A Method for Developing Parallel Vision Algorithms with an Example of Edge Tracking

N.R. Scaife, G.J. Michaelson and A.M. Wallace
Dept. of Computing and Electrical Engineering
Heriot-Watt University
Edinburgh EH14 4AS

Research Memo RM/96/1

January 30, 1996

Abstract

We present a general approach to parallel algorithm development based on protyping and benchmarking in a functional language. Preferred solutions are then translated to an imperative language to run on a MIMD architecture. The methodology is well-suited to vision algorithms, since these show a high degree of data or algorithmic complexity at several levels, and are necessarily dependent on parallel processing for rapid computation. The approach is demonstrated fully on a relatively difficult parallel task, tracking boundaries in an intensity image, showing how the functional prototype provides accurate predictions of the final parallel code. Prototyping is in Standard ML for parallel implementation in occam2 on a transputer array.

Keywords: functional prototyping, parallel vision, distributed memory, edge tracking

Telephone: 0131-451-3422 (Michaelson) -3423 (Wallace)

Fax: 0131-451-3431

email: norman,greg,andy@cee.hw.ac.uk

Correspondence: Dr. Andrew Wallace at above address

1 Overview

1.1 Functional programming for parallelism

Functional programming is an excellent basis for general system development [1]. The combination of canonical data structure representations, pattern matching, case structured functions and recursion enables the construction of succinct programs whose structures correspond closely to those of the data they process. Functions are the primary building blocks of programs and may be treated just like any other values. In particular, common control sequences may be generalised through higher order functions (HOFs), whose parameters and results are functions. When a higher order function is called with a specific function argument, a specialised function is returned. This enables a high degree of modularity and of code reuse. Finally, functional languages are grounded in formal computing theory, easing the proof and transformation of functional programs [2].

Functional programming is particularly useful for parallel system development [3]. The Church-Rosser property of evaluation order independence results in potential parallelism being found at all levels of a functional program. Components of a functional program cannot interact through changes to shared objects. This implies that provided we can define an algorithm in a functional way, we can safely transform the code to run concurrently on a parallel machine, even where the time complexity is data dependent and the order of completion is not predictable.

A functional program may be treated as a prototype for an eventual parallel implementation [4]. Results from the instrumentation of a functional prototype based on higher order functions, combined with performance models for parallel skeletons [5], enable the accurate prediction of the behaviour of the equivalent parallel implementation. The transformation of such programs, for example to improve predicted performance, leads to an equivalent reconfiguration of the corresponding skeleton based implementation.

1.2 Parallelism for vision

Vision is a complex, multi-layered process. The first significant use of parallelism was for low level processing of arrays of pixels in which both the input and output data structures were static. Since the algorithms were often based on local window operators, it was possible to use comparatively simple processors programmed in special purpose languages, and to achieve almost linear increases in processing speed. Further exploitation of parallel processing included the extraction of labelled boundaries or regions/surfaces. In general, while the input data structure was still static and pixel based, the output structure was a more dynamic representation of the scene, in which the number of primitives, i.e. boundaries or regions, was not known generally in advance. SIMD and MIMD parallelism have also been employed for higher level processes such as scene interpretation, although there are fewer examples when compared with the lower level processes, as simple data or functional decomposition is more difficult. Recent surveys of progress on parallel vision algorithms have been published by Chaudhary and Aggarwal [6] and Yalamanchili and Aggarwal [7]. Although parallel algorithm development in computer vision has concentrated primarily on modules, rather than complete systems, there have been examples in which independent parallel algorithms have been linked together in a pipe to form visual systems for pick and place operations [8] or for object recognition [9].

1.3 Skeleton based parallel programming

A schematic of our methodology is shown in Fig. 1. Using HOFs, an algorithm is developed in a functional language. Each HOF has associated with it a corresponding skeleton, which is defined as a 'pre-packaged parallel algorithmic form' [10]. Each skeleton has a meaning defined by the functional language definition, and has a machine-independent behaviour which allows implementation of a functional prototype on any parallel machine. The specific behaviour of skeletons on particular machines is defined by performance models; the low level code for implementing the skeleton is fixed and the complexity of the operation is known and predictable. Hence a programmer can develop parallel algorithms at a high level of abstraction in a functional language, and obtain reliable estimates of the performance of these algorithms on specific parallel machines through prototype instrumentation using typical and pathological data sets.

Figure 1 should be inserted here.

If the performance models show parallel advantage then appropriate skeletons are instantiated for the HOF's on the chosen target machine. If useful parallelism is not identified, a sequential code module may be developed to run on a single processor. Alternatively, we can transform the functional prototype to an equivalent program using sets of transformation rules [11]. The benchmarking/evaluation loop is then repeated.

In our case we prototype in Standard ML for implementation on a transputer based Meiko Computing Surface in occam2. Translation is primarily a manual process, although an experimental compiler from SML to occam2 has been implemented which recognises a subset of HOF's [4]. Manual conversion does place a discipline on the programmer but provided translation rules are strictly followed then the referential transparency and performance predictions of the original functional prototype will be maintained. If imperative constructs are added to "improve" the short term performance then the final implementation may be unpredictable and the process distribution may not be efficient.

2 Expressing Vision Algorithms in Standard ML

2.1 Basic operations in SML

Standard ML (SML) was developed from the meta-language for Milner's LCF theorem prover. It is actually an imperative language but it has a powerful pure functional subset which we employ here. SML programs are based on function definitions and expressions that use them. Types are fundamental to SML, enabling compile time checks to ensure that program parts have been bolted together correctly. In particular, a function's type identifies it's parameter and result type, providing interface information for subsequent use. SML systems show a function's type as a mapping from a domain of formal parameters to a range for the results.

SML provides the usual integer, real, boolean and string atomic types. The tuple data structure enables the grouping of a fixed number of values of *different* types.

A function declaration associates a name and a function value consisting of a pattern of bound variables (i.e. formal parameters) and a body expression. For example, to translate an (x, y) coordinate by d along the x-axis,

```
- fun xtran (d:real) (x,y) = (x+d,y);
> val xtran = fn : real -> real * 'a -> real * 'a
```

This is a function of one parameter `d` which returns a function of one parameter `(x,y)`, not a function of two parameters. No operation is applied to `y` so it may be of any type, denoted by `'a`.

Functions may have several patterns with associated bodies. When such a function is called, each pattern is matched in turn against the argument. When a match succeeds, the associated body is evaluated. Patterns may contain constants as well as bound variables. For example, to negate a binary pixel:

```
- exception Neg;
> exception Neg
- fun neg 0 = 1 |
      neg 1 = 0 |
      neg n = raise Neg;
> value neg = fn : int -> int
```

Note the exception `Neg`. If the function `neg` is called with a value other than `0` or `1`, the third case will succeed, setting `n` to that value and returning control to a system which identifies the exception.

Pattern matching can only determine the presence or absence of values. To test arbitrary properties of values an explicit conditional expression is used. For example, to check if one point is to the left or right of another point:

```
- fun xpos (x1,y1) (x2,y2) =
       if x1<x2-0.0000001
       then "left"
       else
        if x1>x2+0.0000001
        then "right"
        else "close";
> val xpos = fn : real * 'a -> real * 'b -> string
```

The types of `y1` and `y2` are unknown and different. `y1` may be of any type `'a` and `y2` may also be of any type `'b`.

In addition to tuples, lists are a central abstraction for data representation. A list is a sequence of values of the *same* type, ending with the empty list `[]`. Lists are formed with the constructor `::` between a value, the list head, and a list of the same type as the value, the list tail, e.g. `1::2::[]`. A list is displayed in a compact form as a sequence of '`,`' separated values within '`[...]`', e.g. `[1,2]` .

