Prototyping Parallel Algorithms using Standard ML

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

Abstract

We have been developing techniques for deriving parallel implementations of vision algorithms from prototypes written in a functional language (SML). Initially, we analysed simple, well understood algorithms to allow the prototyping methodology to be investigated in a predictable environment. Subsequently, we have extended our approach to more difficult cases such as edge tracking which present problems for parallel system development. Here we demonstrate the power and generality of our approach to parallel algorithm development by a representative set of vision algorithms encoded in SML.

1 Introduction

Functional programming is an excellent basis for *general system development* [6]. In contemporary functional languages, 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. In functional languages, 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, 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.

Functional programming is particularly useful for *parallel system development* [5]. The Church-Rosser property of evaluation order independence results in potential parallelism being found at all levels of a functional program. This abundance of parallelism may be constrained through the use of higher order functions with equivalent parameterised parallel skeletons. Hence, a functional program may be treated as a prototype for an eventual parallel implementation. Results from the instrumentation of a functional prototype based on higher order functions, combined with performance models for parallel skeletons, enables the accurate prediction of the behaviour of the equivalent parallel implementation. The transformation of functional programs based on higher order functions, for example to improve predicted performance, leads to an equivalent reconfiguration of the corresponding skeleton based parallel implementation. Thus, the theoretically based manipulation of functional programs has very useful practical implications for their parallel implementations.

We have been using functional programming in the prototyping of *vision algorithms* for MIMD parallel implementation. We have identified a small set of standard higher order functions that have efficient parallel implementations. In this paper, we develop an overview taxonomy of vision algorithms, illustrated through functional programs based on vision oriented higher order functions. These functions are themselves related to our standard functions allowing us to predict equivalent parallel implementations.

1.1 A Standard Set Of Higher Order Functions

From previous work [8, 3] we have identified a small set of useful HOFs for which we have efficient parallel implementations. These are:

1) `map` which applies a function to each element of a list and can be implemented on a processor farm or as a geometric decomposition.

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

2) `filter` which selects elements from a list according to some criterion, and can be implemented as a modified processor farm using indices to retain list order.

```
fun filter p [] = [] |
    filter p (h::t) = if p h
                      then h::filter p t
                      else filter p t;
```

3) `foldr` and `foldl` which reduce a list of values down to a single value using a combine function. These cannot be implemented directly in parallel but if the combine function is associative and commutative then the two are equivalent. An associative `fold` can be evaluated in parallel using a divide and conquer strategy. The following is `foldr`:

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

4) `compose` is a function which takes two functions and behaves as their composition (it is more usually presented as its infix equivalent `o`). This is equivalent to a pipeline of processors in a parallel system.

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

We sometimes need modified versions of these functions, for instance extending `map` to simultaneously map over two lists `map2`, or generalising `foldr` to folding over lists of lists `foldrll`. Another useful extension for vision systems is to include indices giving the coordinate of an element in the list to the function that operates on that element. For example, `foldrixyll` passes the `(x,y)` coordinate of an element in a list of lists. These modified functions are parallelised in the same way as their parent functions but may differ subtly in their performance characteristics.

2 Vision Algorithms in SML

Applications in vision systems involve a wide range of operations from low-level pixel based operations to high level operations such as model-based interpretation [2, 9]. We have successfully applied our techniques to a variety of algorithms in computer vision. Our methods are more widely applicable to programming in general but we use vision algorithms as a representative subset.

We present a set of image processing algorithms that cover the bulk of algorithm types that any developer in computer vision might be expected to encounter. We give brief examples of each algorithm in SML using the higher order function constructs that we have indicated can be implemented in parallel. Although we do not explicitly state, for each example, how the parallel implementation is derived from the prototype, we have substantial experience in this process [8, 10] using Occam2 as the target language based on a Meiko transputer surface. We have shown that if we can write a functional language program using our standard set of higher order functions then we can develop an equivalent parallel implementation based on the parallel constructs for these functions.

2.1 Image Negation: A Local Pixel Operation

Reversing the intensity of all the pixels in an image is a trivial operation but is one example of many similar pixelwise operations that are generally useful in image processing. For instance, many intensity image processing routes involve global thresholding of the data or conversion from one type of pixel representation into another; an example of this might be to convert a floating point format image into a coarsely quantised byte format image.

