Parallel architectures for image processing

by A. Downton and D. Crookes

Image processing is often considered a good candidate for the application of parallel processing because of the large volumes of data and the complex algorithms commonly encountered. This paper presents a tutorial introduction to the field of parallel image processing. After introducing the classes of parallel processing a brief review of architectures for parallel image processing is presented. Software design for low-level image processing and parallelism in high-level image processing are discussed and an application of parallel processing to handwritten postcode recognition is described. The paper concludes with a look at future technology and market trends.

1 Introduction

Background

Parallel processing has been proposed as a solution to computationally demanding problems for many years, but despite extensive research its general application still appears to be a distant prospect. Nevertheless, in certain areas it has established an effective niche, including for example weather forecasting, finite element analysis, 'grand challenge' problems in applied physics, discrete-event simulation and large-database systems.

Image processing is also often considered a good candidate for parallelism because of the large volumes of data and the complex algorithms commonly encountered in image processing applications; this view was encouraged by the introduction in the mid 1980s of the Inmos transputer, a seemingly ideal component for parallel embedded systems. Subsequent experience has shown, however, that building parallel hardware is less difficult than providing attractive parallel software tools which utilise the hardware efficiently. This paper presents a tutorial introduction to the field of parallel image processing, based in part upon two projects which are part of EPSRC's* directed programme on Portable Software Tools for Parallel Architectures (PSTPA). More comprehensive coverage of practical parallel processing for those wishing to explore the field is given by Chalmers and Tidmus[1], while Pitas[2] focuses particularly on parallel algorithms for image processing and computer vision.

Flynn's classification

In 1996, Flynn proposed a simple categorisation of computer which is still useful today[3] (Fig. 1). This taxonomises parallel computers according to two dimensions of parallelism: the instruction stream and the data stream. Thus, while single instruction stream, single data stream (SISD) represents the conventional 'von Neumann' computer architecture, two other categories, SIMD (single instruction stream, multiple data stream) and MIMD (multiple instruction stream, multiple data stream) have also been extensively investigated. The fourth category, MISD (multiple instruction stream, single data stream), is sometimes disregarded, although in fact it could be considered to be an appropriate description of some of the systems described in Section 4 of this paper.

SIMD specifies a style of processing which operates on vectors of data. All execution units are synchronised and operate in lock-step in response to instructions from a single instruction stream. This approach has often been seen as well-suited to low-level image processing, where each pixel of an image could potentially be operated on by a separate processing element. In contrast, MIMD defines an architecture consisting of multiple independent processors, each operating on its own, potentially independent, data. This approach is more suited to higher-level image analysis and computer vision operations where complex multicomponent algorithms are applied to

	Single data	Multiple data
Single instruction	SISD	SIMD
Multiple instruction	MISD	MIMD

Fig. 1 Flynn's classification of parallel processing

*EPSRC, the UK Engineering and Physical Sciences Research Council

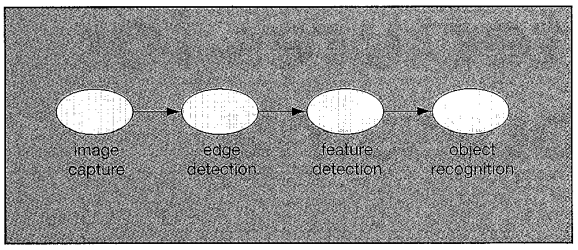


Fig. 2 Pipeline parallelism

abstract data structures. This dichotomy immediately highlights a central problem in parallel image processing, the fact that no single architectural approach is well suited to all applications, or even all components of a single application.

MIMD machines can be further subdivided into shared-memory (bus-based) systems (SM-MIMD) and distributed-memory systems (DM-MIMD) interconnected via some other network of connections, which may be a set of point-to-point processor connections or (more commonly in recent machines) a software-controlled switching network. SM-MIMD machines are widely used as high-performance multitasking computers, but the bottleneck of processor-memory bandwidth limits their scaleability to only a few processors. In contrast, it is relatively easy to build DM-MIMD machines with hundreds of processors, but much more difficult to program them to utilise all the processors effectively.

Forms of parallelism for image processing

There are two common approaches to applying parallelism to image processing applications: *pipeline* parallelism and *data* parallelism.

Pipeline parallelism: Since a complete image processing system usually requires a sequence of different tasks to be performed on an image, it is sometimes possible for these tasks to run concurrently on different processors (Fig. 2). For instance, if a sequence of images is being processed, then, once one task has processed its first image, it can pass the intermediate result on to the next task and immediately take in the next image in the sequence and start processing that. In fact, even when processing a single image, the system can sometimes be pipelined so that tasks further down the line can receive and begin processing part of the image before previous tasks have completed processing of the rest of the image. This MIMD parallelism is sometimes called pipeline parallelism. It suffers, however, from two main disadvantages. Firstly, as with all pipelined systems, the rate of throughput is dictated by the speed of the *slowest* task in the pipeline. Secondly, the software is not *scaleable*, since it is usually impossible to decompose a problem into an arbitrary number of concurrent tasks.

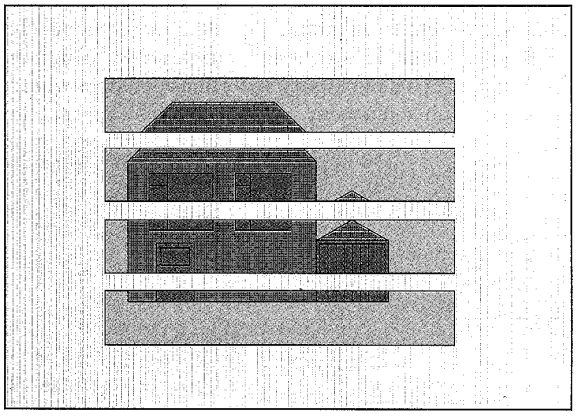


Fig. 3 Division of data using static parallelism

Data parallelism: A more general-purpose (and generally more effective) way to parallelise low-level image processing operations is to partition an image into sections, distribute these sections across a network of processors and have all processors carry out the same task on their own section of the image. This is an application of data (SIMD) parallelism, though it is often implemented on a MIMD network of processors. For low-level image processing, where pixel-level and neighbourhood operations predominate, the operations are very localised, and so the processors can usually operate more or less independently and in parallel.

These two forms of parallelism can often be conveniently combined, by first partitioning the complete image processing application into a pipeline of concurrent stages and then applying data parallelism within each stage to balance the pipeline. Both aspects of this process are areas of current research in parallel image processing. Software tools for exploiting data parallelism automatically at the component algorithm level are discussed in Section 3, and a top-down design approach for integrating these components using the pipeline model is presented in Section 4.

If data parallelism is to be used, and image data is to be processed across a set of processors, then a follow-on decision which needs to be made is how the image data is to be distributed, and where it is to be stored. There are basically two options: static distribution and dynamic distribution.

Static against dynamic parallelism

Static data distribution: Here, an image is partitioned into an appropriate number of segments and distributed one segment per processor, as illustrated in Fig. 3 for the case of four processors. All processors maintain and process their own (static) segment of the image simultaneously.

This is probably the simplest approach to data parallelism, and it can be very satisfactory in many cases. Processing can often proceed independently on each processor, with very little communication. A slight problem arises when performing a neighbourhood operation at the edge of a segment — say, the bottom edge of the top segment above. For a pixel on this edge, the neighbourhood appears to straddle two processors (top and next to top). To overcome this problem efficiently, the usual technique is to duplicate segment edges on both processors. If the neighbourhoods being used are never more than 3×3, then a one-pixel-wide border extension is all that is necessary.