The list constructor may be used in bound variable patterns. For example, we might negate all of a list of pixels:

```
- fun negall (0::t) = 1::negall t |
      negall (1::t) = 0::negall t |
      negall (h::t) = raise Neg;
> val negall = fn : int list -> int list
```

Here, the pattern `(0::t)` matches a list whose head is `0`, sets bound variable `t` to the tail and returns a list whose head is `1` and whose tail is the negated tail of the original list. Similarly, the pattern `(1::t)` matches a list whose head is `1`, sets bound variable `t` to the tail and returns a list whose head is `0` and whose tail is the negated tail of the argument list. Finally, for any other list, `h` is set to the head, `t` is set to the tail, and the negation exception is raised to stop processing.

2.2 Image data structures in functional languages

Lists and tuples are used extensively in vision algorithms, for example to represent regions in a segmented image, or point coordinates in 3D space. However, the pixel array is a fundamental image data structure, providing a multi-dimensional matrix of changeable elements, each of which can be accessed at random. Arrays generalise variables as modifiable name/value associations and so are contrary to the basic notion of referential transparency in a pure functional language; the basis of evaluation order independence. This problem is now addressed.

Unlike imperative languages, pure functional languages lack assignment. Functions cannot affect each other by changing shared variables so the evaluation order is not necessarily prescribed. This is the basis of functional parallelism, and theories about functional languages need not make concepts of time explicit, making it comparatively easier to develop formal systems for program proof and transformation.

Arrays are direct access structures with constant access costs, $O(1)$. Individual elements may be changed *in situ*. These assignable data structures are inconsistent with pure functional languages. In a functional language, we can use a list structure to represent a one dimensional array, a list of lists to represent a two dimensional array, and so on. Therefore, an arbitrary element is accessed by visiting all the preceding elements. Similarly, the effect of modifying an arbitrary element is achieved by copying the whole structure with a change in the appropriate place. Here, both access and "modification" have linear cost, $O(N)$.

Given that our goal is to develop prototypes to obtain reliable estimates of parallel performance in a target language which *does* allow array structures with random access, it might appear that functional languages are not an appropriate choice for parallel program development. On the contrary, we can still employ functional prototypes successfully, identifying two types of vision algorithm.

In the first instance, many algorithms are based on regular traversal of an image, for example negating an image or computing the mean and standard deviation of pixel intensity. Here, the cost of the nested list representation is the same as that of the array representation.

In the second instance, arbitrary, or at least non-regular array access is required. The example we shall develop later is tracking a boundary in an array of edges. As noted above, the cost of arbitrary pixel access in a list of lists is substantially greater than in an array. If we use a functional prototype as a basis for instrumentation of our final parallel occam2 program then the time taken to access the nested list representation will be a very poor predictor of performance.

There are two ways to solve this problem. First, some functional languages such as New Jersey SML provide imperative array packages so that consistent constant time prototype behaviour is achievable. However, the direct use of arrays does compromise program transformation as the evaluation order may become significant. Therefore we use the instantiation of matrix representations into arrays only after program transformation is complete. This provides accurate predictions for the occam2 programs.

Alternatively, we could hide the representation of images within an abstract data type.

Provided that whole images are always passed explicitly from function to function, rather than manipulated indirectly through assignment, then higher order functions and transformation rules for lists may be reconstructed with respect to the image type. Instrumentation then requires an *estimate* of the the constant array access time, rather than the *actual* time from the functional implementation.

2.3 Higher order functions

Our approach is based substantially on higher order functions (HOFs), i.e. functions which take other functions as arguments or yield functions as results. These can be used as generic operators on list structures. The first fundamental HOF is the `map` denoting linear processing of a list, where the same activity is applied to each element.

```
- fun map f [] = [] |
      map f (h::t) = f h::map f t;
> val map = fn : ('a -> 'b) -> 'a list -> 'b list
```

Function `f` is applied to each element of a list in turn to form a new list. If the list is of type `'a list` and the result is a list of possibly different type `'b list`, then the function `f` must have an `'a` argument and return a `'b` result. For example, to translate all of a list of points by `10.0` along the x-axis:

```
- val xtran10all = map (xtran 10.0);
> val xtran10all = fn : (real * 'a) list -> (real * 'a) list
```

The second fundamental HOF is the `filter` which selects elements from a list according to some criterion:

```
- fun filter p [] = [] |
      filter p (h::t) = if p h
                        then h::filter p t
                        else filter p t;
> val filter = fn: ('a -> bool) -> 'a list -> 'a list
```

`p` is some predicate which when applied to a value of type `'a` returns a boolean to indicate whether or not that value satisfies the predicate. Thus, `filter` checks each element of a `'a list` and keeps those that satisfy `p`.

For example, to select all the points in a list which lie above the x axis:

```
- fun abovex (x,y) = y>0.0;
> val abovex = fn : 'a * real -> bool


- val allabovex = filter abovex;
> val allabovex = fn: ('a * real) list -> ('a * real) list
```

The third fundamental HOF is `fold`. Given a list of values, an operation is applied to successive elements to form a final value. The general case is to apply some function `f` to each value and the result of applying `f` across the rest of the values, where `b` is some base value:

```
- fun foldr f b [] = b |
      foldr f b (h::t) = f h (foldr f b t);
> val foldr = fn : ('a -> 'b -> 'b) -> 'b -> 'a list -> 'b
```

For example, to sum an integer list:

```
- fun add (x:int) y = x+y;
> val add = fn: int -> int -> int


- val addall = foldr add 0;
> val addall = fn: int list -> int
```

This HOF is called `foldr` because it folds a function across a list from left to right. A function may also be folded across a list from right to left:

```
- fun foldl f b [] = b |
      foldl f b (h::t) = foldl f (f b h) t;
> val foldl = fn : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

`foldl` and `foldr` are equivalent when the folded function is both associative and commutative, for example addition. As we shall see, this leads to simple parallelism. If the function is neither associative nor commutative, for example subtraction, then it is much harder to find parallel equivalence since the result is dependent on the order of evaluation.

The final HOF is for explicit function `composition`. Consider applying a function `f` to the result of applying a function `g` to some value `x`:

```
- fun compose f g x = f (g x)
> val compose = fn : ('b -> 'c) -> ('a -> 'b) -> 'a -> 'c
```

For example, we may translate all the points above the x axis by `10.0` along the x axis:

```
- val xtran10allabovex = compose xtran10all allabovex;
> val xtran10allabovex = fn : (real * real) list -> (real * real) list
```

It is usual to use the infix operator `o` for composition, for example

