

A FUNCTIONAL METHODOLOGY FOR PARALLEL IMAGE PROCESSING DEVELOPMENT

D J Johnston, M Fleury, A C Downton
University of Essex, UK

1 INTRODUCTION

Parallel Image Processing has been a topic of interest for many years, but has not yet delivered a design methodology which can maintain portability across rapidly changing parallel implementation technologies. The two hurdles to be overcome are the difficulties of re-arranging application code into a parallel form and the lack of a general and universally supported parallel machine model. This paper addresses these problems using a functional language transformation, illustrated in the context of a particular application. We show that the overhead introduced by the transformation is relatively small, but the benefit derived is substantial, since the functional programming discipline enforces an implementation-independent definition of core parallel requirements, which can then be mapped onto a broad set of parallel architectures, ranging from shared and distributed memory conventional multiprocessors to direct hardware implementations constructed using silicon compilers and FPGA technology.

2 ML and Concurrent ML

Figure 1 shows the software development flowchart for the methodology. Conventional parallel development involves taking a serial application (A_{C++}) and converting it into a form (A'_{C++}) which interfaces with a parallel harness (H_{C++}): software which controls the runtime execution on a parallel architecture. This is a difficult process because the harness is normally inflexible third-party software which was abstracted with an imperative language mind-set. The proposed methodology takes a sequential application expressed in the functional language ML (A_{ML}); ports this to a parallel form (A'_{CML}) in the language CML (this is ML with a parallel library designed by Reppy [4] based on the CSP model due to Hoare [1]); and then finally abstracts this into both a parallel harness in CML (H_{CML}) and a sequential application in ML (A''_{ML}).

Although the overall development methodology may seem more complex, the application developer is only asked to perform the transform $A_{ML} \rightarrow A''_{ML}$, which by careful choice of the parallel abstractions is actually a very minimal change, unlike the equivalent conventional parallelisation in C++. All other transformations only need to be implemented once as part of developing the methodology. Ultimately, the user still has to translate

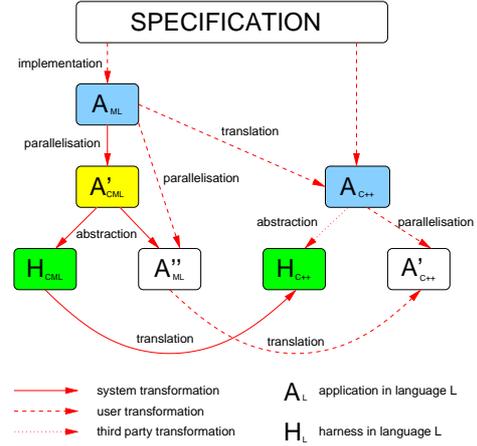


Figure 1. intermediate ML development

ML software to C++ (this stage has not yet been automated, though previous work shows it is possible [5]), and this is done at the end of the development cycle. Translating at the earlier stage is pure sequential ML prototyping, though it is then also possible to parallelise within C++ using the proposed method by interfacing to the harness C++ API which is the same as the ML API. For C++ coders, the method introduces a new H_{C++} module, but one providing flexibility and composability, deriving from the functional way it was conceived.

3 SIMPLE EXAMPLE

Before introducing the application under discussion, it is useful to illustrate the functional style by the tractable and **complete** expression of a convolution algorithm.

```
fun convolve2D (src, filter) =
let
  (* get sizes of src image and filter *)
  val (sh,sw) = dimensions src;
  val (fh,fw) = dimensions filter;

  (* produces a single element of convolution matrix *)
  fun single_convolve (r0,c0) =
  let
    (* multiply corresponding filter and src elements *)
    (* offset into src is (r0,c0) *)
    val product = tabulate
      (fh,fw,fn(i,j) => sub(src,i+r0,j+c0)
                          * sub(filter,i,j) );
  in
    (* add together elements of product matrix *)
    (* reduction/fold operation is add *)
    fold ( fn (a,b) => a + b ) 0.0 product;
  end;
in
  (* result is size of src minus size of filter *)
  tabulate ( (sh-fh)+1, (sw-fw)+1, single_convolve );
end;
```