However, problems can arise with this static approach when the *information* content of the image (e.g. the edge

information) is unevenly distributed throughout the image. More generally, when working in parallel with features of an image (e.g. lines or regions) rather than just pixels, the problem of *load balancing* is frequently encountered. This is because, although pixels are evenly distributed in an image, the information content (and hence the work load) is usually unevenly distributed. Attempting to distribute the image in a way which balances the work load is called load balancing. The static distribution strategy outlined above is susceptible to load balancing problems, and for this reason it may be worth considering the alternative of a more dynamic approach.

Dynamic data distribution — farming: The method of processing an image by farming involves holding the image on a controlling processor, dividing it up into small packets, and 'farming out' the processing of these work packets to a pool of worker processors. The worker processors return result packets and wait for another work packet to process (Fig. 4).

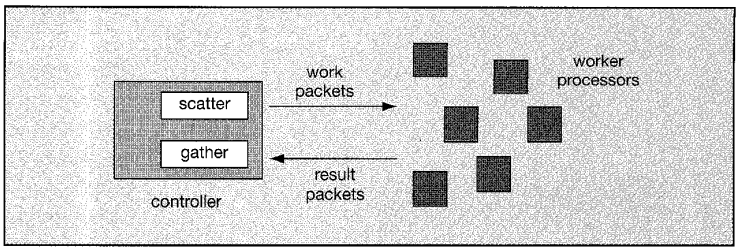


Fig. 4 Dynamic load balancing using processor farming

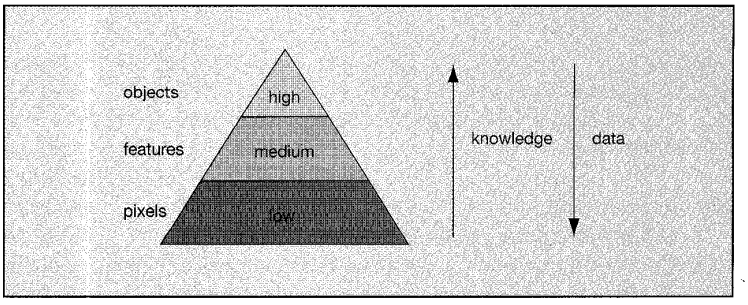


Fig. 5 The image processing pyramid

If a worker receives a work packet which requires very little computation, it will quickly be available to receive another; a 'heavy' work packet, however, will keep a processor busy for longer, and so it will not process as many packets. This flexibility enables better load balancing to be achieved when the work load is unevenly distributed throughout the image.

Although providing better load balancing, the farming approach also has drawbacks. A potential problem is the amount of communication involved. Also, careful scheduling is necessary to achieve optimised dynamic behaviour[4]. If the computational effort involved in the entire operation is relatively small, then the overheads in farming out the work packets and receiving the result packets can considerably outweigh the saving from improved load balancing. The choice between a static and dynamic distribution depends on the amount of computation involved in processing a work packet, as well as the unevenness of the work load.

2 Architectures for parallel image processing

The image processing hierarchy

The range of typical operations presented above can be classified into three broad categories, or *levels*, which are generally taken to be as follows:

- *low-level* operations. These take a pixel image and generate another pixel image. Examples include contrast adjustment and edge detection. Data is typically geometric, regular and mainly local.
- *medium-level* operations. These take a pixel image and generate a set of features, such as a set of lines or regions. Feature data is usually symbolic and non-local.
- *high-level* operations. These take a set of features and generate a set of objects. Data is symbolic and complex, and processing is often model-driven.

Because the amount of raw data decreases as we move up the hierarchy of levels (from a large number of pixels at the low level to a small number of objects at the high level), the relationship between the levels is often presented as a pyramid (Fig. 5).

The high computational requirements of image processing applications has generated many high-performance architectures specifically tailored to the image processing pyramid. These have typically incorporated parallel processing in some form or another. Some of the more common classes of architecture are summarised below.

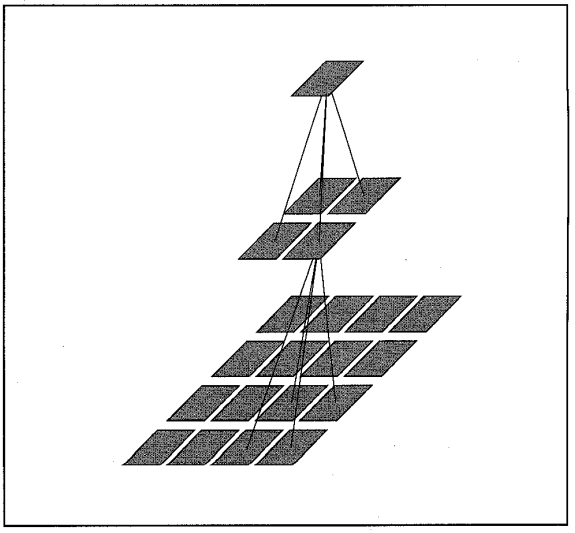


Fig. 6 Pyramid processor architecture for image processing

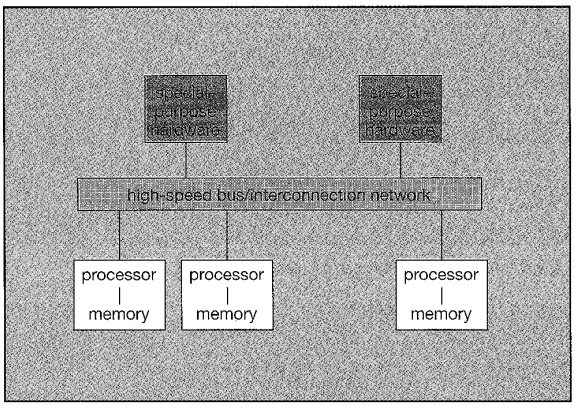


Fig. 7 Bus-based shared-memory MIMD architecture for image processing

Pyramid architectures

Pyramid architectures attempted directly to match the image processing pyramid by providing each level with its own set of processors, and matching the number of parallel processors with the level (Fig. 6). However, it transpired that mesh parallelism at the medium and high levels was not flexible enough, so these machines were not commercially successful. Examples can be found in References 5 and 6.

Bus-based architectures

A more flexible arrangement, which can include special-purpose hardware for common low-level operations (e.g. convolver chips) and multiple processors possibly with virtual shared memory for flexible communications at the higher levels is based upon the shared-memory MIMD model (Fig. 7). For real-time processing, versions of this type of architecture have been successful in practice[7].

Finally, in recent years, there has been much interest in using a cluster of standard workstations connected by standard network technology as a way of realising a processor array. Although this approach is very useful for proof of concept, there are problems in trying to achieve real-time performance[8].

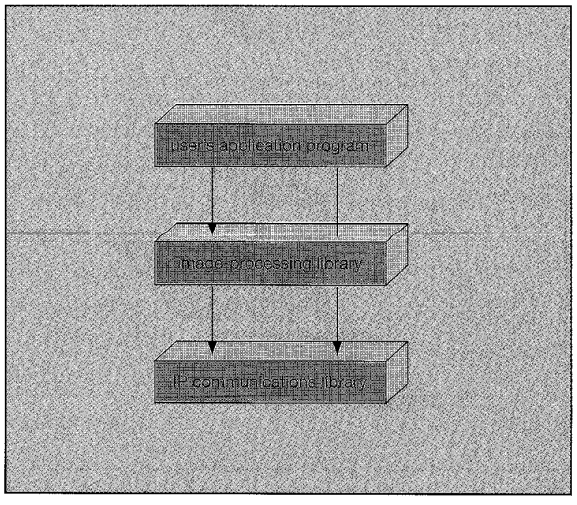


Fig. 8 Layered image-processing software model

3 Software for low-level parallel image processing

Issues and goals in software design

The very fact that writing parallel software is a difficult task means that any parallel software is a valuable resource. It is not economic to rewrite a whole program just because a few more processors need to be added to try to speed up performance. To retain investment in software over a long period of time, there are at least three factors which can change and which must be allowed for in designing maintainable software:

- *Scaleability:* varying the number of processors in an existing processor configuration.
- *Portability.* Given the rate of developments in parallel computing, it is likely that any parallel software will need to run on more than one parallel computer. As much of the software as possible should be written to be independent of the hardware.
- *Extensibility.* More functionality will be added to the software over a period of time. It should be as easy as possible to add new operations.