```
- val xtran10allabovex = xtran10all o allabovex;
```

2.4 Higher order functions and parallelism

HOFs have natural parallel skeleton equivalents on MIMD architectures. Therefore, if we can specify algorithms in this way, then we can instantiate the appropriate skeletons having applied the machine specific performance models.

The `composition` function may be implemented as a pipeline of processes; this skeleton provides linear job parallelism as the data can be streamed through each process in the pipeline, as shown in Fig. 2.

Figure 2 should be inserted here.

The `map` function may be implemented as a process farm, illustrated in Fig. 3. A farmer is connected to a chain of workers each of which can run the `map` function `f`. The farmer receives the argument list and sends each element down the chain to a free worker. The worker applies `f` to the element and returns the result. Finally, the farmer assembles the new list.

Figure 3 should be inserted here.

The `filter` function may be implemented as a modified process farm, illustrated in Fig. 4. Each worker runs the `filter` predicate `p` and must indicate whether or not the elements satisfy the predicate. The farmer keeps or rejects list elements accordingly. The values must also be indexed to retain list order.

Figure 4 should be inserted here.

For `foldr` and `foldl`, strict left to right or right to left sequential processing is unavoidable unless the folded function `f` is associative and commutative, as discussed in the previous section: Then, `foldr` and `foldl` are equivalent and may be implemented as a tree structured divide and conquer process hierarchy, illustrated in Fig. 5. Each process splits its input list and sends half to each of its two children. It then receives their results and applies the folded function `f` to them.

Figure 5 should be inserted here.

2.5 Transformation with higher order functions

Program transformation is used to change programs [12], for example to make them more efficient, without affecting their meaning. For parallel programs, it is important to find optimal balances of communication and processing. Where communication costs outweigh processing costs, transformation may be used to group program components together to increase processing relative to communication. Alternatively, where processing costs are much higher than communication costs, transformation may be used to ungroup program components so that they run in parallel.

A wide variety of transformations on compositions of higher order functions have been proved correct. Here we present only a few examples and show their effects on the equivalent parallel process configurations.

Consider the composition of three functions as a process pipeline. Composition is associative:

function1 o *function2* o *function3* <==>

(*function1* o *function2*) o *function3* <==>

function1 o (*function2* o *function3*)

Bracketing composed functions together may be viewed as grouping what were separate processes into a single process shown in Fig. 6.

Figure 6 should be inserted here.

Second, composition distributes over mapping, so that a pipeline of farms transforms to a farm of pipelines shown in Fig. 7.

(`map` *function1*) o (`map` *function2*) <==>

`map` (*function1* o *function2*)

Figure 7 should be inserted here.

As a further example, filtering and mapping may be combined through the function:

```
- fun mapfilter p f [] = [] |
      mapfilter p f (h::t) = if p h
                             then f h::mapfilter p f t
                             else mapfilter p f t
> val mapfilter = fn : ('a -> bool) -> ('a -> 'b) ->
                        'a list -> 'b list
```

so:

`map` *function1* (`filter` *function2 list*) <==>

`mapfilter` *function2 function2 list*

Thus a pipeline of a filter farm and a map farm may be transformed to a single `mapfilter` farm, illustrated in Fig. 8.

Figure 8 should be inserted here.

3 Prototyping the boundary tracker

In this section, we demonstrate the effectiveness of our approach through the example of a parallel boundary tracker. This is perceived as a difficult problem [7] because communication patterns are not necessarily regular and are highly data-dependent. Thus, it forms a severe test of our functional approach because of the pseudo-random tracing of edge points within the image structure, and because the final data structures (i.e. the boundaries formed by linked edges) are not known in advance.

We present two solutions which are applied to edge strength and orientation maps produced by the Canny edge detector [13]. The merits of each algorithm and likely parallel implementations are discussed, leading to a third, hybrid algorithm which balances the merits of the competing approaches.

3.0.1 Algorithm 1: sequential tracking from seed points

An *imperative* expression of an edge tracking process is defined in Figure 9. This is based on a similarity measure between pixels; the accurate detection of lines is a balance between the ability to reject noise points but to effectively negotiate discontinuities. Popular metrics include edge strength and angular continuity, segment curvature or proximity to existing contours. In our algorithm, all points over a given strength threshold are treated as seed points and a simple angular threshold is used to guide the growing process.

Figure 9 should be inserted here.

This algorithm can be directly encoded in SML, outlined in figure 10. The basic structure is a `fold` operation over the entire image, in which each point is a potential seed point and those values over the edge strength threshold are tracked in an imperative style tracking operation. The `fold` accumulates grown lines in a separate list.

The tracking process is a recursive function using the size of the list of remaining neighbouring points as a convergent. For efficiency, all the neighbours for each pixel are checked at each point by `fold`ing over a list of neighbouring point offsets and the resulting list of compatible neighbours is then added to the list of pending neighbours.

Figure 10 should be inserted here.

In general, the order of points in the final lines is not contiguous across the lines. This will affect later processing where point order is required, for instance to find the end points of the lines. However, there will be a great deal of structure within the point lists which could be exploited by a subsequent sorting process. For instance straight lines are guaranteed to be ordered up to branches and intersections. The ordering is set by the fixed order in which the neighbours are checked and this could itself be used to improve the final ordering using a more rigorous regime.

In this algorithm, the `getedge` and `getangle` operations work by scanning the image from $(0, 0)$ each time. To access pixel (x, y) in image ea the construct `nth (nth (ea,y),x)` is used to select the `nth` member of a list. However, these are easily converted into array based processing using New Jersey Standard ML's `Array2` facility. This is the basis for the second version of the program where the `getedge`, `getangle` and `member (x,y) V` constructs are replaced by array manipulation. The array `a2img`, an array of tuples of pairs of integers of type `(int * int) Array2.array2`, is created by converting the list based image. A similar method is used to construct the `V` array. The `a2img` operations are shown in Figure 11. As an illustrative example, this reduces the processing time for a 256 by 256 pixel image from 232s to 4s of user time.

Figure 11 should be inserted here.

When working with arrays in SML, unlike list based processing, there have to be numerous explicit tests for array bounds which make the code untidy. When an error occurs in an array based operation it can be much harder to track down. However, this version of the program is likely to behave much more like a parallel implementation that uses arrays so the lack of functional purity may be tolerated.

3.0.2 Algorithm 2: pairwise comparison of sparse point lists

The second version of the edge tracking algorithm is based on an alternative description of the process in Figure 12. This is much closer to a declarative description of the problem which states that the set of tracked edges is the set of sets of pixels such that each pixel is compatible with and a neighbour of at least one other pixel in the same set and with no other pixels in the other sets.

Figure 12 should be inserted here.

This algorithm works by starting from a list of singleton lines derived by thresholding the image. Each line in the image (`L1`) is then compared with all lines that have already been checked (`LL'`) and a list of compatible edges (`LLC`) and a list of non-compatible lines (`LLNC`) are built up. After each line is checked against all existing lines a new list of lines with no compatible pairs is generated from these lists. The result is a set of tracked edges which can be filtered to remove small lines (it is likely that there will be a substantial quantity of singleton lines remaining when the algorithm terminates).

The SML algorithm is shown in Figure 13. The scanning of point and line lists is carried out by a series of nested `fold` operations. This program is much more succinct, showing the declarative nature of the specification.

Figure 13 should be inserted here.

The problem with algorithm 2 is that it is effectively of $O(M^2)$ complexity, where M is the number of thresholded edges in the image, since each edge has to be compared with all other edges. This means that the parallel performance of the system has to be very good to overhaul the fast imperative sequential algorithm.

3.0.3 Algorithm 3: a hybrid edge tracking algorithm

Neither version of the first algorithm is ideal for direct implementation from the prototype. The first version requires access to the data in a manner that is difficult to express in pure functional SML, bearing in mind that we wish to keep as close a correspondence between the prototype and implementation as possible. The second approach has higher computational complexity so direct, fine-grained implementation of this algorithm is unlikely to lead to respectable parallel performance compared with a sequential implementation of the imperative method. It is also expressed as nested, non-associative `fold` functions which cannot be implemented in parallel.

The `genlist` function in Figure 13 is not associative. However, this can be overcome by spotting that the only difference between the left and right folded outputs (`foldl` and `foldr`) is the order of the points in the lines and the order of the lines in the lists of lines, for example

```
- val xyal = [[(0,0,15)],[(3,0,65)],[(1,1,15)],[(2,1,64)],
              [(1,2,62)],[(2,2,250)],[(0,3,61)],[(3,3,250)]];

- foldl genlistl [] xyal;
val it =
  [[(3,3,250),(2,2,250)],[(1,1,15),(0,0,15)],
   [(0,3,61),(1,2,62),(2,1,64),(3,0,65)]] : (int * int * int) list list

