.91	0.29	1.34
widget.canny	4	4	10	128	64	64	1	10.0	5.0	0.3	15.0	17.91	0.68	0.31	1.35
widget.canny	4	4	10	128	64	64	1	10.0	10.0	0.3	15.0	17.16	0.67	0.3	1.27
widget.canny	4	4	10	128	64	64	1	10.0	5.0	0.01	15.0	18.39	0.98	0.31	2.23
widget.canny	4	4	10	128	64	64	1	10.0	5.0	0.05	15.0	18.61	0.89	0.29	1.72
widget.canny	4	4	10	128	64	64	1	10.0	5.0	0.1	15.0	19.15	0.94	0.3	1.66
widget.canny	4	4	10	128	64	64	1	10.0	5.0	0.3	15.0	19.17	0.81	0.31	1.32

Table 6: Image: Synthetic, single object

Image	xp	yp	ovr	low	ρ-inc	θ-inc	peakth	cd	md	ma	mll	tfn	acfn	ppfn	merge
phwidg.canny	4	4	10	0	64	64	1	10.0	5.0	0.3	15.0	82.41	98.63	16.28	257.62
phwidg.canny	4	4	10	64	64	64	1	10.0	5.0	0.3	15.0	19.95	4.96	0.6	4.84
phwidg.canny	4	4	10	128	64	64	1	10.0	5.0	0.3	15.0	18.13	1.98	0.38	1.86
phwidg.canny	4	4	10	250	64	64	1	10.0	5.0	0.3	15.0	17.67	0.76	0.31	0.71
phwidg.canny	4	4	10	128	8	8	1	10.0	5.0	0.3	15.0	18.1	0.23	1.94	5.12
phwidg.canny	4	4	10	128	16	16	1	10.0	5.0	0.3	15.0	19.05	0.56	0.94	4.4
phwidg.canny	4	4	10	128	32	32	1	10.0	5.0	0.3	15.0	18.99	1.09	0.66	2.76
phwidg.canny	4	4	10	128	64	64	1	10.0	5.0	0.3	15.0	18.0	1.96	0.41	1.84
phwidg.canny	4	4	10	128	128	128	1	10.0	5.0	0.3	15.0	18.03	1.21	0.34	1.52
phwidg.canny	2	2	10	128	64	64	1	10.0	5.0	0.3	15.0	4.42	2.23	0.35	1.66
phwidg.canny	4	4	10	128	64	64	1	10.0	5.0	0.3	15.0	19.34	2.54	0.43	1.89
phwidg.canny	8	8	10	128	64	64	1	10.0	5.0	0.3	15.0	71.89	1.13	0.47	2.09
phwidg.canny	16	16	10	128	64	64	1	10.0	5.0	0.3	15.0	303.38	0.52	0.44	2.19
phwidg.canny	4	4	0	128	64	64	1	10.0	5.0	0.3	15.0	18.51	2.45	0.36	1.62
phwidg.canny	4	4	10	128	64	64	1	10.0	5.0	0.3	15.0	18.14	1.93	0.41	1.86
phwidg.canny	4	4	20	128	64	64	1	10.0	5.0	0.3	15.0	18.21	2.08	0.47	2.1
phwidg.canny	4	4	30	128	64	64	1	10.0	5.0	0.3	15.0	18.52	2.75	0.54	2.6
phwidg.canny	4	4	40	128	64	64	1	10.0	5.0	0.3	15.0	20.12	3.66	0.64	3.06
phwidg.canny	4	4	10	128	64	64	1	10.0	5.0	0.3	15.0	18.55	2.58	0.36	1.9
phwidg.canny	4	4	10	128	64	64	1000	10.0	5.0	0.3	15.0	18.67	3.0	0.35	1.37
phwidg.canny	4	4	10	128	64	64	2000	10.0	5.0	0.3	15.0	18.1	2.34	0.3	0.64
phwidg.canny	4	4	10	128	64	64	3000	10.0	5.0	0.3	15.0	18.29	2.3	0.22	0.28
phwidg.canny	4	4	10	128	64	64	1	1.0	5.0	0.3	15.0	18.02	1.96	0.85	1.31
phwidg.canny	4	4	10	128	64	64	1	2.0	5.0	0.3	15.0	17.81	1.93	0.85	1.45
phwidg.canny	4	4	10	128	64	64	1	5.0	5.0	0.3	15.0	19.41	2.27	0.47	1.86
phwidg.canny	4	4	10	128	64	64	1	10.0	5.0	0.3	15.0	17.78	1.99	0.38	1.85
phwidg.canny	4	4	10	128	64	64	1	10.0	1.0	0.3	15.0	18.08	1.86	0.44	2.47
phwidg.canny	4	4	10	128	64	64	1	10.0	2.0	0.3	15.0	19.35	2.17	0.41	2.07
phwidg.canny	4	4	10	128	64	64	1	10.0	5.0	0.3	15.0	18.45	1.93	0.37	1.95
phwidg.canny	4	4	10	128	64	64	1	10.0	10.0	0.3	15.0	17.81	2.38	0.37	1.74
phwidg.canny	4	4	10	128	64	64	1	10.0	5.0	0.01	15.0	18.51	2.7	0.43	4.11
phwidg.canny	4	4	10	128	64	64	1	10.0	5.0	0.05	15.0	17.66	1.91	0.39	2.93
phwidg.canny	4	4	10	128	64	64	1	10.0	5.0	0.1	15.0	18.05	1.94	0.46	3.03
phwidg.canny	4	4	10	128	64	64	1	10.0	5.0	0.3	15.0	19.03	2.6	0.37	1.86