The natural SML implementation of this is the application of nested `map` operations to a `list list` representation of the image:

```
fun local_pixel ff image = map (map ff) image

fun negate x = (255 - x)
fun quantise_pixel (x:real) =
  floor (((x - minx) / (maxx - minx)) * 255.0)

val negative = local_pixel negate
val quantise = local_pixel quantise_pixel
```

Here, we are creating a higher order function `local_pixel` which encapsulates the structure of the operation we want to perform but not the details of the calculations. We then define the particular operations we want to perform on a given pixel, `negate` and `quantise_pixel`, and then instantiate our higher order function to create the desired operations.

This is a good example of code development in SML. One is always looking for abstractions, ie. commonality between different parts of the code to maximise the recognised benefits of code reuse such as reducing the amount of time spent debugging.

2.2 Edge Detection: Local Window Operations

Another fundamental set of algorithms with many applications in vision systems are local window operations. Here, the region surrounding each pixel, usually a square neighbourhood, is processed to give a new value at that pixel. These include gradient calculations, filtering and smoothing functions such as Gaussian filtering, pattern matching and local function fitting routines.

A large subclass of these algorithms concerns the 2D convolution operation in which a mask is multiplied and summed with the region surrounding each pixel:-
$f(x,y) \ * \ g(x,y) \ = \ \sum_m \sum_n f(m,n).g(x \ - \ m, y \ - \ n)$

This operation is extensively used in implementations of the Canny edge detector [4], an edge enhancement regime that includes Gaussian filtering and edge thinning giving good localisation of edges. The Canny edge detector involves several phases; for clarity, we illustrate the process in SML using the Sobel edge detector. This is a double convolution, calculating the gradient in the X and Y directions separately, combining the two into a single edge strength value which can then be thresholded to localise the edges:

```
(* Local window HOF *)
fun rmap f [] = []
  | rmap f l = (f l)::(rmap f (tl l))

fun cmap f [] = []
  | cmap f ([]::t) = []
  | cmap f l = (f l)::(cmap f (map tl l))

fun local_window ff img = rmap (cmap ff) img

(* Convolution operators *)
fun convol2 fc ff def fltX fltY img =
  fc (foldr2ll ff def fltX img) (foldr2ll ff def fltY img)

(* Define Sobel edge detector *)
val sobelY = [[1,0,~1],[2,0,~2],[1,0,~1]]
val sobelX = [[~1,~2,~1],[0,0,0],[1,2,1]]

fun addmul sofar v1 v2 = sofar + (v1 * v2) : int
fun euclidean x y =
  floor (sqrt (((real x) * (real x)) + ((real y) * (real y))))
val sobeloper = convol2 euclidean addmul 0 sobelX sobelY

val sobel = local_window sobeloper
```

The procedure is the same as before. We develop a HOF which has the basic, top level structure of the local window operations we want to perform (`local_window`). We then define the specific operations that make up our intended operator (`sobeloper`) and instantiate our HOF with those functions. In this case, there is more structure to the code, in that the operator itself is composed of HOFs, (`convol2`), which are instantiated with specific code to perform summations (`addmul`) and combinations (`euclidean`).

This is a completely general local window operation function. Although the Sobel edge detector is a single operation based on combining two convolutions, we can define any similar operation such as Gaussian filtering or the Canny edge detector by instantiating with the relevant operators and composing successive operations together.

2.3 Hough Transform: Global Image Transformation

The Hough transform is a technique for detecting patterns in data by accumulating points which are characteristic of a given pattern [7]. This is one example of a range of transforms that may be performed on images, another example is the discrete Fourier transform. In these transformations the entire input image may be required to calculate one value in the output transformation space.