- foldr genlistr [] xyal;
val it =
  [[(0,0,15),(1,1,15)],[(3,0,65),(2,1,64),(1,2,62),(0,3,61)],
  [(2,2,250),(3,3,250)]] : (int * int * int) list list
```

This implies that we could implement an edge tracking system using an associative fold. However, we still have to maintain the property of the non-associative folding system that ensures all points are compared with all other points. We can use the comparison operator to `filter` a list of all point pairs and implement the merges as a sequential operation. This is illustrated in Figure 14.

Figure 14 should be inserted here.

The characteristics of these prototypes suggest that a composite (or hybrid) algorithm may be more suitable for coarse-grain distributed memory systems such as networks of transputers. On the one hand we have the fast sequential method (algorithm 1) which is highly suitable for implementation on a single transputer and on the other we have a method for merging small sets of lines (algorithm 2) which is suitable for situations in which M is small enough for the $O(M^2)$ complexity to be less prominent. Combining these two methods into a single parallel algorithm therefore involves running the sequential tracking algorithm on disjoint regions of the image (a *tracking* phase), generating modestly sized lists of line segments and then running the line merging system to combine these segments (a *combine* phase).

Figure 15 should be inserted here.

This solution still has one major drawback. It depends for its efficiency on a substantial reduction in the volume of data at the first phase. This is not guaranteed since noisy images tend to produce a large number of short or single-point lines which are not aggregated during tracking. We have reduced the probability of this by implementing the first phase such that *only* lines which are either greater than a minimum line length or continue over the edge of the image portion into the next portion are retained.

4 Instrumentation and evaluation of the edge tracking algorithm

At this point we have an SML prototype constructed using a set of HOFs that we know have efficient parallel equivalents. The objective now is to investigate the viability of the parallel implementation by making measurements on the behaviour of the working SML code.

The exact measurements required depend on the nature of the performance model. Static program analysis can be divided up into *microanalysis* [14] where the costs of elementary functions are accumulated and *macroanalysis* in which a function is characterised by the number of times a dominant part of the function is executed. Static analysis has practical disadvantages; one problem is the treatment of branches since the average execution time is dependent on the number of times each branch is taken. An alternative solution is *benchmarking* where measurements are made during program run-times. This can be done in two ways; *sampling based* profiling where the program is halted at regular intervals and its current execution position noted and histogrammed over time [15] and *counter based* profiling where specific points in the program are associated with counters which are incremented each time that point in the program is executed [16].

For our purposes we intend to use an arbitrary topology of transputers to implement `map` and `filter` using processor farming. Our performance model is based on the size of the inputs and outputs of the functions (or predicates) evaluated by the remote processors and the average time required to execute these functions. Thus we use an annotated program in which we replace the centres of parallelism (for instance a suitable `map` function) with equivalent functions which collect the information we require.

We have implemented a complete SML prototype based on Figure 15. This comprises a `map` over a geometric decomposition, returning partial line segments followed by a `filter` over all pairs of line segments feeding into a non-associative `fold`. This identifies two potential centres of parallelism, the `map` over the geometric decomposition and the `filter` over all pairs of partial lines. These HOFs are replaced by calls to equivalent functions which accumulate data sizes and function execution times. Figure 16 shows the equivalent function for the `filter` skeleton.

Figure 16 should be inserted here.

Table 1 shows the average processing times and input and output sizes for the first parallel construct, the `map ffportion` component in `geomdecomp` for various sub-divisions of the test image of Figure 22(b) into $\sqrt{N} \times \sqrt{N} = N$ portions. Execution was in New Jersey Standard ML Version 0.93, on a Sun Sparc 10 system. The mean and standard deviations are computed from 10 executions of the SML code in each case. The figures are data and parameter dependent [9], but may be considered representative for images of this type, in which case the major parameter in the final parallel implementation is the granularity of the decomposition.

4.1 Evaluation of the performance model

The next activity is to predict the performance of the parallel implementation from the measurements made on the prototype, applied to images similar to Figure 22. A detailed

Averages for `etportion` per portion				
Decomposition	Time (s)	Time (σ)	Input size (bytes)	Output size (bytes)
2 × 2	1.282	0.045	8.0	269.8
3 × 3	0.922	0.038	8.0	119.4
4 × 4	0.814	0.058	8.0	61.2
5 × 5	0.745	0.042	8.0	41.1
6 × 6	0.718	0.035	8.0	26.9
7 × 7	0.730	0.048	8.0	17.7
8 × 8	0.709	0.035	8.0	13.0
9 × 9	0.704	0.038	8.0	10.8

Table 1: SML timing summary for *tracking* phase

investigation into the performance modelling of farms and pipelines was given by Pelagatti [17]. Simpler models to predict processor requirements in the context of an automatic parallelising compiler were given by Bratvold [4]. A performance model for processor farms can also be found in [18].

A common approach is to estimate likely parameters for the process, such as the time to apply a function to a set of data (t_f), the time for the worker to acquire (t_k) and pass on (t_p) a given data element, the number of data portions (N) and the number of processors (P). Generally, the analysis proceeds in worker time space with a purely empirical treatment of parameters such as startup overheads and network latencies. Bratvold [4] used a linear pipeline of processors and assumed a linear relationship between a processor's position in the pipeline and the number of data elements computed by that worker. Predictions were typically within 10-15% of the actual measured values.

This purely theoretical modelling has limited use when some of the parameters are difficult to predict. For instance the relationship between a processor's position in an arbitrary topology and the number of elements computed is non-trivial, even for simple arrangements such as 2D arrays. Here, we present a much simpler method which does not make any assumptions about the physical nature of the system being modelled. However, it is tied to one particular implementation of a skeleton and is likely to be well-behaved only for parameters within a small range of values.

The idea is to maintain *live* models of our parallel constructs, executing on the target parallel architecture and parameterised according to the data provided by the prototype measurements. For instance, we implement a processor farm with delays for processing times and artificially generate input and output data sets of a given size and distribution to model the implementation of a `map` HOF. Plugging the prototype measurements into this working model gives a system which behaves like our intended implementation. The "dummy" processor farm is applied to a suitable range of values for the parameters and then a function may be fitted to the performance as the dependent variable, which is used for subsequent predictions of behaviour from new prototype instrumentation data. The controllable parameters are the number of processors (P), number of portions (N), average processing time per element ($t_f\mu$) and standard deviation of the t_f values, ($t_f\sigma$). Figure 17 shows the performance of the dummy farm when run using parameters taken from the performance of the tracking phase prototype.

Figure 17 should be inserted here.

There are several points to note about this modelling. If N is not a multiple of P and there are relatively few portions compared to the number of processors then the odd portions drastically affect the performance (consider 3 portions processed by 2 processors). We have specifically avoided this problem by forcing N to be a multiple of P, although if the variances in the t_f values become large this problem could still arise.

Furthermore, the communications size is not included in the parameters. It was found by trial and error that better fits were obtained by omitting these, the small size of the data being transmitted means that the performance is not greatly dependent on the communications load, although if much larger numbers of processors were required the communications may become more significant.