This Section presents a simple design approach for achieving these goals, and outlines how to implement parallel image processing operations. It is presented as a tutorial starting-point rather than a definitive methodology.

A layered software model

A standard way of providing a hardware-independent interface to application programmers is to provide a library of routines which hides all (or most) of the architectural details from the programmer. This principle can also be applied in implementing the library itself, giving a layered model of the whole system (Fig. 8).

The image processing library

Programming languages have been developed which facilitate data parallel programming, including HPF (High Performance Fortran). In HPF, the programmer is responsible for specifying the data distribution strategy for declared arrays and the compiler will do the parallelisation of statements which act on distributed arrays. There are also other general-purpose parallel programming models, including BSP (Bulk Synchronous Parallelism)[9]. However, one of the purposes of this paper is to demonstrate how the parallelism is exploited (even if these details are subsequently hidden by standard software engineering methods). Therefore we will assume a very simple programming model comprising a standard programming language (such as 'C') with a message-passing communication shell.

A surprising range of image processing operations can be expressed with a relatively small number of routines if the library is written in a language which supports the passing of functions as parameters (such as 'C'), and if the operations are based on the basic operators of Image Algebra[10,11]. Here, a key data structure is the *template*, which represents the usual concept of a moving, weighted

window in neighbourhood operations.

A useful software library will have at least the following sections:

Creation and storage management for data objects. There are three special data types:

- *Images.* These are either distributed statically or held on a controller, ready for dynamic distribution.
- *Vectors.* These 1-D arrays for histograms, lookup tables, etc. are replicated on each processor.
- *Templates.* These are replicated on each processor.

Object management operations. The library should firstly contain routines for creating and managing storage for these basic objects. These routines are typically:

```
create_image (Image, Num_rows, Num_cols)
create_vector (Vector, Size)
create_template (Template, ...template details (size,
                                        weights, etc.)...)
release_image (Image)
release_vector (Vector)
release_template (Template)
```

Pixel-level operations

```
image_image (Dest_image, Source_image1, Op,
                                    Source_image2)
```

where 'Op' is a function which takes two source pixels and returns a result pixel.

```
image_scalar (Dest_image, Source_image, Op,
                                    scalar_value)
```

Typical functions for Op might be: the usual arithmetic operations (plus, minus, times, divide), logical and comparison operations, and other more complex user-defined operations. In practice, it is wise to have individual, specific `image_image` routines for the more common arithmetic operations, to avoid the overheads of function calling.

Neighbourhood-level operations. There are several different ways of applying neighbourhood operations, and at any one position in an image there is a set of common neighbourhood functions which can be carried out at that point. The commonest mode applies a function to all overlapping neighbourhoods in a source image, where the neighbourhood is defined by a template (which also contains weights for each position):

```
image_template (Dest_image, Source_image, N_op,
                                    Template)
```

Here, '`N_op`' is a function which takes a neighbourhood of an image (the region around position <*i,j*> as defined by `Template`), plus the template weights, and returns a single pixel result which is stored in `Dest_image`. Commonly-used neighbourhood functions include:

```
convolution              ( Σ ( pixel * weight ) )
abs_convolution          ( abs ( Σ ( pixel *
                                        weight ) ) )
additive_maximum         ( Max ( pixel + weight ) )
additive_minimum         ( Min ( pixel + weight ) )
multiplicative_maximum   ( Max ( pixel * weight ) )
multiplicative_minimum   ( Min ( pixel * weight ) )
median                   ( median of pixels )
```

This set can be extended by the user with no adverse implications on the parallelisation task (since it can be arranged that each processor has full access to the whole neighbourhood).

Data-reducing window-based template operations in which the neighbourhoods generated by the template are *non-overlapping* are called *tile* operations. There is a different routine for applying a template in this way:

```
image_tile (Dest_image, Source_image, N_op,
                                    Template)
```

For so-called *recursive* neighbourhood operations (in which the result pixel is stored back in the original source image), it is usual to have two complementary scanning directions — forwards and backwards:

```
image_template_forward (Source_image, N_op,
                                    Template)
image_template_backward (Source_image, N_op,
                                    Template)
```

All the above operations on images can have their counterpart on vectors (or a vector can be represented locally as a 1 by *N* image).

Global-level operations. Some library operations require access to the full image:

```
find_histogram (Source_image, Histogram)
find_cumulative_histogram (Source_image,
                                    Histogram)
apply_lookup_table (Dest_image, Table,
                                    Source_image)
```

`image_maximum (Image)` — returns the maximum pixel value in the image

`image_minimum (Image)` — returns the minimum pixel value in the image

`image_sum (Image)` — returns the sum of all the pixels in the image

There will also be a set of input–output routines (e.g. for reading an image from file, grabbing a frame from camera, displaying an image on screen, and writing an image to file).

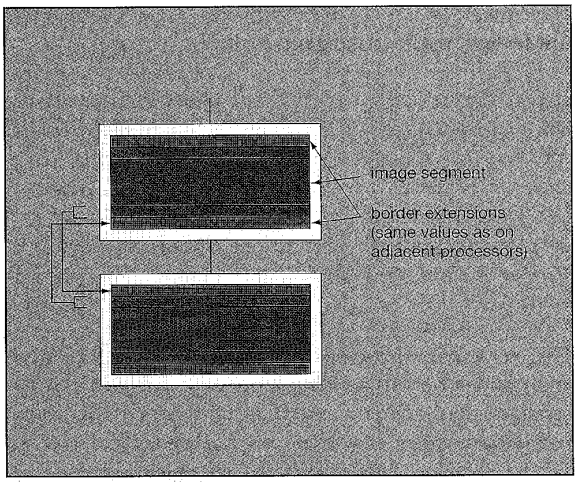


Fig. 9 Image partitioning for data parallelism

Implementation of the image processing library routines

Parallel implementation of the library routines is based on distributing image data across the set of available processors. The interface above doesn't assume a particular way of doing this (e.g. static or dynamic). However, in what follows, for simplicity it will be assumed that data parallelism is used, with static distribution over a network of P processors.

It is convenient, and quite efficient, to partition an image into P horizontal strips. When each processor obeys a *create_image* command, it will first create a descriptor containing all the image's details (Num_rows, Num_cols, etc.), and then allocate a 'data' field containing a segment of storage to hold one strip; but to make neighbourhood processing more efficient, it will duplicate the segment boundary regions (Fig. 9). This border region is like a small cache which, if up to date, will save communicating with the neighbouring processor.

The width of the border extension depends on the maximum window size which will be applied to the image. In practice it is sufficient to have some user-defined (or system default) maximum value 'max_border'. *Create_image* with image size Num_rows by Num_cols should therefore allocate storage for an array:

```
pixeltype Image [segment_X][segment_Y]
```

where

```
segment_X  =  Num_rows/P + 2*max_border
segment_Y  =  Num_cols
```

It also improves efficiency to record whether or not the border regions are currently up to date (by maintaining a flag with the image, which can be set and cleared by two operations *define_borders* and *undefine_borders*).