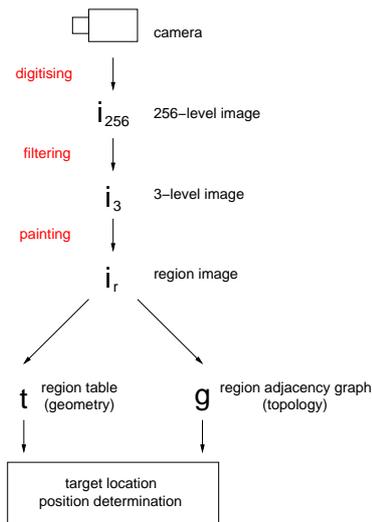


Figure 2. serial image processing

In (C)ML, functions can be manipulated with the facility of data, so a result matrix is specified solely by its dimensions and a rule (or function) to produce each element. These are passed to a higher-order function `tabulate`.

Traditionally the low execution performance of ML has relegated it from the development of “real” applications. However, with advances in software and hardware even compute intensive operations, such as this convolution, are now becoming practical under ML.

4 PAINTER’S ALGORITHM

Position determination within an Augmented Reality system is a typical combined image processing and computer vision application [2]. Targets are identified within a video stream, by decomposing each frame into regions using the “Painter’s Algorithm”. This does not fit comfortably into existing parallel paradigms and so presents a realistic challenge for parallel image processing.

The target identification image processing pipeline is shown in Figure 2. The captured 256-level greyscale image (Figure 3) is filtered to a 3-level greyscale image (Figure 4) to enhance the visibility of regions within black and white targets. Each distinct connected region within the 3-level greyscale image is then painted a unique colour (Figure 5). Indeed this colouring algorithm, the “Painter’s Algorithm”, is the means by which distinct regions are computed. Eventually this image information is distilled into a Region Adjacency Graph (RAG) which holds topological information of region connectivity and a region table which contains geometric information, such as the centroid, for each region in the graph. The RAG and table are used to identify the targets. Then from the known locations of these targets in the real world, it is possible to work out the position of the camera (for fixed targets) or target-tagged objects

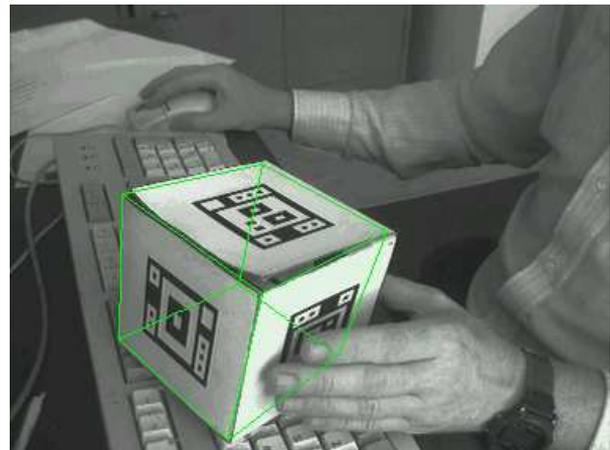


Figure 3. i_{256}



Figure 4. i_3

(for fixed camera). The super-position of a wireframe cube in Figure 3 indicates that the location of the cube has been successful.

Consider the conceptual parallelisation in Figure 6. For simplicity, it is assumed that the processing is split geometrically into just two parts: a right-hand and a left-hand half image. As the corresponding 3-level greyscale images are produced by adaptive spatial filtering, edges of the input 256-level greyscale half-images have to be swapped at their join. The Painter’s Algorithm can be run independently on each half to produce the conventional serial output of a table and graph. (The parallelisation of either the target detection or subsequent position determination is not addressed. However, these only constitute around 2% of the overall processing time.)

How are the right and left RAGs (g_R and g_L respectively) and the right and left tables (t_R and t_L respectively) joined? This requires further information describing the region connectivity across the split, which is derived from the line of pixels along each side of the split from the $(i_3)_L$, $(i_3)_R$, $(i_r)_L$ and $(i_r)_R$ images.