To get a prediction from this model we need to take into account the discrepancy in processing times between SML and occam2. One approach is to match up constructs between the two languages and use only relative processing times in the prediction, this approach can be useful in predicting processor loadings in pipelines [9]. Alternatively, we could generate the occam2 automatically from the SML [11] which would ensure closer correspondence between the two languages, although possibly at the expense of sequential performance.

Here we use a different approach; equivalent functions are written in SML and sequential occam2 and timed on the same data sets. This gives a direct comparison between the two and allows a simple correction to be applied, Figure 18 shows average SML and occam2 runtimes for the `etportion` function which indicates a fixed overhead in the processing times for the SML function which can be taken into account when predicting the parallel occam2 performance.

This discrepancy is a result of the impedance mismatch between our source and target languages and would be less of a problem if the same language were used for both prototyping and implementation. Making performance predictions across language paradigms is a potential weakness in our methodology, yet we have found reasonably accurate predictions to be possible even with simple corrections like that given in Figure 18. This process of prediction should be considered a part of the performance model which is used to design a particular implementation and is not intrinsic to the use of skeletons.

Figure 18 should be inserted here.

4.2 Parallel implementation

The occam2 implementation is a straight encoding of the SML code, placing top-level sequential code on the farmer processor, `map` and `filter` operators on the worker processors and implementing both `map` and `filter` as processor farms.

The structure of the final version is illustrated in Figure 19. This indicates a pipeline of four processes, two of which are sequential (`indx/comb` and `foldl`) and two of which are parallel (`map` and `filter`). We could implement these physically on separate processors. However, the prototype indicates that there is a great disparity between the amount of computation between the individual elements in the pipeline, which would make this difficult to balance. For instance, the `indx/comb` node has almost no computation at all. The system is therefore implemented as a single farmer with dual purpose workers processing successively the `etportion` and `compatl'` functions.

Figure 19 should be inserted here.

Figure 20 gives a comparison of the **measured** parallel performance of the tracking phase implemented as a processor farm in occam2 with that **predicted** from the measurements on the dummy farm for 25 image portions. In this case, we have compared the final implementation with the predictions for 1 to 26 processors and obtained excellent agreement. The effect of residual computations in parallel processing can be seen in the deviation of the measured from the predicted values except for 25 processors (1 portion per processor) and 12 processors ($\approx$ 2 portions per processor). The prediction is effectively an upper bound on performance when there is an even distribution of work among the t_f values.

Figure 20 should be inserted here.

Figure 21 shows the overall performance for the edgetracking algorithm applied to the data of Figure 22(b). This is dominated by the performance of the tracking phase for small numbers of image portions but as this increases the amount of data being passed forward from the tracking phase increases and the $O(M^2)$ complexity of the merge phase starts to take effect. Of course this is data dependent; since the complexity of the first phase is of $O(E)$, where E defines the number of thresholded edges, and of the second phase $O(M^2)$, where M defines the number of line segments passed forward, it depends on both the spatial distribution and number of initial edges. For the sparse image illustrated the balance is manifest as a very sublinear speedup at higher numbers of processors. Nevertheless, the ML prototype combined with the performance model provides an accurate prediction of final performance for equivalent, or similar data, as demonstrated by the example of Figure 20.

Figure 21 should be inserted here.

Figure 22 should be inserted here.

5 Conclusions

The principal objective of our work has been to present an approach to parallel algorithm development based on functional prototyping. This retains the advantages of the pure functional methodology, yet allows usable performance predictions of the final parallel implementation. Our approach is skeleton based, augmented with prototype instrumementation to identify and optimise useful parallelism. Rather than using an abstract general performance model, we have formed a concrete model by instrumenting a parameterised dummy skeleton in occam2 on the target Meiko platform with typical communication and processing loads. Similar instrumentation of the equivalent SML higher order functions enabled the prototype based performance prediction to be compensated for the impedance mismatch between the functional prototype and imperative parallel implementations.

Although our method is general, we have directed our work specifically towards visual processing because of the need to develop parallel algorithms to handle large data sets or algorithms of considerable complexity. Thus a second objective of this paper was to demonstrate the efficacy of the approach on the specific problem of boundary tracking, chosen as a severe test in comparison with previous parallel work on edge detection, Hough transformation, feature matching and pose estimation [19, 9, 20]. The occam2 was implemented directly from the prototype guided by the presence of our recognised HOF skeletons. Good agreement was obtained between the performance of the final algorithm encoded in occam2 and the predictions of the ML prototype. The measured speedup peaked at 16 for 25 processors showing worthwhile improvement on a process which is known to be difficult for parallel implementation.

6 Acknowledgement

We should like to acknowledge the support of the UK Engineering and Physical Sciences Research Council, Grant Reference GR/J07884.

References

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

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

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

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

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

[6] V Chaudhary and J.K. Aggarwal. Parallelism in computer vision: A review. In V. P. Kumar, P. S. Gopalakrishnan, and L. N. Kanal, editors, *Parallel Algorithms for Machine Intelligence and Vision*. Springer-Verlag, 1990.

[7] S. Yalamanchili and J.K. Aggarwal. Parallel processing methodologies for image processing and computer vision. *Advances in Electronics and Electron Physics*, 87:259–299, 1994.

[8] M. Rygol, S.B. Pollard, and C.R. Brown. A multiprocessor system for pick and place. In J.E.W. Mayhew and J.P. Frisby, editors, *3D model recognition from stereoscopic cues* . , 1991.

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

[10] 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.

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

[12] 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.

[13] J. F. Canny. A computational approach to edge detection. *IEEE Trans. PAMI*, 8(6):679–698, 1987.

[14] J. Cohen. Computer Assisted Microanalysis of Programs. *CACM*, 25(10):724–733, Oct 1982.

[15] A. W. Appel, B. F. Duba, and D. B. MacQueen. Profiling in the Presence of Optimization and Garbage Collection. Technical Report CS-TR-197-88, Princeton University, Dept. Comp. Sci., Princeton, NJ, USA, November 1987.

[16] V. Sarkar. PTRAN — the IBM parallel translation system. In B.K. Szymanski, editor, *Parallel Functional Languages and Compilers.* ACM Press, 1991.

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

[18] J. Darlington and H. W. To. Building Parallel Applications without Programming. In *Second Workshop on Abstract Machine Models for Highly Parallel Computers*, Leeds, 1993.

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

[20] N. Scaife, G.J. Michaelson, and A.M. Wallace. Prototyping parallel algorithms using standard ML. In *Proc. of 6th British Machine Vision Conference*, pages 671–680, 1995.

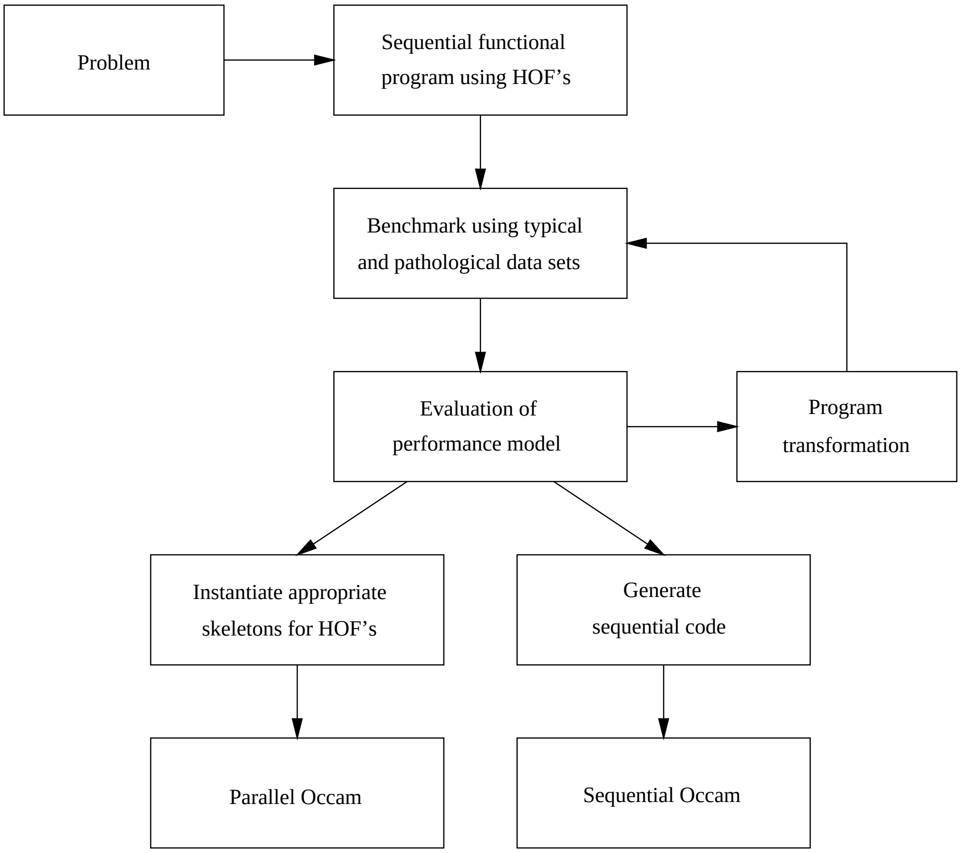


Figure 1: Developing parallel vision algorithms from functional prototypes

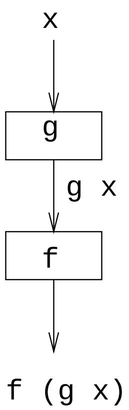


Figure 2: Composition as process pipeline

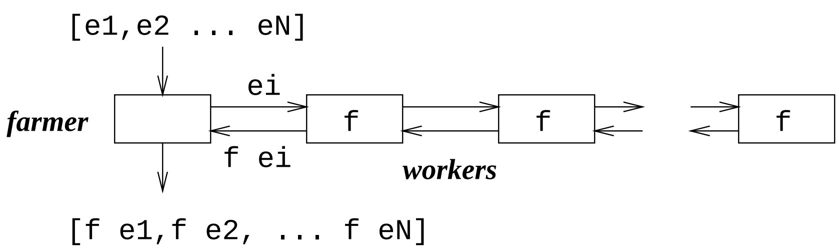


Figure 3: Map as process farm

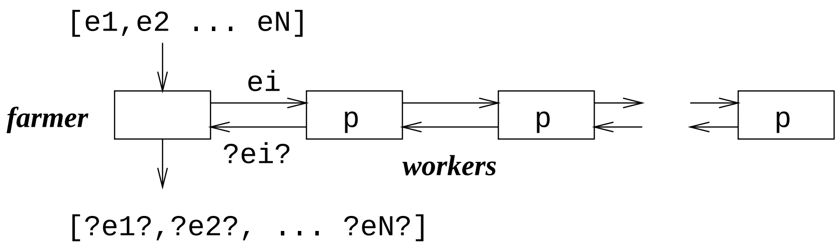


Figure 4: Filter as process farm

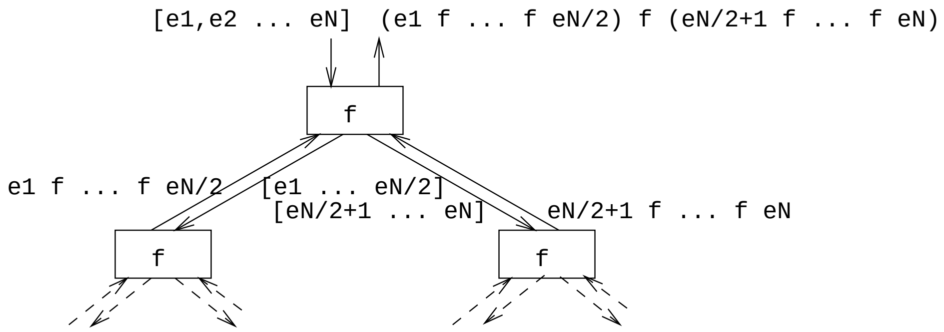


Figure 5: Fold as divide and conquer process tree

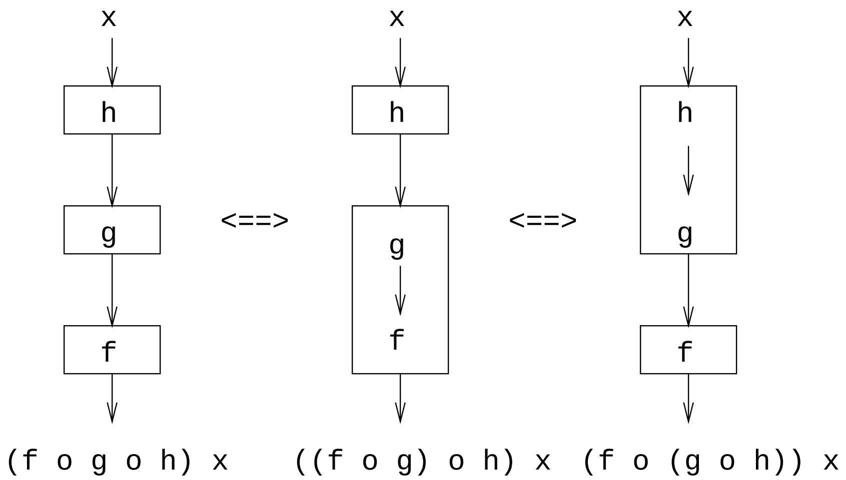


Figure 6: Transforming composition sequences

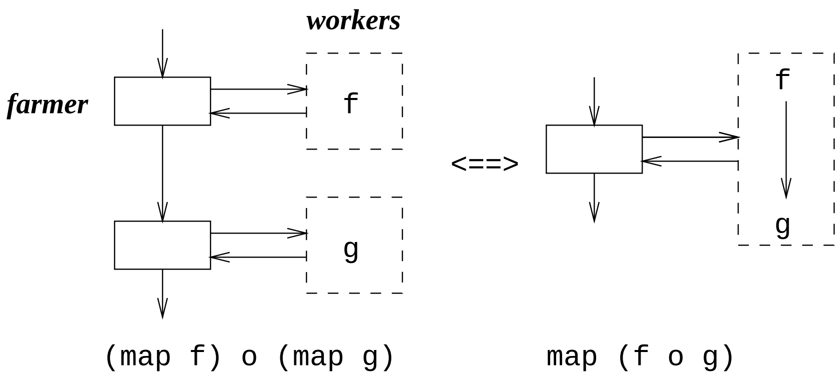


Figure 7: Transforming composed maps

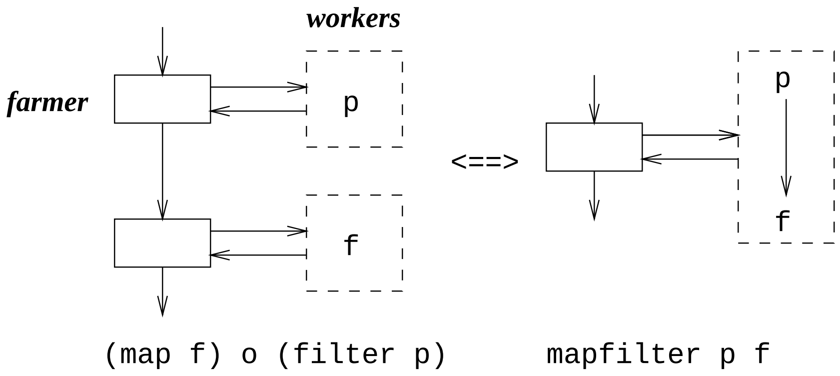


Figure 8: Transforming composed filter and map