Table 7: Image: Photographic, single object

Image	xp	yp	ovr	low	ρ-inc	θ-inc	peakth	cd	md	ma	mll	tfn	acfn	ppfn	merge
postvan.canny	4	4	10	0	64	64	1	10.0	5.0	0.3	15.0	80.33	68.14	13.55	145.07
postvan.canny	4	4	10	64	64	64	1	10.0	5.0	0.3	15.0	24.18	13.59	1.61	23.46
postvan.canny	4	4	10	128	64	64	1	10.0	5.0	0.3	15.0	21.66	8.59	1.22	13.33
postvan.canny	4	4	10	250	64	64	1	10.0	5.0	0.3	15.0	19.8	4.11	0.68	5.97
postvan.canny	4	4	10	128	8	8	1	10.0	5.0	0.3	15.0	21.85	0.47	14.75	9.99
postvan.canny	4	4	10	128	16	16	1	10.0	5.0	0.3	15.0	23.13	1.19	6.3	12.81
postvan.canny	4	4	10	128	32	32	1	10.0	5.0	0.3	15.0	21.38	3.42	2.07	13.11
postvan.canny	4	4	10	128	64	64	1	10.0	5.0	0.3	15.0	23.15	10.53	1.32	13.33
postvan.canny	4	4	10	128	128	128	1	10.0	5.0	0.3	15.0	21.73	7.89	0.93	9.69
postvan.canny	2	2	10	128	64	64	1	10.0	5.0	0.3	15.0	4.94	10.86	1.07	11.89
postvan.canny	4	4	10	128	64	64	1	10.0	5.0	0.3	15.0	20.01	9.46	1.29	13.55
postvan.canny	8	8	10	128	64	64	1	10.0	5.0	0.3	15.0	87.53	3.28	1.57	14.26
postvan.canny	16	16	10	128	64	64	1	10.0	5.0	0.3	15.0	372.5	2.49	1.76	17.57
postvan.canny	4	4	0	128	64	64	1	10.0	5.0	0.3	15.0	23.14	8.37	1.34	11.91
postvan.canny	4	4	10	128	64	64	1	10.0	5.0	0.3	15.0	21.8	10.94	1.24	13.33
postvan.canny	4	4	20	128	64	64	1	10.0	5.0	0.3	15.0	19.74	11.33	1.16	14.09
postvan.canny	4	4	30	128	64	64	1	10.0	5.0	0.3	15.0	20.33	12.85	1.91	20.88
postvan.canny	4	4	40	128	64	64	1	10.0	5.0	0.3	15.0	22.52	13.71	2.23	24.47
postvan.canny	4	4	10	128	64	64	1	10.0	5.0	0.3	15.0	19.86	10.96	1.27	13.25
postvan.canny	4	4	10	128	64	64	1000	10.0	5.0	0.3	15.0	21.48	11.18	1.2	12.75
postvan.canny	4	4	10	128	64	64	2000	10.0	5.0	0.3	15.0	22.13	8.6	1.03	5.78
postvan.canny	4	4	10	128	64	64	3000	10.0	5.0	0.3	15.0	22.32	13.21	0.9	3.26
postvan.canny	4	4	10	128	64	64	1	1.0	5.0	0.3	15.0	22.32	10.86	2.87	7.69
postvan.canny	4	4	10	128	64	64	1	2.0	5.0	0.3	15.0	21.52	10.52	2.71	11.63
postvan.canny	4	4	10	128	64	64	1	5.0	5.0	0.3	15.0	19.78	11.08	1.86	14.37
postvan.canny	4	4	10	128	64	64	1	10.0	5.0	0.3	15.0	19.55	11.07	1.28	13.22
postvan.canny	4	4	10	128	64	64	1	10.0	1.0	0.3	15.0	21.73	9.57	1.24	15.31
postvan.canny	4	4	10	128	64	64	1	10.0	2.0	0.3	15.0	23.63	10.46	1.22	13.38
postvan.canny	4	4	10	128	64	64	1	10.0	5.0	0.3	15.0	22.9	10.24	1.17	13.66
postvan.canny	4	4	10	128	64	64	1	10.0	10.0	0.3	15.0	22.65	9.54	1.26	10.7
postvan.canny	4	4	10	128	64	64	1	10.0	5.0	0.01	15.0	22.91	8.83	1.23	28.85
postvan.canny	4	4	10	128	64	64	1	10.0	5.0	0.05	15.0	22.2	8.51	1.27	24.28
postvan.canny	4	4	10	128	64	64	1	10.0	5.0	0.1	15.0	19.52	11.02	1.28	18.76
postvan.canny	4	4	10	128	64	64	1	10.0	5.0	0.3	15.0	21.91	8.74	1.28	13.6