We have studied the Hough transform and have implemented a straight line detector in SML and Occam2 [8]. This code works by mapping the transformation equation as a function over the image giving a transformed image of tuples consisting of the (x, y) co-ordinates, the co-ordinates of the point in parameter space (ρ, θ) and the edge strength value. The indexing of these tuples could be transformed into an iconic representation of parameter space by accumulating values in a (ρ, θ) indexed image. To find lines in the image, the transformed image is converted into a histogram by accumulating edge strength values against (ρ, θ) values, the resulting histogram can be thresholded to give a crude list of peaks in parameter space:

```
(* Transform function: rho = x * cos(theta) + y * sin(theta) *)
fun transformfn low rinc tinc ymax (x,y,edgeval,angleval) =
  let
    val yy = (ymax - 1) - y
    val ri = real rinc
    val theta = real(angleval) * 360.0 / 255.0
    val beta = (90.0 - theta) * pi / 180.0
    val rho = real(x) * cos(beta) + real(yy) * sin(beta)
    val thetastep = floor(theta * (real tinc) / 360.0 + 0.5)
    val rhostep = floor(((rho * ri / 720.0) + ri / 2.0) + 0.5)
  in
    (x,y,thetastep,rhostep,edgeval)
  end

(* Transform image into parametric form *)
val tfn = transformfn low rhoinc angleinc (length edges)
val trans_data = map2ixyll tfn 0 0 edges angles

(* Generate histogram in hough space *)
fun tohistfn low hist (x,y,thetastep,rhostep,edgeval) =
  if edgeval >= low
    then addHist (thetastep,rhostep) (real edgeval) (x,y) hist
    else hist
val histogram = foldrll (tohistfn low) emptyHist trans_data

(* Threshold peaks *)
fun peakthfn peakth ((t,r),wgt,c) = wgt > (real peakth)
val peaks = filter (peakthfn peakth) (showHist histogram)
```

2.4 Edge Tracking: Image to Feature Transformation

Unlike the previous examples, edge tracking is representative of a class of visual algorithms which determine features by growth from an initial seed region (regional segmentation often uses this approach) and has *not* proved easy to express in parallel form. Hence we shall treat this case in more detail.

Edge tracking [1] involves determining seed points in an edge enhanced image and following lines or curves by examining neighbourhoods for compatible pixels. Compatible neighbours then become candidates for further growing and the process continues until no more compatible neighbours can be found.

Processing an edge tracking algorithm requires access to the image in a manner which precludes regular row-column scanning. This makes list based processing highly inefficient compared with array based implementations. It is still possible, however, to develop edge tracking algorithms using SML and, potentially, to infer more efficient parallel implementations from the SML prototype.

We have investigated three implementations of an edge tracking algorithm which works on the edge strength and orientation maps produced by the Canny edge detector. We have working SML for these algorithms from which parallel implementations can be deduced, although we do not have working parallel code. However, if we can express an edge tracking algorithm using our standard set of higher order functions we stand a very good chance of being able to generate a parallel implementation.

2.4.1 List Based Processing of Iconic Data

An intuitive *imperative* expression of the edge tracking process has been directly encoded in SML:

```
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) = foldr2ixyll et (0,0) ([],[]) ea
```

The basic structure of the algorithm is a `fold` operation over the entire image, each point in the image is treated as 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.

The main problem with this program is that the `getedge` and `getangle` operations work by scanning the image from $(0,0)$ each time, for instance to access pixel (x,y) in image ea the construct `nth (nth (ea,y),x)` is used. This is a fundamental property of the way the data is organised into lists of lists. Having such a high complexity operation at the very lowest level of the algorithm causes the prototype to be very inefficient. This increases the development time for the prototype and accentuates the differences between list based processing in the prototype and array based processing in the parallel code.

2.4.2 Array Based Processing of Iconic Data

The above algorithm is, however, 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 manipulations. The array `a2img:(int * int) Array2.array2` is created by converting the list based image into an array based image:

```
(* 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)
```

One point to note about working with arrays in SML is that, 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 the source of the error than for an equivalent list based program. This version of the program, however, is likely to behave much more like a parallel implementation that uses arrays so the lack of functional purity could be tolerated. Overall, however, we lose some of the flexibility and power of our methodology.

2.4.3 List Based Processing of Sparse Point Lists

The final version of the edge tracking algorithm is based on an alternative description which is much more declarative than the previous one. This 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.

This algorithm works by starting from a list of singleton lines derived by thresholding the image. Each successive iteration tries to merge as many of these lines as possible and when a stable set of lines is reached 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).