```
I = {edges in image}, LL = ∅, V = ∅
for each point p ∈ I, p ∉ V
  if edge_strength(p) > threshold1
    set V = V ∪ {p}, L = {p}, G = {p}
    while G ≠ ∅
      g ∈ G, G = G - {g}
      for each point n in the neighbourhood of g, n ∉ V
        if compatibility_constraint(g,n) < threshold2
          set G = G ∪ {n}, L = L ∪ {n}, V = V ∪ {n}
    LL = LL ∪ {L}
```

`I` is the input image (edge strength and orientation), `LL` is the growing list of tracked lines, `V` is the set of points that have been added to lines, `L` is the line currently being grown and `G` is the set of compatible points in the neighbourhood of `L` still to be checked.

Figure 9: Imperative Description of Edge Tracking

```
fun track V L [] = (V,L)
  | track V L ((x,y)::t) =
  let
    val (NV,l) = foldl (chknbrs (x,y) (getedge x y,getangle x y)) (V,[]) nbrs
  in
    track NV (L @ l) (t @ l)
  end

fun et (V,LL) (x,y,(edgeval,angleval)) =
  if (member (x,y) V) orelse (edgeval < edgethresh)
    then (V,LL)
    else
      let
        val (NV,L) = track ((x,y)::V) [(x,y)] [(x,y)]
      in
        if (length L) >= minlinelen
          then (NV,L::LL)
          else (NV,LL)
      end

val (V,LL) = foldr2ixy et (0,0) ([],[]) ea
```

The core of the list based edge tracking algorithm. The function `chknbrs` accumulates a list `l` of compatible neighbouring pixels that are not in `V` and also returns an updated version of `V`.

Figure 10: Edge Tracking Algorithm in SML