Table 8: Image: Photographic, complex scene

Image	μ_{tfn}	σ_{tfn}	μ_{acfn}	σ_{acfn}	μ_{ppfn}	σ_{ppfn}	μ_{merge}	σ_{merge}
random	68.5	117.2	118.90	40.20	15.70	17.20	345.00	208.80
widget	29.6	49.2	1.16	2.24	0.46	0.39	1.62	1.89
phwidg	29.9	50.4	4.89	16.60	0.97	2.72	9.72	43.80
postvan	35.2	61.5	10.75	10.70	2.32	3.15	17.90	23.10

Table 9: SML timing summary (μ_x = mean for phase 'x', σ_x = standard deviation, values in seconds

C Occam2 performance measures

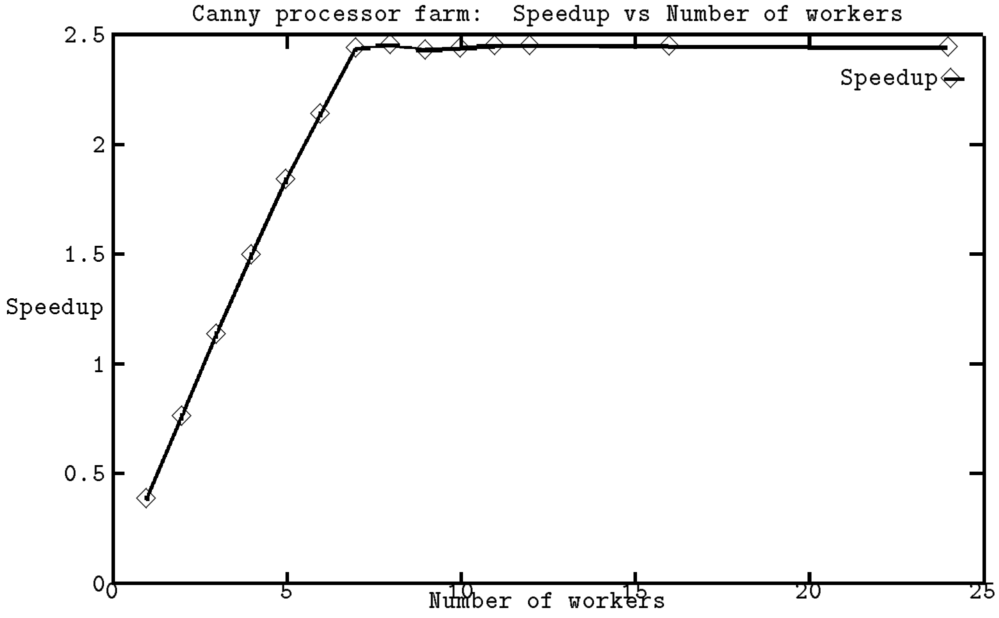