The coding for pixel-level operations is particularly straightforward, since no communication with other processors is involved. For instance:

```
imagetype  image_image  (imagetype Dest,
                         imagetype Source1, pixeltype
                         ...Op..., imagetype Source2)
{ for (i=max_border; i <
                    Source->Num_rows+max_border; i++)
     for (j = 0;  j < Num_cols; j++)
         Dest->data[i][j] = (*Op) (Source1->
                    data[i][j], Source2->data [i][j];
     undefine_borders (Dest);
     return Dest;
}
```

Similarly for neighbourhood operations:

```
imagetype  image_template (imagetype Dest,
             imagetype Source, pixeltype ...Op...,
                                      templatetype T)
{ if (borders_out_of_date(Source))
           update_borders(Source, T);
  for (i=max_border;  i < Source->Num_rows+
                                    max_border;  i++)
       for (j = max_border;  j < Source->Num_cols -
                                     max_border; j++)
          Dest->data[i][j]  = (*Op) (Source, i,
                                                j, T);
  extend_perimeter (Dest);
  undefine_borders (Dest);
  return Dest;
}
```

The call to extend_perimeter is necessary to define the outer perimeter of the destination image which is always left undefined by a neighbourhood operation. This has nothing to do with parallel implementation, but results from keeping the window entirely within the source image.

This simple image_template routine will carry out any neighbourhood operation which is passed in. Depending on how templates are represented, Op might be the Convolution function below (in which a template holds the co-ordinates and weight of all defined window positions):

```
pixeltype Convolution (imagetype In, int i,
                              int j, templatetype T)
{ pixeltype sum = 0;
  int r,c,w;
  for (w=0; w < T->Num_weights; w++)
  { r = T->rows[w];
    c = T->cols[w];
    sum += In->data[i+r][j+c]  *  T->wt[w];
  }
  return sum;
}
```

Since the automatic bordering means that all pixels in the neighbourhood are available, an application programmer can develop any user-defined function such as the above and apply it to the whole image, in parallel, by using the *image_template* routine.

Examples

- *Thresholding*:
  Desired operation:
  Result = (Image1≥Threshold) * 255;

  Coding using the (parallel) IP library:

```
Result = image_scalar (image_scalar (Image1,
            ge_eq_op, Threshold), mult_op, 255);
```

- *Sobel edge detection:*
  Desired operation:

  Result = abs_convolution(Image1, SobelH)+
  abs_convolution(Image1, SobelV);

Coding using the (parallel) IP library:

```
Result = image_image (
     image_template(temp_image, Image1,
                        abs_convolution, SobelH),
     plus_op,
     image_template(Result, Image1,
                    abs_convolution, SobelV)  );
```

The IP Communications Library

Communication is required at several points during certain of the above IP library routines. The main communication routines in the Communications Library include:

- *Update borders* (which updates the upper and lower borders of all segments). This can be programmed as follows:

  All even-numbered processors do:
  send top border up;
  send bottom border down;

  All odd-numbered processors do:
  receive bottom border from below;
  receive top border from above;

  followed by the same thing with odd and even reversed.
- *Image input* (either from file or from a camera source). This requires the central controller processor to send out each segment separately to each worker processor. Image output is similar.
- *Histogram gathering.* To calculate a histogram, each processor calculates the histogram of its local segment. Then these are accumulated, usually by sending to the controlling processor, and the result is broadcast to each worker processor.
- *Global functions.* Properties like the maximum or minimum pixel value, or the sum of all pixels, again start with a local evaluation of the result followed by an 'accumulation' of the intermediate results into a final result which is then broadcast back to the workers.

Implementation of these routines is most portable if the low-level message passing is coded using a small subset of a standard communications shell such as MPI (Message Passing Interface[12]) or PVM (Parallel Virtual Machine[13]). The main drawback with this is that it can be rather slow. If the code is to run on a tightly-coupled processor network and performance is a top requirement, then it is advisable to implement one's own version of the MPI subset, targeted for the actual hardware being used. In this way, the entire Image Processing Library and the IP Communications Library both remain portable, and do not need to be changed[14].

4 Parallelism in high-level image processing

For low-level image processing, very good speed-ups have been achieved using parallelism. For instance, with 16 processors, performance improvements of up to 15 have been reported[15]. These improvements are because of the regular, geometric nature of the data, and the data intensive and predictable nature of the processing — all of which are features of low-level image processing operations. Unfortunately, as we move up the image processing hierarchy towards high-level processing, these features become less characteristic. Two reasons will be immediately obvious:

- In an image, although pixel data is uniformly distributed, information content typically is not. As we move up the hierarchy, information rather than pixel data is processed, so load balancing becomes a problem. This in turn suggests that dynamic data distribution (using a processor farm, for instance) could be more appropriate.
- Because the operations at higher levels are less data intensive, there tend to be more implicitly sequential parts of the code.

These two effects are now considered in more detail: the second one is the subject of Amdahl's law, and the first suggests an extended processor farm model — the pipeline processor farm.

Amdahl's law

Amdahl's law[16,17], often described as 'the Ohm's law of parallel processing', states that the scaling performance of a parallel algorithm is limited by the number of inherently sequential operations in that algorithm. Consider a problem where a fraction f of the work must be performed sequentially. The speed-up, S, possible from a machine with N processors is:

$$S \leq \frac{1}{f + (1-f)/N}$$

If $f = 0{\cdot}2$ for example (i.e. 20% of the algorithm is inherently sequential), then the maximum speed-up, however many processors are added, is 5 (Fig. 10). Effective system parallelisation thus cannot be achieved by 'cherry picking'

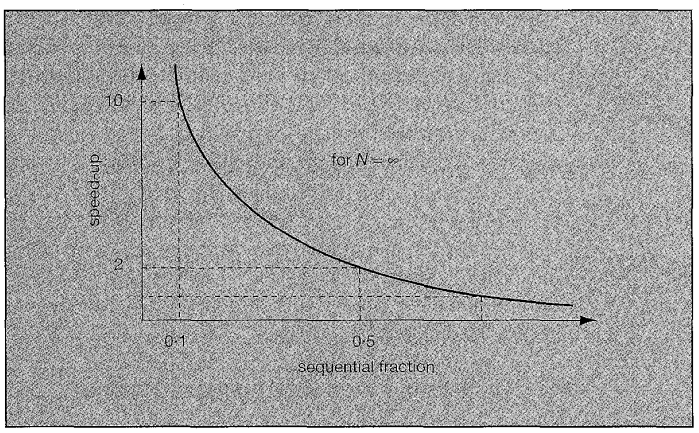


Fig. 10 Amdahl's law implies that, to achieve large speed-ups, virtually no part of an application must be inherently sequential

the components of the algorithm which are most easily parallelised, but instead requires a method of parallelising the complete application algorithm. Hence the techniques introduced in Section 3 to support low-level image processing are necessary but may not be sufficient to achieve the required speed-up or throughput for a complex multicomponent application. In this Section, a high-level view of how to parallelise a complete application in a top-down fashion is therefore taken.

The pipeline processor farm (PPF) logical architecture

The approach described in this Section exploits the commonality inherent in a particular domain of applications within the field of image processing: *embedded systems with continuous data flow*[18]. For example, many applications in robotics, machine vision, image coding, pattern recognition and signal processing satisfy the paradigm that the application algorithm processes a continuous stream of image data.