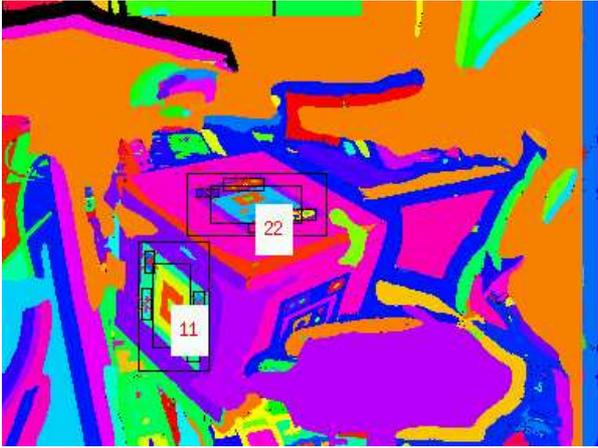


Figure 5. i_r

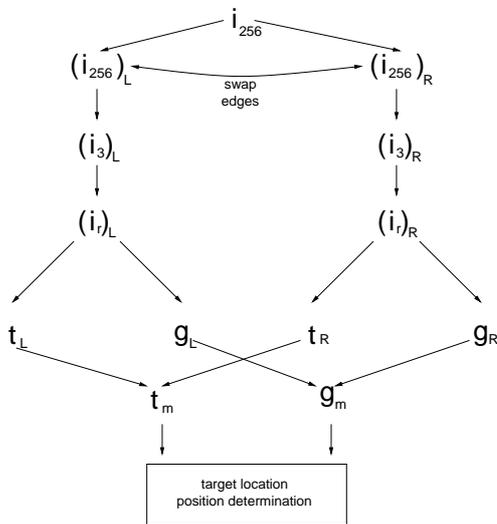


Figure 6. parallel image processing

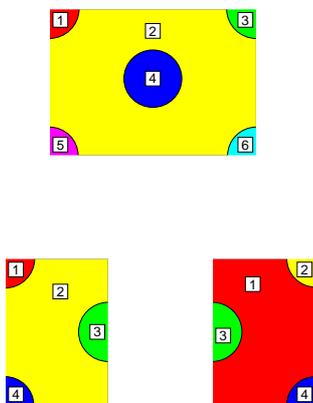


Figure 7. geometric subdivision of the Painter's Algorithm

To appreciate the problem consider the painted image and the corresponding two painted half-images in Figure 7. It can be seen that the painting (or region numbering) occurs scanning top-left to bottom-right. We cannot put the two painted half images together and get automatic agreement on region numbering. Instances of all possible cases occur:

- the same region numbered differently
 L_2 and $R_1 \rightarrow$ wrong
- different regions numbered the same
 L_4 and $R_4 \rightarrow$ wrong
- the same region numbered the same
 L_3 and $R_3 \rightarrow$ coincidentally consistent
- different regions numbered differently
 L_1 and $R_2 \rightarrow$ coincidentally consistent

The parallel algorithm to reconcile region numbering is similar to the serial Painter's Algorithm. In fact, only the numbering in one half image has to be changed - and indeed only the pixels at the boundary of this half-image have to be physically repainted for the algorithm to work. Note that an image is not part of the original serial output, but partial images (edges) need to form part of the output of the parallel processing for edge-data accumulation. These are calculated anyhow (as intermediate results) so this is no great hardship. We can categorise the serial and parallel processing by their CML type signatures:

```
type serial_input      = image;
type serial_output    = graph * table;
type serial_process   =
  serial_input -> serial_output;
```

```
type parallel_input   = image;
type parallel_output  = graph * table * edge * edge;
type parallel_process =
  parallel_input -> parallel_output;
```

What further infrastructure is required to run the application in parallel? We require functions to split the input and combine the output:

```
type split_input      =
  parallel_input -> parallel_input * parallel_input;

type combine_output   =
  parallel_output * parallel_output -> parallel_output;
```

Splitting a geometric structure such as an image is sufficiently regular, that a function matching `split_input` does not have to be supplied by the programmer. Interestingly, the final output from the parallel and serial implementations of the Painter's Algorithm is different in general, but they will be equivalent up to a re-ordering.

The remaining concern is how to swap the edge data for initial filtering? A compromise was reached that though this communication has still to be explicitly actioned by