Figure 11: Canny farm speedup

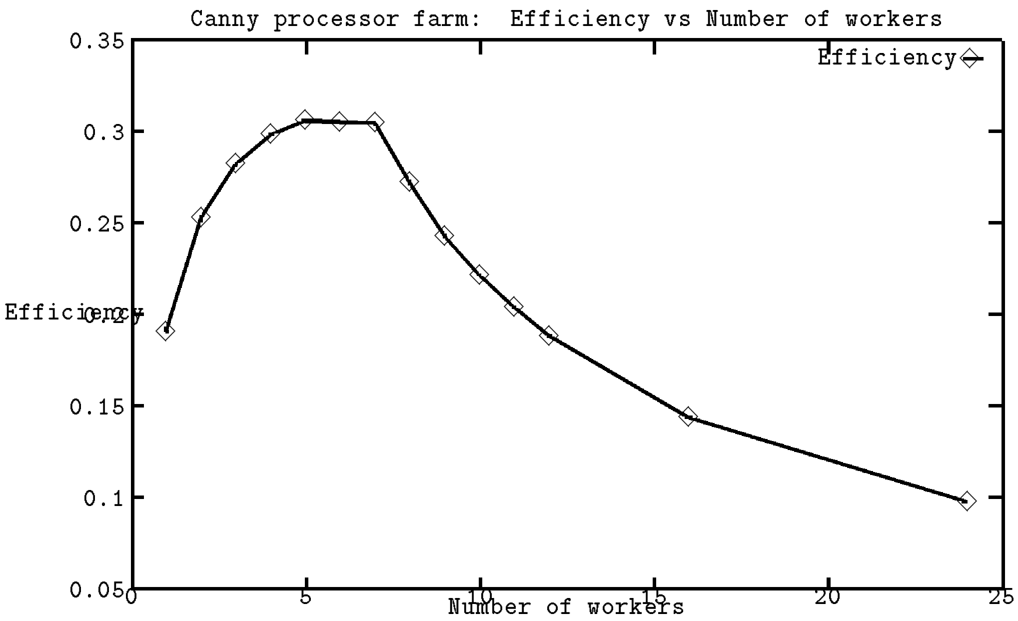

Figure 12: Canny farm efficiency

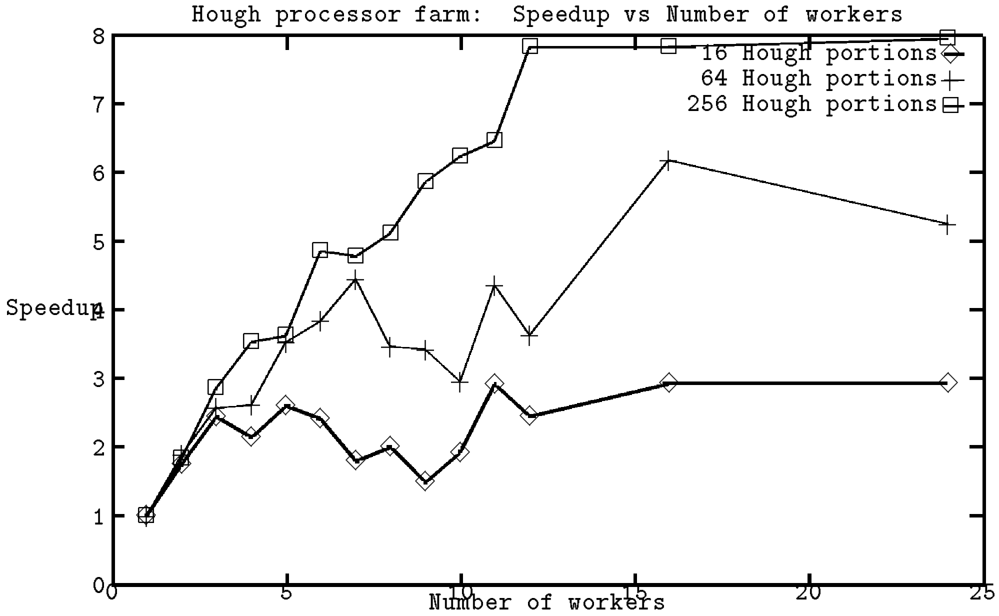

Figure 13: Hough farm speedup

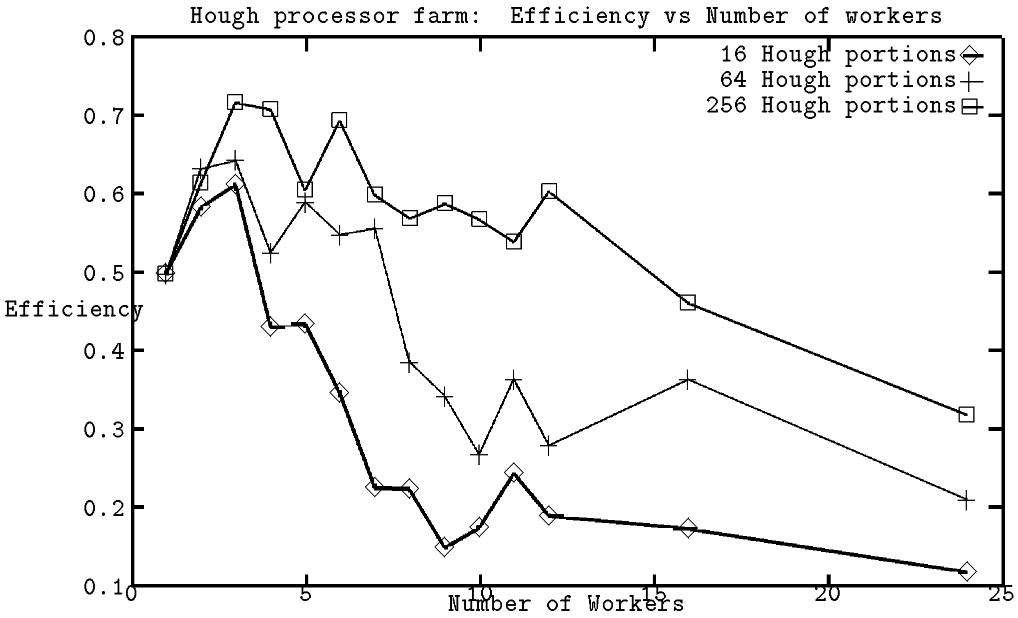

Figure 14: Hough farm efficiency