We also assume that the required design process involves parallelisation of 'legacy code', i.e. software which was originally written to run on a serial processor. This is motivated by the observation that, due to algorithm complexity, relevant applications are increasingly developed as a proof of concept by simulation in C/C++ on a PC or workstation. This leads to a working but non-real-time algorithm which requires parallelisation to meet specified real-time throughput and latency constraints.

To be useful, the approach has to balance generality with performance by avoiding introducing specific architecture and processor dependencies until the later stages of the parallel-design process. The solution produced must also be incrementally scaleable to allow easy porting between different processor types and systems. Finally, to minimise design effort, the approach should also maximise the granularity of the parallel implementation, i.e. minimise the effort expended on partitioning the serial algorithm.

The PPF design model combines the *concurrent pipeline* and *processor farm* paradigms of parallelisation which were introduced in Section 1. The sequential algorithm is first partitioned into a pipeline of separable components, allowing speed-up to be achieved by overlapping execution of the pipeline stages. Within each pipeline stage, a processor farm is then used to speed up execution, allowing a balanced pipeline to be achieved. Processor farms are used because of their flexibility, since a single generic farm template can accommodate data and/or pipeline parallelism, is incrementally scaleable and topology independent, and provides dynamic load balancing.

Parallel waterfall design methodology

Fig. 11 illustrates the design methodology, which is a simple extension of the software engineering waterfall design method of successive refinement. The parallel waterfall design methodology adopts this classic approach wholesale, but extends it by adding some additional stages at the end, once a successful sequential simulation of the algorithm has been written.

Parallelisation is split into four incremental stages. First, the execution time of the sequential algorithm is profiled in a top-down fashion, using tools such as Quantify[19]. The objective is to identify both mean execution time and dynamic variation within the components of the application algorithm. Next, the application is partitioned into a pipeline of separate stages, by identifying logically independent components of the application algorithm, and splitting at points which minimise the data communication requirements between stages. By adding processors to each pipeline stage until their throughput is balanced, a logically complete parallel solution is achieved, although, since both this and the previous stage may be undertaken using a network of workstations/PCs rather than the dedicated target architecture, the solution may not achieve real-time performance.

The final stage involves porting the solution to the dedicated target architecture, where software tools which instrument processor load and interprocessor communica-

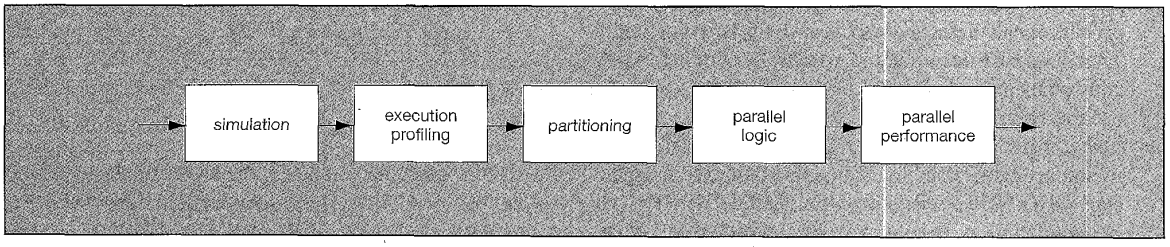


Fig. 11 Stages in the parallel waterfall design methodology

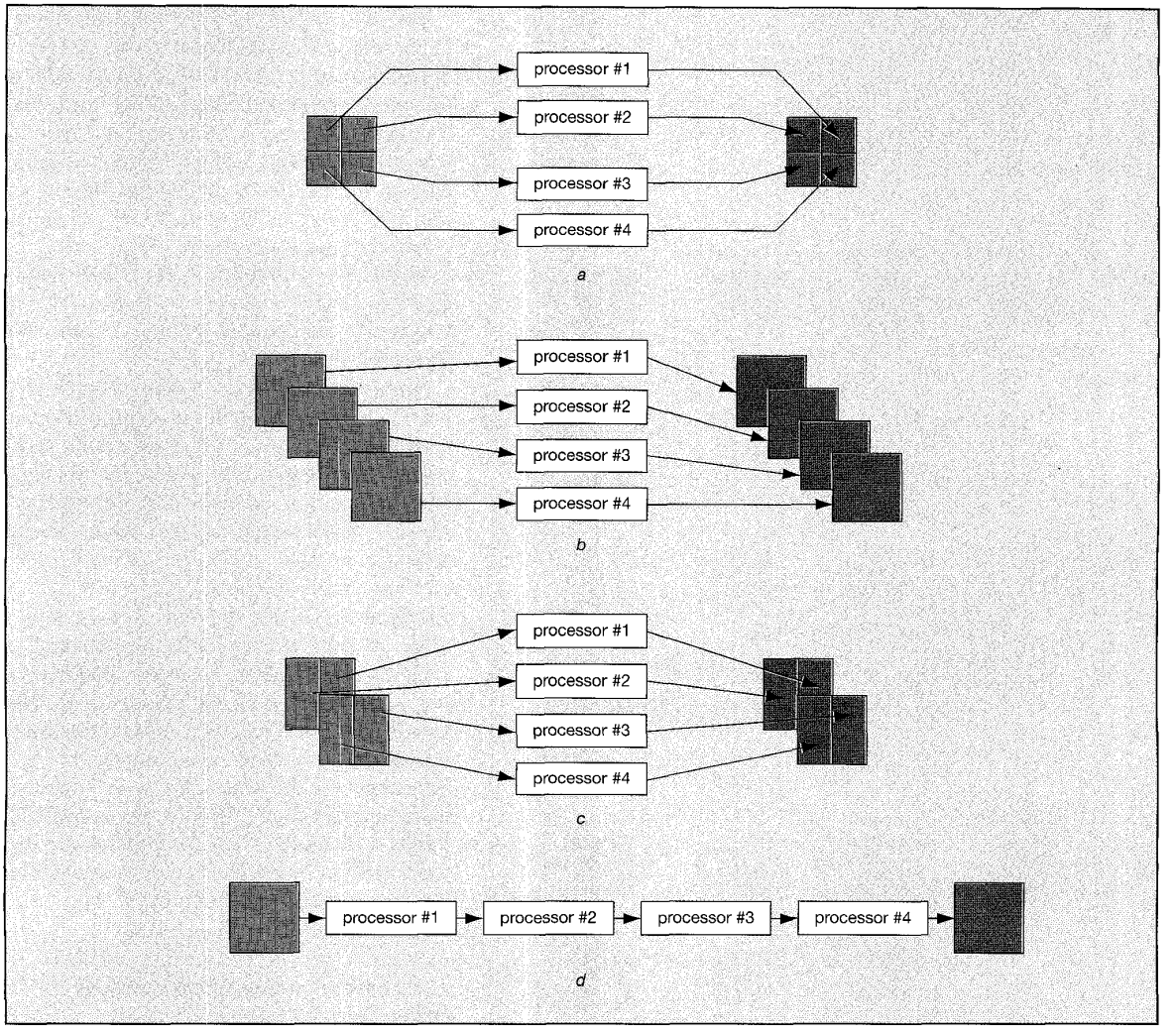


Fig. 12 Pipeline processing paradigms: (*a*) spatial multiplexing; (*b*) temporal multiplexing; (*c*) spatial/ temporal multiplexing; (*d*) pipelining

tion enable an optimised and balanced parallel solution to be achieved. Note that it is only at this final stage that the target real-time hardware is essential; thus, application hardware development costs can be postponed at least until it is clear that a parallel solution with the required real-time performance is achievable.