```
(* Turn list based image into array based image *)
val a2img = Array2.array (cols,rows,(0,0))
fun seta2 (x,y,e,a) = Array2.update (a2img,x,y,(e,a))
val dummy = map2ixy seta2 0 0 edges angles

(* Indexing routines for array based image *)
fun get ff x y def =
  if inrange x y
    then ff (Array2.sub (a2img,x,y))
    else def
fun getedge x y = get #1 x y 0
fun getangle x y = get #2 x y (~1 - anglethresh)
```

Figure 11: Array Based Operations for the Algorithm 1

```
I = {edges in image}, LL = ∅
for each point p ∈ I
  if edge_strength(p) > threshold
    LL = LL ∪ {{p}}
LL' = ∅
for each line L1 ∈ LL
  LLC = ∅, LLNC = ∅
  for each line L2 ∈ LL'
    if p1 ∈ L1 is compatible with p2 ∈ L2
      then LLC = L2 ∪ LLC
      else LLNC = {L2} ∪ LLNC
  LL' = {L1 ∪ LLC} ∪ LLNC
```

I is the input edge and angle map image, LL is the initial list of lines, LL' is a growing version of LL containing implemented merges, LLC is a list of points compatible with L1, LLNC is a list of lines containing no points compatible with L1 and p1 and p2 are arbitrary pixels in L1 and L2, respectively.

Figure 12: Alternative Description of Edge Tracking

```
fun genlist ll l =
  let

    fun compatl' l l' =
      let
        fun ff sofar p = sofar orelse (foldl (compat' p) false l')
      in
        foldl ff false l
      end

    fun addlist l (llc,llnc) l' =
      if compatl' l l'
        then (l' @ llc,llnc)
        else (llc,l'::llnc)

    val (llc,llnc) = foldl (addlist l) ([],[]) ll

  in
    (l @ llc)::llnc
  end

fun edgetrack img = foldl genlist [] img
```

This code requires a thresholded image in the form of a linear list of (x,y,angleval) tuples (img). The function genlist is a fold operator that accumulates the original folded list, but merging compatible lines in the list. The function compat' is a fold operator that accumulates a boolean value which is true if any of the folded list elements is compatible with p.

Figure 13: Line Merging Version of the Edge Tracking Algorithm

```
(* Generate list of tuples of all combinations of elements in list *)
fun tup a b = (a,b)
fun maptup x l = map (tup x) l
fun comb [] = []
  | comb (h::t) = (maptup h t)@(comb t)

(* Return true, if two lines contain compatible points *)
fun compatlt (l,l') =
  let
    fun ff sofar p = sofar orelse (foldl (compat' p) false l')
  in
    foldl ff false l
  end

(* Map compatibility function over pairs *)
val compatiblepairs = filter compatlt (comb xyal)
```

The function `comb` generates lists of pairs from a given list (`comb [1,2,3] == [(1,2),(1,3),(2,3)]`). The compatibility function can then be used to `filter` this list to give a list of compatible pairs.

Figure 14: Filtering a list of line pairs with the compatibility operator

```
fun geomdecomp (cols,rows,dc,dr) ffportion ffcombine =
  let
    val decomp = flatten (calcgeomdecomp cols rows dc dr 0)
  in
    ffcombine (map ffportion decomp)
  end;

fun edgetrack (dc,dr) edgethresh anglethresh minlinelen (edges,angles) =
  let
    val ea = map2ll I edges angles
    val (cols,rows) = (length (nth(ea,0)),length ea)
  in
    geomdecomp (cols,rows,dc,dr)
               (etportion edgethresh anglethresh ea)
               (etcombine minlinelen)
  end
```

Here, an edge tracking algorithm is expressed as a general purpose geometric decomposition algorithmic skeleton. The `etportion` function is the code in Figure 11 adapted to work within a geometric boundary. A list of boundaries is generated by `calcgeomdecomp` by splitting the image into `dc` column segments and `dr` row segments. The `etcombine` function is based on the code in Figure 13 modified to use the code in Figure 14 to calculate the merge requirements followed by a `fold` operation to implement the merges.

Figure 15: A general purpose geometric decomposition skeleton