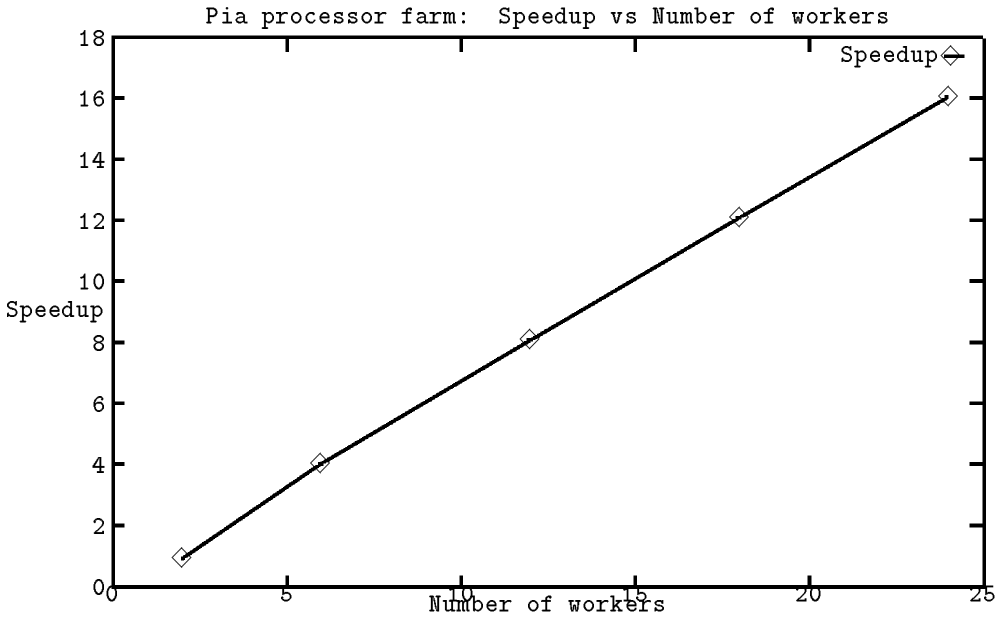

Figure 15: Pia farm speedup

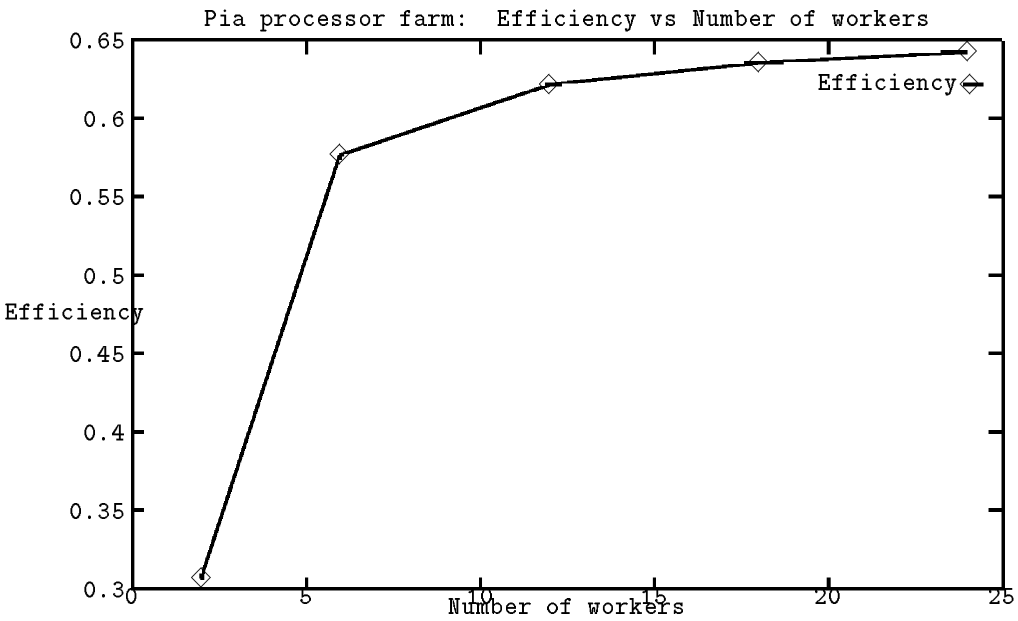

Figure 16: Pia farm efficiency

References

[AC86] Arvind and David E Culler. Dataflow Architectures. *Annual Reviews in Computer Science*, (1):225–253, 1986.