Pipeline processing paradigms

Within the PPF design methodology, the basic parallelisation technique of data parallelism (or spatial multiplexing, Fig. 12*a*) can be extended in several ways. The continuous data flow through the system can be subdivided so that consecutive data sets are placed on alternate processors (temporal multiplexing, Fig. 12*b*). This has the effect of speeding up the overall throughput of the system by the number of processors used, but without affecting the latency (time between data input and results output for any individual data set), since each individual set of data is allocated to a single processor. This form of parallelism is the easiest of all to implement, since neither the data nor the algorithm has to be partitioned over multiple processors. Spatial and temporal multiplexing can also be mixed in any proportions (Fig. 12*c*). If components of an algorithm are completely independent of each other (task parallelism), a similar approach to data parallelism can be adopted by placing each component on a separate processor. If, however, the components are sequentially dependent, true pipelining (Fig. 12*d*) can be exploited to take advantage of the continuous data flow.

A simple example

Fig. 13 illustrates the process of PPF parallelisation using a simple example. The initial sequential application (Fig. 13*a*) processes one set of data every 11 seconds, giving a throughput of 0·09 data sets/s and a latency of 11 s. Analysis of the sequential code shows that it consists of four separable components, with mean execution times of 1 s, 2 s, 5 s and 3 s, respectively, which can be placed as separate components within a pipeline (Fig. 13*b*).

A particularly attractive feature of the PPF design approach is that the use of a pipeline automatically focuses design effort on that part of the system which constrains

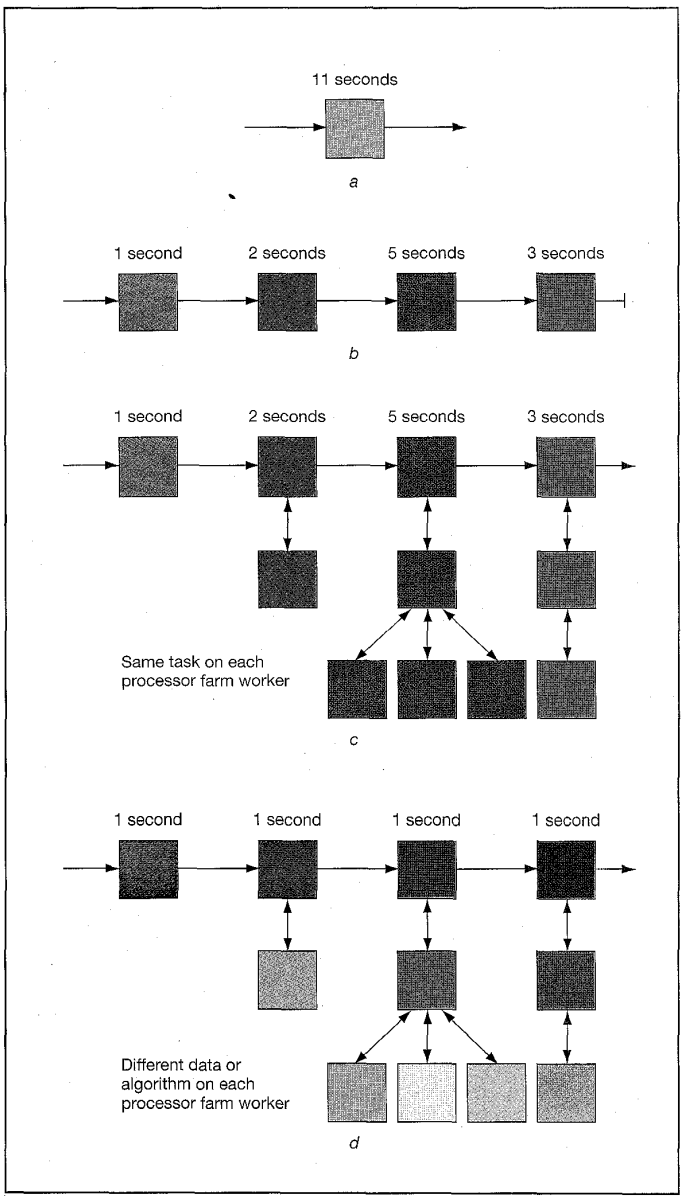


Fig. 13 Toy application, simple unbalanced pipeline parallelisation and alternative parallelisations of the pipeline: (*a*) original sequential application with throughput = 0·09 tasks/s and latency = 11 s; (*b*) simple pipeline of processors, with throughput = 0·2 tasks/s and latency = 18 s; (*c*) PPF with temporal multiplexing with throughput = 1 task/s and latency = 11 s; (*d*) PPF with farmed data/algorithmic parallelism with throughput = 1 task/s and latency = 4 s

the overall performance. In this case, the throughput and latency are constrained by the third stage, and the other three stages are idle for up to 80% of stage cycle time as they wait for stage 3 to complete its processing. Hence the average throughput is defined simply by the inverse of the cycle time for stage 3 (1 data set per 5 s gives 0·2 data sets/s) and the steady-state latency is 5 s at each of the first three stages, followed by 3 s in the final stage (18 s total). Clearly, the initial effort to balance the pipeline should be incurred in parallelising stage 3.

The simplest way of balancing the pipeline is to use temporal multiplexing in each stage. If the number of processors used are *pro rata* to the stage execution times (Fig. 13*c*), the average throughput at each stage will be one data set per second, and tracking a particular data set through the complete pipeline shows that its latency is 11 s (as for the original sequential application). Of course, this is hardly a useful result, since identical speed-up could be achieved using a single farm of temporally multiplexed processors with no pipelining at all!

A more useful approach is to assume that it is possible to introduce data or task parallelism in each stage (Fig. 13*d*), which produces the same average throughput, but with a reduction in latency to 4 s (1 s/stage) for the balanced pipeline. In practice, a 'mix and match' design approach is used, in which sufficient data/pipeline parallelism is sought to meet the latency specification of the real-time system, while temporal multiplexing (easier to program) is sufficient to achieve balance in the remainder of the pipeline. A particular advantage of this approach is that it hides the effect of residual, inherently sequential, components of the algorithm, which would otherwise have the impact predicted by Amdahl's law. In the PPF design philosophy, such components can be placed within the pipeline as a separate stage to which temporal multiplexing can be applied to balance the pipeline.

A real application: handwritten postcode recognition

Real-time recognition of handwritten postcodes was an application investigated for Royal Mail, as a possible addition to their printed-mail sorting systems[20]. The system specification was to achieve a throughput of 10 envelopes per second to match the existing mechanical sorting pipeline, with a maximum latency of 9 s, constrained by the capacity of the mechanical pipeline between the imaging camera and the phosphorescent dot printer which prints a machine readable postcode on each envelope.

Postcode recognition is a typical multilevel computer vision problem, where initial processing of image pixels extracts features which are used to recognise the postcode characters. The final contextual processing stage uses a dictionary of UK postcodes to validate the output of the optical character recognition (OCR) system. Each envelope image is independent, so any convenient combination of temporal multiplexing, data and task parallelism can be used. Many parallelisations are possible, depending upon the design choices made between these options. The example implementation illustrated in Fig. 14 split the application

into three pipeline stages and exploited data parallelism (processing each character in parallel) in the first two stages to reduce the latency to an acceptable level. Temporal multiplexing was used in the final dictionary stage, because the dictionary operates on complete postcodes.

The postcode recognition software was intially developed on a Sun workstation and then ported to transputers for the parallel implementation. Table 1 shows that the mean execution times of the three pipeline stages were in the approximate ratio 3:3:1, with an overall sequential latency of 10·43 s per image. Some data or pipeline parallelism was therefore essential to meet the latency specification.