Again, this algorithm can be directly encoded in SML:

```
fun genlist (sofar,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 (sofar,ll') l' =
      if compatl' l l'
        then (true,(l' @ l)::ll')
        else (sofar,l'::ll')

    val (added,newll) = foldl (addlist l) (false,[]) ll

  in
    (sofar orelse added,if added then newll else l::ll)
  end

fun iterate (false,LL) = LL
  | iterate (true,LL) = iterate (foldl genlist (false,[]) LL)

fun edgetrack img = iterate (true,img)
```

The scanning of point and line lists is carried out by a series of nested `fold` operations and the overall convergence is controlled by the `iterate` function. This program is much more succinct than the previous versions showing the declarative nature of the algorithm.

This version is also much more efficient than the previous list based code. Although the algorithm is iterative it turns out that the bulk of the merging is carried out by the first iteration and very few iterations are required after that.

2.5 Maximal Cliques Algorithm: Interpretation Using Non-iconic Structures

There are a number of ways of matching features extracted from an image with features in the objects expected to be found in such images. These include interpretation tree search, graph matching and relaxation labelling.

These algorithms all have characteristic data structures, such as trees or graphs and have to be considered on their own merits. SML is highly capable of handling these structures in an efficient manner. We illustrate this with an example from graph theory, that of identifying objects in a labelled image using a maximal cliques algorithm applied to an association graph [2].

This method is itself well-known, and, in fact, the SML implementation is virtually this algorithm written in SML statements, no coding is required to turn the declarative description into a workable solution. Another point about this algorithm is the use of set based operations. It is easy to define an abstract data type to handle sets in SML, hence the ease of implementing the algorithm. This, however, demonstrates an alternative approach to parallel system building whereby the set abstract data type operations could be implemented in parallel, roughly corresponding to our prototyping methodology but in a situation where the prototype information can be reused without further effort. A cliques finding algorithm is as follows:

```
fun connected ((n1,a1):node) ((n2,a2):node) = (member n1 a2) orelse (member n2 a1)
fun iscon (n1:node) sofar (n2:node) = sofar andalso (connected n1 n2)
fun allcon (s:node SET) (n:node) = setfold (iscon n) true (remove n s)

fun allcliques N =
  let

    fun cliques X Y =
      let
        val y = setfilter (allcon X) (difference Y X)
      in
        if isemptyset y
          then [X]
          else (cliques (union (sethd y) X) Y)
             @ (cliques X (difference Y (sethd y)))
      end

  in
    cliques emptynodeset N
  end
```

3 Conclusions

The five examples we have presented cover low, intermediate and high-level vision tasks. There are other algorithm types that we have not discussed, such as geometric transformation of images (eg. rotation, scaling and reflection) and statistical summaries of images (eg. histograms). There are also other algorithm subclasses within the five types of algorithms, (for example, interpretation tree searches as opposed to graph matching). However, we have tackled a sufficient number of different cases to demonstrate that our methodology is flexible and powerful enough to be considered a general tool for developing parallel vision systems.

As a rule of thumb, if it is difficult to express an algorithm in a functional language then it is usually difficult to implement in parallel as well, this is exemplified by the edge tracking algorithm we discussed. We have achieved respectable parallel performance for hand translations of functional vision algorithms [8] and have developed a general purpose parallelising compiler to automate the translation process [3] which has shown promising results with simple programs.

Acknowledgements

We would like to acknowledge our colleagues in the vision group for their assistance. We are grateful to the EPSRC for funding this work with grant GR/J07884.

References

[1] G. P. Ashkar and J. W. Modestino. The Contour Extraction Problem with Biomedical Applications. *Computer Graphics and Image Processing*, 7:331–355, 1978.

[2] D. H. Ballard and C. M. Brown. *Computer vision.* Prentice-Hall, 1982.

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

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

[5] 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.

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

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

[8] G. J. Michaelson and N. R. Scaife. Prototyping a parallel vision system in Standard ML. *Journal of Functional Programming*, To Appear, 1995.

[9] A. Rosenfeld and A. C. Kak. *Digital picture processing*, volume 1-2. Academic P., 2nd edition, 1982.

[10] 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.