[AS94] W. J. Austin and N. R. Scaife. Reconfigurable Parallel Vision System: Informal Specification (Working Document). Technical Report RM/94/4, Dept. of Computing and Electrical Engineering, Heriot-Watt University, Apr 1994.

[AWF91] 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.

[Bac78] J. Backus. Can programming be liberated from the von Neumann Style? *CACM*, 21(8):613–641, 1978.

[Bal81] D. H. Ballard. Generalising the Hough Transform to Detect Arbitrary Shapes. *Pattern Recognition*, 13:111–122, 1981.

[BHSI87] F. Bastani, W. Hilal, and S. Sithrama Iyengar. Efficient Abstract Data Type Components for Distributed and Parallel Systems. *IEEE Computer*, pages 33–44, 1987.

[Bir87] Richard S Bird. An introduction to the theory of lists. In M Broy, editor, *Logic of Programming and Calculi of Discrete Design*, volume F36 of *NATO ASI Series*, pages 5–42. Springer-Verlag, 1987.

[Bra91] Tore A Bratvold. Functional Components for Parallel Programming. Master's thesis, Department of Computer Science, Heriot-Watt University, Edinburgh, September 1991.

[Bra92] Tore A Bratvold. Determining Useful Parallelism in Higher Order Functions. In *Proceedings of the 4th International Workshop on the Parallel Implementation of Functional Languages, Aachen, Germany*, pages 213–226, September 1992.

[Bra93] Tore A 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.

[Bra94] T. Bratvold. *Skeleton-based Parallelisation of Functional Programmes.* PhD thesis, Dept. of Computing and Electrical Engineering, Heriot-Watt University, 1994.

[Bur88] A. Burns. *Programming in Occam2.* Addison-Wesley, 1988.

[Bus90] D. Busvine. Transformation Techniques for Functional Programmes. Tech. Rep. 90/3, Dept. of Computing and Electrical Engineering, Heriot-Watt University, Riccarton, Edinburgh, Jul 1990.

[Bus91] D. Busvine. Translation of SML to Sequential Occam2. Technical Report TR91/7, Dept. of Computing and Electrical Engineering, Heriot-Watt University, Riccarton, Edinburgh, 1991.

[Bus93] David Busvine. Implementing Recursive Functions as Processor Farms. *Parallel Computing*, 19:1141–1153, 1993.

[Can86] J. Canny. A Computational Approach to Edge Detection. *IEEE Trans. Pattern Analysis and Machine Intelligence*, PAMI-8:679–698, 1986.

[CDF+87] 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.

[Col89] M. I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* Pitman/MIT, 1989.

[Dav90] E. R. Davies. *Machine Vision: Theory, Algorithms, Practicalities.* Academic Press, 1990.

[Dav92] E. R. Davies. Simple two-stage method for the accurate location of Hough transform peaks. *IEE Proc. Comput. and Digital Tech.*, 139(3):242–248, 1992.

[Dea81] S. R. Deans. Hough Transform from the Radon Transform. *IEEE Trans. Pattern Analysis and Machine Intelligence*, PAMI-3:185–188, 1981.

[Deb86] A. Deb. *Parallel Garbage Collection for Graph Machines*, pages 252–243. LNCS 279. Springer-Verlag, 1986.

[DFH+93] 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.

[dGHM93] I P de Guzmán, P G Harrison, and E Medina. Pipelines for Divide-and-Conquer Functions. *The Computer Journal*, 36(3):254–268, 1993.

[DH72] R. O. Duda and P. E. Hart. Use of the Hough Transform to Detect Line and Curves in Pictures. *CACM*, 15:11–15, 1972.

[DL78] S. A. Dudani and A. L. Luk. Locating straight-line edge segments on outdoor scenes. *Pattern Recognition*, 10:145–147, 1978.

[DMPV91] M. Danelutto, R. Di Meglio, S. Pelagatti, and M. Vanneschi. High level language constructs for massively parallel computing. In M Baray and B Ozguc, editors, *Computer and Information Sciences IV*, pages 777–788. Elsevier Science Publishers B.V., 1991.