The performance achieved is illustrated in Fig. 15, which shows the overall speed-up for several different cases where equal numbers of processors are used in the preprocessing and classification pipeline stages (as required to achieve static balance according to Table 1), and the number of processors in the dictionary stage is then varied. For example, the speed-up graph '9, 9, x' corresponds to an implementation in which 9 worker processors were used in the processing stage, 9 in the classification stage and a variable number in the dictionary stage.

The set of graphs in Fig. 15 highlights several important points:

- The 'ideal' speed-up graph is a line of gradient 1 which represents a parallel implementation in which all processors operate at 100% efficiency and no time is required for communication between processors. The speed-up trend shown by the graphs diverges from this ideal as more processors are used, as might be expected due to increasing communication overheads and processor inefficiencies.
- For each graph, as the number of processors in the dictionary stage is increased, the achieved speed-up increases until saturation occurs when the dictionary stage is no longer the slowest within the pipeline.
- According to the static execution statistics given in Table 1, balanced pipelines should be obtained for 7 (3+3+1), 12 (5+5+2), 16 (7+7+2), 21 (9+9+3), and 26 (11+11+4) workers for the five speed-up graphs shown. In fact, significant improvements in scaling performance are achieved by adding further processors to the dictionary stage in each case. This is a result of a bimodal distribution of execution times in the dictionary stage, since 7 character postcodes take about 5 times as long as 6 character postcodes to process. This dynamic effect leads to queuing delays at the output of the dictionary, since postcode ordering within this stage must be preserved. This example highlights both the importance of dynamic effects and the flexibility of the design approach in compensating for them.
- An overall maximum throughput of just over 2 postcodes/s and a latency of 3·2 s were achieved with a PPF of 32 transputers (29 workers and 3 masters). These results corresponded to a speed-up of 21·4 for the application as a whole, and an overall processor efficiency of 67%.

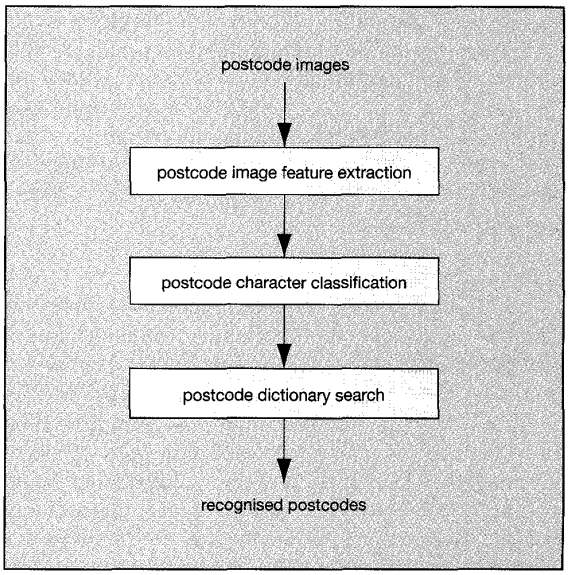


Fig. 14 Three stage pipeline implementation of hand-printed postcode recognition

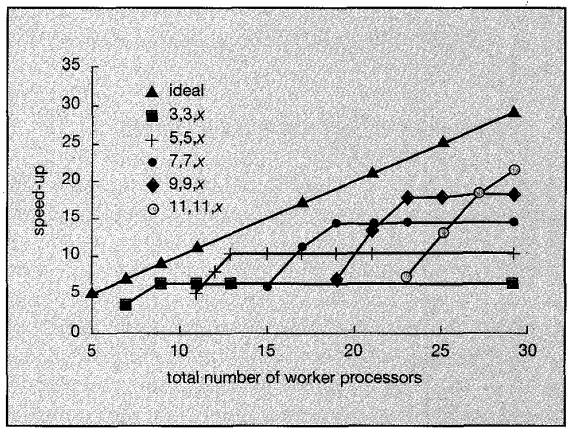


Fig. 15 Speed-up graphs for the postcode recognition pipeline

Although the results achieved demonstrated the flexibility and potential of the design approach, a more powerful computational element was required to meet the throughput specification. A later implementation therefore ported the application to a more powerful parallel processing system which used compound processing

Table 1: Average execution time statistics and communication packet sizes for the postcode recognition pipeline functions

Function	Time, s/image	Ratio	Data packet size, bytes
Preprocessing	4·48	3	2144
Classification	4·45	3	54
Dictionary	1·50	1	202
Overall process	10·43		

nodes, each comprising a transputer communications processor with an i860 computational accelerator. This system achieved a throughput of over 11 postcodes per second with a three-stage pipeline of 7 processors, fully meeting the real-time specification. A number of other application examples have also been investigated, illustrating other aspects of the PPF design methodology[21–24].

Additional refinements

The first-level design approach described above can be quite easily carried out as a paper design exercise, and then transferred to real systems where an optimised parallel architecture can be derived heuristically. The idealised model, however, ignores communication overheads and dynamic effects and so is often subject to substantial inaccuracies, although it provides a useful architecture-independent prediction of upper-bound performance trends. More accurate analysis requires detailed information about the particular processors and communication network on which the parallel application will run. A suite of portable software tools designed to support all aspects of the PPF design process is currently under development to address this need.

5 Future technology and market trends

Applying parallel processing to solve computationally demanding problems involves aiming at rapidly moving target technologies: there is only a limited window of opportunity to exploit a particular processor architecture before it is overtaken by its successors. Designers must therefore have a clear view of future technology and market trends so as to exploit expected developments and at the same time avoid implementing obsolescent solutions. The major technology trends and their potential impact upon parallel processing are therefore outlined below.

Processors

The best uniprocessor performance increases by about 50% per year, and is now totally dominated by developments in microprocessors. The manpower effort required to achieve this improvement is immense, and dwarfs any effort available to develop specialised parallel processing architectures such as SIMD. It has become increasingly difficult for these architectures to compete with the mainstream of MIMD parallel processors built using off-the-shelf microprocessors, and this conservative trend can be predicted to continue.

Memory

Since the 1960s, memory sizes have increased in line with Moore's law by a factor of 4 every 3 years. Unfortunately, memory speed is increasing much more slowly, with the result that progressively more sophisticated caching techniques are required to maintain processor execution speed. At the same time, improvements in compiler design are being sought which exploit locality of reference more effectively. Parallel processing simply exacerbates caching problems, since design decisions have to be made about where to place common data amongst a set of distributed memory multiprocessors.

VLSI

Within the image processing field, two approaches to VLSI (very large scale integration) implementation have commonly been adopted[25]. The first is to utilise a set of bespoke VLSI components typically interconnected in a pipeline architecture to build an application chip. This leads to a minimum chip size and cost, but at the expense of increased design effort. An alternative approach has been to use more general programmable multiprocessor chip architectures (e.g. the Texas Instruments TMS320C80/MVP), which increasingly resemble general-purpose parallel processors-on-a-chip, leading to reduced design effort but at the expense of a larger and more expensive chip. Interestingly, both of these approaches have much in common with the larger scale parallel processing approaches outlined in this paper, inviting speculation as to whether future research could extend architecture independence for parallel processing all the way from general-purpose multiprocessors down to direct VLSI implementations.

Interconnection network

The bandwidth for multiprocessor interconnection networks is increasing, not only by improved processor packaging but also by increasing the number of bits per link. More importantly, the last few years have seen a strong trend away from point-to-point interconnected-processor networks towards topology-independent architectures based upon packet switching. Increasingly, we can expect to see ideas borrowed from the telecommunications field appear as communication technologies for close-coupled multiprocessors.