```
fun filter_timed timecnt p psize [] = ([],[],[],0,0)
  | filter_timed timecnt p psize (h::t) =
  let
    val (pval,(ut,_,_,_)) = timen timecnt p h
    val (pres,timelst,sizelst,passed,failed) = filter_timed timecnt p psize t
  in
    if pval
      then (h::pres,ut::timelst,(psize h)::sizelst,passed + 1,failed)
      else (pres,ut::timelst,(psize h)::sizelst,passed,failed + 1)
  end
```

The predicate `p` is applied `timecnt` times to each element `h` and the average user time `ut` is calculated. The `psize` function is a caller supplied function which calculates the input size of an element (the output size is fixed in a filter implementation). The result is a tuple comprising the actual result of the filter along with lists of times and sizes and the accumulated pass and fail counts.

Figure 16: Timed `filter` function

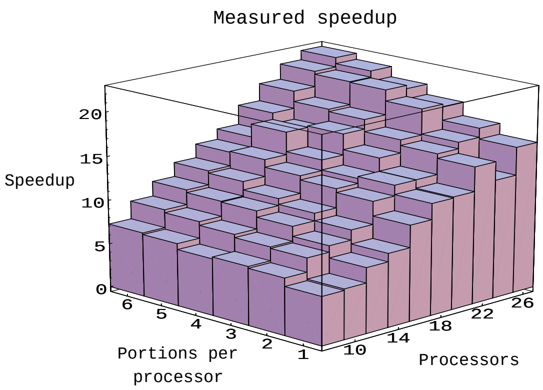


Figure 17: Measured speedup for dummy farm with parameters from prototype for tracking phase

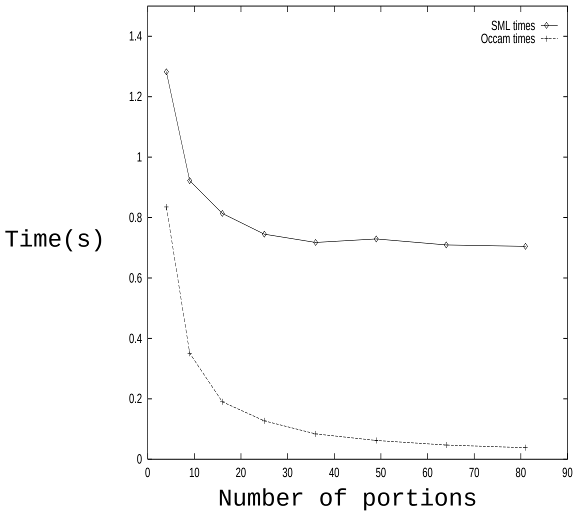


Figure 18: Correction for sequential processing in the tracking phase

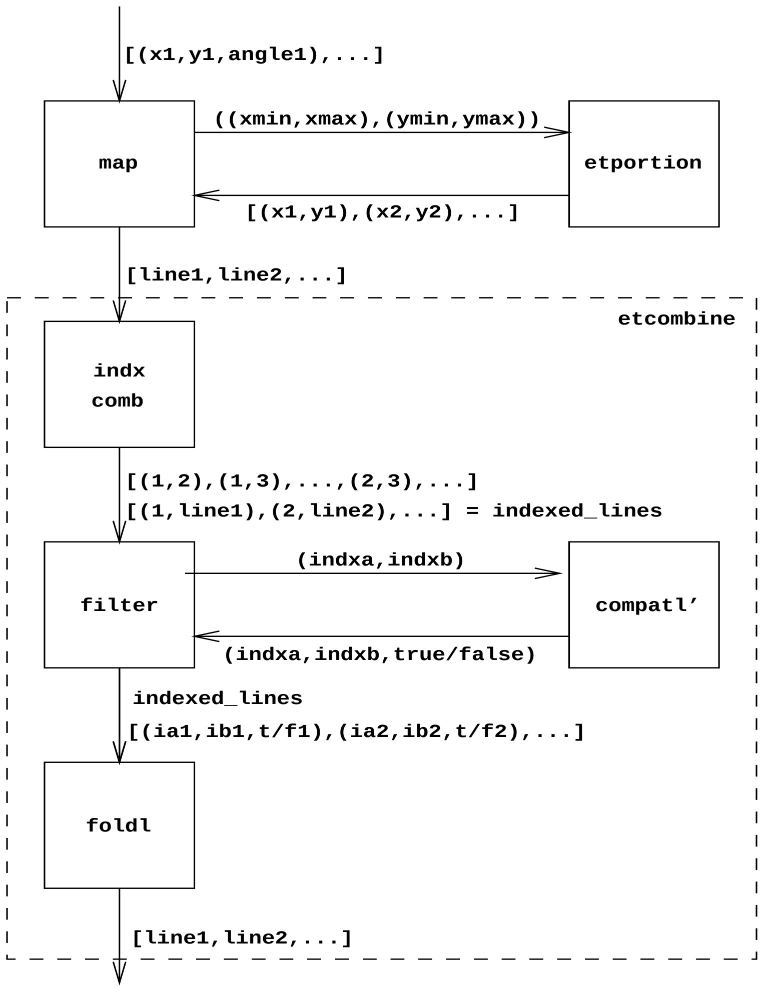


Figure 19: Structure of the parallel implementation

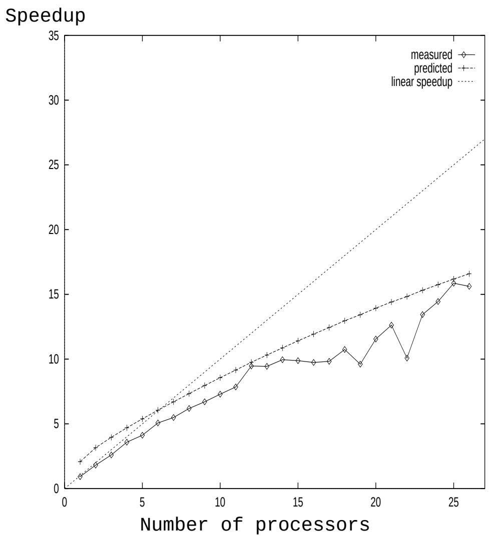


Figure 20: Comparison of predicted and measured performance for tracking phase

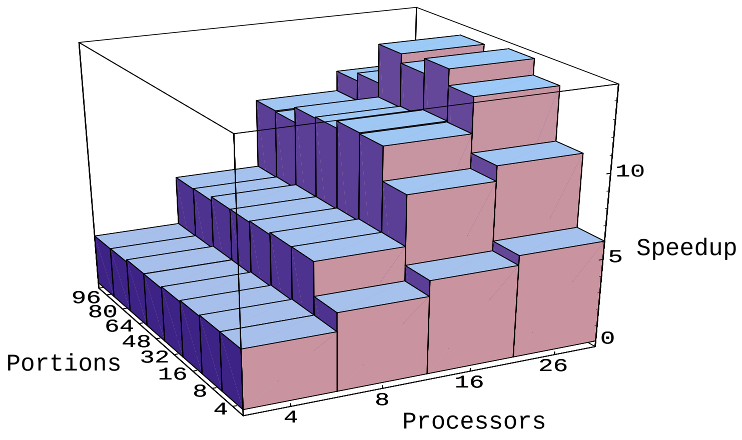
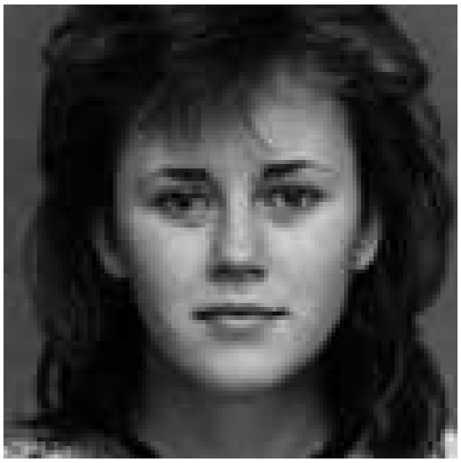
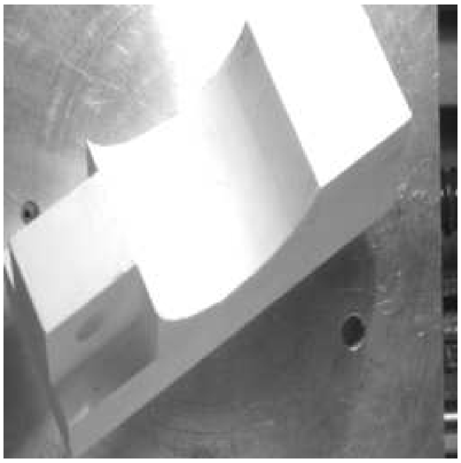


Figure 21: Performance of the occam2 edgetracking system

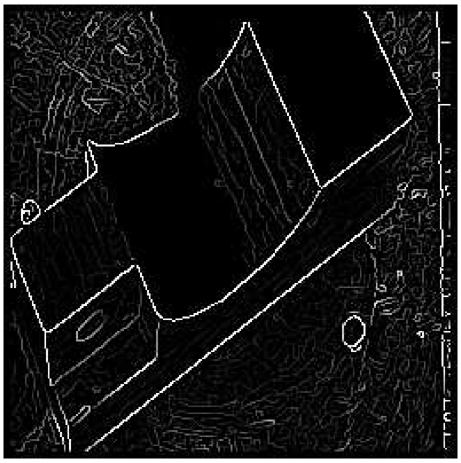


(a): intensity image of face



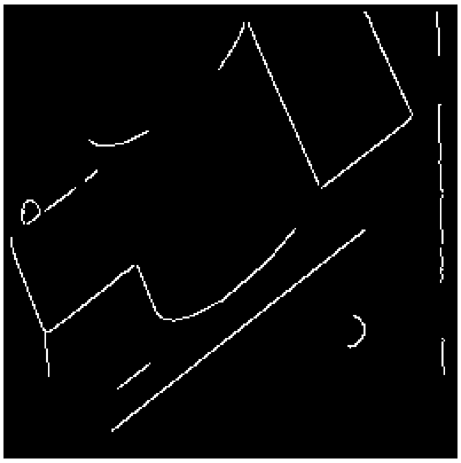
(b): intensity image of block

(c): edge data obtained for the face



(d): edge data obtained for the block

(e): tracked edges for face using the hybrid algorithm



(f): tracked edges for block using the hybrid algorithm

Figure 22: Illustrations of the edge detection and tracking process