[DR81] J Darlington and M Reeve. ALICE - A multi-processor reduction machine for the parallel evaluation of applicative languages. In *Proc. ACM Conf. Functional Programming Lang. Comput. Arch.*, pages 65–75, 1981.

[Geh84] N. H. Gehani. Broadcasting Sequential Processes. *IEEE Trans. Software Engineering*, pages 343–351, Jun 1984.

[GH78] J. V. Guttag and J. J. Horning. The Algebraic Specification of Abstract Data Types. *Acta Informatica*, 10:27–52, 1978.

[Gut77] J. V. Guttag. Abstract Datatypes and the Development of Data Structures. *Communications of the ACM*, 20:397–404, 1977.

[HJ92] Kevin Hammond and Simon L Peyton Jones. Profiling scheduling strategies on the GRIP parallel reducer. In H Kuchen and R Loogen, editors, *Proceedings of the 4th International Workshop on the Parallel Implementation of Functional Languages, Aachen, Germany*, September 1992.

[Hou62] P. V. C. Hough. Method and Means for Recognising Complex Patterns. U.S. Patent No. 3069654, 1962.

[Hud86] Paul Hudak. Para-Functional Programming. *Computer*, 19(8):60–70, August 1986.

[Hug89] J. Hughes. Why Functional Programming Matters. *J. Comput.*, 32(2):98–107, 1989.

[IK87] J. Illingworth and J. Kittler. The Adaptive Hough Transform. *IEEE Trans. Pattern. Analysis and Machine Intelligence*, PAMI-9(5), 1987.

[IK88] J. Illingworth and J. Kittler. SURVEY: A Survey of the Hough Transform. *CVGIP*, 44:87–116, 1988.

[JCSH87] 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.

[Jon89] S. L. Peyton Jones. Parallel Implementations of Functional Programming Languages. *Computer J.*, 32(2):175–186, 1989.

[Jon91] G.A. Jones. 3D Plane Segmentation from Range Data using the Hough Transform Technique. Technical Report LAIRD-MAN-UOS-GAJ-911201, LAIRD Consortium: University of Surrey, Feb 1991.

[Kou93] 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.

[LB87] V. F. Leavers and J. F. Boyce. The Radon transform and its application to shape parameterization in machine vision. *Image Vision Comput.*, 5:161–166, 1987.

[LDM+89] 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.

[Lea93] V. F. Leavers. Survey: Which Hough Transform? *CVGIP: Image Understanding*, 58(2):250–264, Sep 1993.

[LYT93] Z. N. Li, B. Yao, and F. Tong. Linear GHT and its Parallelization. *Image Vision Computing*, 11:11–24, 1993.

[McA90] P. McAndrew. *Recognising and Locating Objects in Two Dimensional Perspective Views*. PhD thesis, Heriot-Watt University, Dept. of Computing and Electrical Engineering,, Sep 1990.

[MNV73] Z. Manna, S. Ness, and J. Vuillemin. Inductive Methods for Proving Properties of Programmes. *CACM*, 16(8):491–502, Aug 1973.

[NZ88] T. Nagata and H. B. Zha. Determining Orientation, Location and Size of Primitive Surfaces by a Modified Hough Transform Technique. *Pattern Recognition*, 21(5):481–491, 1988.

[Pel93] S. Pelagatti. *A methodology for the development and the support of massievly parallel programs*. PhD thesis, Università di Pisa-Genova-Udine, Italy, March 1993.

[Ski90] David B Skillicorn. Architecture-Independent Parallel Computation. *Computer*, 23(12):38–51, December 1990.

[Som92] I. Sommerville. *Software Enghineering*. Addison-Wesley, 4 edition, 1992.

[VG81] T. M. van Veen and F. C. A. Groen. Discretization Errors in the Hough Transform. *Pattern Recognition*, 14(1-6):137–145, 1981.

[WMM90] 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.

[WMM+92] 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.

[YA85] S. Yalamanchili and J. K. Aggarwal. A System Organisation for Parallel Image Processing. *Pattern Recognition*, 18(1):17–29, 1985.