Software

Whereas parallel processor hardware and interconnections have been subject to rapidly changing technologies over the last decade, progress in solving the problems of parallel software development has been much slower. Lack of a standard topology in the available commercial parallel processors, high software overheads of communication, the trend towards packet-switched parallel processor networks, and the need to accommodate fault tolerance and scaleability have all resulted in a strong trend away from topology-specific parallel algorithms.

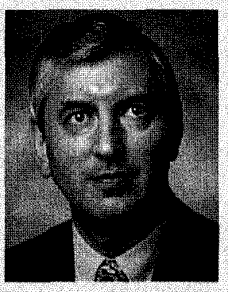
The current requirement is for improved portable software tools to support fast, easy and efficient parallel application development and to avoid the craft mentality inherent in the low level at which most parallel software currently has to be written. The present evidence suggests that although it is unlikely that any parallel software panacea will emerge in the near future, solid advances are being made in a number of discrete niche application areas of parallel processing.

References

1 CHALMERS, A., and TIDMUS, J.: 'Practical parallel processing' (International Thomson Computer Press, 1996), ISBN 1-85032-135-3
2 PITAS, I. (Ed.): 'Parallel algorithms for digital image processing, computer vision and neural networks' (Wiley, New York, 1993), ISBN 0471-93566-2
3 FLYNN, M. J.: 'Very high-speed computing systems', *Proc. IEEE,* 1996, **54,** (12), pp. 1901–1909
4 MULYE, S., and WAGNER, A.: 'A mapping strategy for reconfigurable transputer networks'. Proc. 1995 World Transputer Congress, IOS Press, 1995, pp. 410–421
5 PAGE, I. (Ed.): 'Parallel architectures and computer vision' (Oxford Science Publications, 1988)
6 KUMAR, V. K. P.: 'Parallel architectures and algorithms for image understanding' (Academic Press, 1991)
7 HIRSCH, E.: 'Heterogeneous parallel processing structures for real time image processing applications: reconfigurable and flexible structures versus modular functional structues'. Proc. ESPRIT BRA 3035 Workshop, 'From pixels to features', Bonas, France, September 1990
8 HAMDI, M., and LEE, C. K.: 'Dynamic load balancing of image processing applications on clusters of workstations', *Parallel Computing,* January 1997, **22,** (11), pp. 1477–1492
9 VALIANT, L. G.: 'A bridging model for parallel computation', *Commun. ACM,* 1990, **33,** (8), pp. 103–111
10 RITTER, G. X., WILSON, J. N. and DAVIDSON, J. L.: 'Image algebra: an overview', *Computer Vis. Graph. Image Process.,* 1990, **49,** pp. 297–331
11 CROOKES, D., MORROW, P. J., and McPARLAND, P. J.: 'IAL: a parallel image processing programming language', *IEE Proc. I Commun. Speech Vis.,* 1990, **137,** (3), pp. 176–182
12 GROPP, W., LUSK, E., and SKJELLUM, A.: 'Using MPI: portable parallel programming with Message Passing Interface' (MIT, Cambridge, USA, 1994), ISBN 0-262-57104-8
13 GEIST, A., BEGUELIN, A., DONGARRA, J., JIANG, W., MANCHEK, R., and SUNDERAM, C.: 'A user's guide and tutorial for networked parallel computing' (MIT, Cambridge, USA, 1994), ISBN 0-262-57108-0
14 CROOKES, D., MORROW, P. J., BROWN, T. J., McALEESE, G., ROANTREE, D., and SPENCE, I. T. A.: 'Achieving portability and efficiency through automatic optimisation: an investigation in parallel image processing'. Proc. EUROPAR 98, Southampton, September 1998 (to appear)
15 MORROW, P. J., and PERROT, R. H.: 'The design and implementation of low level image processing algorithms on a transputer network', *in* PAGE, I. (Ed.): ' Parallel architectures for computer vision' (Oxford Science Publications, 1998), pp. 243–259
16 AMDAHL, G. M.: 'Validity of the single processor approach to achieving large scale computing capabilities'. Proc. AFIPS Spring Joint Computer Conf. 30, Atlantic City, NJ, 1967, pp. 483–485
17 AMDAHL, G. M.: 'Limits of expectation', *Int. J. Supercomput. Appl.,* 1988, **2,** pp. 88–94
18 DOWNTON, A. C., TREGIDGO, R. W. S., and CUHADAR, A.: 'Top-down structured parallelisation of embedded image processing applications', *IEE Proc., Vis. Image Signal Process.,* December 1994, **141,** (6), pp. 431–437
19 Pure Software Inc., 1309 South Mary Ave, Sunnyvale CA.: 'Quantify user's guide', 1992
20 CUHADAR, A., and DOWNTON, A. C.: 'Structured parallel design for embedded vision systems: a case study', *Microprocess. Microsyst.,* 1997, **21,** pp. 131–141

Andy Downton was awarded a first class honours degree in Electronic Engineering in 1974 and a PhD in 1982 from Southampton University. In 1986 he moved to the University of Essex as a Senior Lecturer in the Department of Electronic Systems Engineering, where he heads the Vision and Speech Architectures Laboratory. In 1995 he was promoted to a personal Chair. His current research interests are in image analysis and pattern recognition, virtual reality, parallel processing and human–computer systems. He currently supervises research projects on recognition of handwritten forms (the OSCAR project), on parallel architectures for embedded signal processing systems (PSTESPA) and on human body motion tracking and interpretation. He is an IEE Member, a past member of IEE Professional Group E4 and a past Associate Editor of ECEJ.

Address: Department of Electronic Systems Engineering, University of Essex, Wivenhoe Park, Colchester, Essex CO4 3SQ, UK. Email: acd@essex.ac.uk



Danny Crookes has been Professor of Computer Engineering at the Queen's University of Belfast and Head of Department of Computer Science since 1993. He graduated with a BSc degree in Mathematics and Computer Science in 1977 and a PhD in Computer Science in 1980. He currently leads the High Performance Image Processing research group at Queen's. His research interests have centred round designing and developing software tools and techniques for efficiently exploiting high-performance hardware. This has included: intelligent optimising compilers for DSP processors; programming languages and parallelising environments for image processing on parallel machines; and automatic generation of high-performance architectures using FPGAs.

Address: Department of Computer Science, The Queen's University of Belfast, Belfast BT7 1NN, UK. Email: d.crookes@qub.ac.uk

21 DOWNTON, A. C.: 'Generalised approach to parallelising block-based image sequence coding algorithms', *IEE Proc. Vis. Image Signal Process.,* December 1994, **141,** (6), pp. 438–445
22 DOWNTON, A. C.: 'Speedup trend analysis for H.261 and model-based image coding algorithms using a parallel-pipeline model', *Signal Process. Image Commun.,* November 1995, **7,** (4–6), pp. 489–502
23 CUHADAR, A., SAMPSON, D. G., and DOWNTON, A. C.: 'A scaleable parallel approach to vector quantization', *J. Real Time Imaging,* October 1995, **2,** pp. 241–247
24 SAVA, H., FLEURY, M., DOWNTON, A.C ., and CLARK, A. F.: 'Parallel pipeline implementation of wavelet transforms', *IEE Proc. Vis. Image Signal Process,* December 1997, **144,** (6), pp. 355–359
25 PIRSCH, P., and GEHRKE, W.: 'VLSI architectures for video signal processing'. Proc. 5th IEE Int. Conf. on Image Processing and its Applications, 1995, pp. 6–10

First received 25th February and in final form 23rd April 1998