A Dual Source, Parallel Architecture for Computer Vision

by

Norman Raymond Scaife

Submitted for the Degree of

Doctor of Philosophy

at Heriot-Watt University

on Completion of Research in the

Department of Computing and Electrical Engineering

May 2000

I hereby declare that the work presented in this thesis was carried out by myself at Heriot-Watt University, Edinburgh, except where due acknowledgement is made, and has not been submitted for any other degree.

Norman Raymond Scaife

Contents

List of Tables

List of Figures

Acknowledgements

This thesis was written with the help of many people whom I would like to thank for their time and their patience.

I would like to thank my supervisors, Greg Michaelson and Andy Wallace for proposing and managing the project. Greg has taken the time to review the numerous documents that I have produced as part of this thesis and made many other suggestions during the course of my work. Andy has also read much of the written work, particularly those aspects involving computer vision in which I needed considerable support. I would like to thank both my supervisors for the many comments and suggestions they have made.

Bill Austin has been my coworker throughout this project. He has made many contributions to the work in this thesis and has put up with my many programming errors during the system's development.

Guanghua Zhang, Jim Clark and Manickam Umasuthan have all been valuable sources of information and consultation concerning the acquisition and manipulation of image data. I would also like to thank Manuel Trucco for reading the first draft of the JFP paper.

Tore Bratvold provided the inspiration for many of the ideas in this thesis. He also provided valuable introductions to the Meiko system and many references in algorithmic skeletons.

This work has been funded by an EPSRC grant, project number GR/J07884.

Abstract

Computer vision is a very important area of computer science and artificial intelligence with many interesting, varied and practical applications. Parallel processing reduces processing times by simultaneous computation on many processors. Parallel programs are difficult to develop which hinders the use of parallel machines in general computing applications. Functional programming and in particular algorithmic skeletons have been involved in addressing these problems and have the potential to make parallel applications more robust, efficient and easy to write.

This thesis brings all these activities together in a single system with an associated methodology. A method of developing parallel computer vision algorithms is proposed. This employs functional prototyping, algorithmic skeletons and performance modelling. This is evaluated in the development of two parallel computer vision algorithms, Gaussian filtering and edge tracking in intensity images. A range of vision algorithms is developed in a functional language to prove that the method can be applied in general computer vision.

A complete system is built incorporating components which have been developed using the methodology. This is the segmentation component for a parallel object recognition system. Two sources of data, perspective intensity data and orthographic range data are processed in parallel using techniques which ensure commonality of structure between the two routes. This thesis presents the design, implementation and evaluation of the resulting system, developed in Occam2 on a 32 node transputer surface.

Dedication

I would like to dedicate this thesis to my family, in various parts of the world, for their support over many years.

Chapter 1

Introduction

1.1 Introduction

Computer vision has many applications [1]. The problem with many of these applications is that since most images are two-dimensional or of higher dimensionality they are computationally intense and require large amounts of memory. This is where parallel processing can help [2]. The use of a parallel machine can speed up processing times by significant factors, possibly making the difference between a practical and an unusable computation in real-world applications.

Unfortunately, parallel programming carries with it additional problems over and above the difficulties of developing sequential programs. Consider the following quotations:

- Gorn 1958 [3]: "Not only would (parallel) programming techniques be vastly different from the serial ones in the serial languages we now use, but they would be humanly impossible without machine intervention."

- Pepper 1993 [4]: "The diversity of architectures makes parallel programming so much more confusing than sequential programming with its unique von Neumann target."

- Skillicorn 1993 [5]: "The expense of developing and maintaining software is the major obstacle to the routine use of parallel computation." "Parallel computation is still the domain of researchers and those desperate for performance, rather than the normal way of doing things."

- Wray et al. 1993 [6]: "constructing reliable parallel programs is a very demanding task." "parallel programming languages are not, in general, portable." "parallel programs are often hard to read and are difficult to prove correct."

- Darlington et al. 1993 [7]: "Programming parallel machines is notoriously difficult. Factors contributing to this difficulty include the complexity of concurrency, the effect of resource allocation on performance and the current diversity of parallel machine models."

- Skillicorn 1995 [8]: "Building a cost calculus for a parallel program development environment is difficult because of the many degrees of freedom available in parallel implementations, and because of difficulties with compositionality."

- Kesseler 1995 [9]: "The development of efficient parallel programs is an important non-trivial problem that programmers and compiler writers have to deal with."

- Deldarie et al. 1995 [10]: "Parallel programming is widely regarded as a complex, machine-dependent and time-consuming task. This perceived difficulty has hindered the more widespread use of parallel computer systems."

This is where *algorithmic skeletons* [11] are useful. These are a means of developing parallel programs by reusing existing parallel algorithmic structures. The constructs are abstracted and parameterised by relating a construct in a sequential language to the parallel construct with the same semantics, in much the same way a sequential compiler replaces constructs in the source language with semantically equivalent machine code instructions.

Although algorithmic skeletons can be defined in any sequential language and any parallel system, by far the most common host languages are functional languages [12]. This is because functional languages provide a ready-made facility for defining skeletons, higher order functions (HOFs).

Perhaps the simplest example of a skeleton is the well-known functional `map` which applies a function to each element of a list:

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

This can easily be implemented in parallel by distributing the list to several processors, applying the function remotely to each sublist and communicating the results to a central processor so the resultant list can be reconstructed. Since `map` is a very common construct in functional languages it offers the possibility of developing parallel code simply through the use of appropriately placed `map`s.

This suggests a methodology for constructing parallel applications:

- An algorithm is written in a functional language using HOFs and tested on real data.

- An equivalent program is developed using the particular HOFs that are known to have efficient parallel equivalents.
- The parallel constructs are instantiated with code equivalent to that from the functional language.
- This leads to a parallel implementation.

There are other dimensions to this framework, however, such as deciding which constructs to use both for the sequential version and for the parallel version so that the source program is easy and reliable to develop while ensuring that the parallel program runs efficiently.

This idea forms the core of this thesis; parallel vision algorithms are developed in a functional language and translated into parallel versions which are then used to build a complete system through composition.

Some decisions concerning the system are pregiven, however. The only readily available parallel machine was a Meiko Transputer surface [13] with 32 T800 transputers, each with 4M Bytes of memory. This can be programmed in Occam2 [14] or in C. Occam2 was chosen due to its faster execution times. Similarly Standard ML [15] was the functional language with the greatest usage in previous work in this area [16] and with its clean semantics and module system it is ideal for this work. All the working programs in this thesis are therefore based on sequential source code in Standard ML (where appropriate) and parallel implementations in Occam2 on the Meiko transputer surface. Note that these are **not** design decisions, they were provided as part of the original project specification.

1.2 Motivation

One of the problems here is that understanding the whole methodology requires knowledge in several wide-ranging disciplines. The application area is one, image processing and computer vision [17]. Another is parallel programming [2] and parallel architectures [18, 19]. The host languages also need to be considered, Standard ML [15, 20] based on lambda calculus [21] and structural operational semantics [22] and Occam2 [23, 14] based on Hoare's CSP [24, 25]. There are also the implications of combining these disciplines in a uniform manner, for instance parallel functional languages [26, 27] and parallel computer vision [28, 29] each of which is a substantial avenue of research in its own right. There are unique problems created by the juxtaposition of the different histories, philosophies and methodologies upon which each of the component disciplines are founded. Perhaps the odd one out here is the use of functional languages in computer vision which is practically unheard of [30].

Apart from these core areas there are numerous additional considerations which become necessary in the development of multidisciplinary projects. The use of SML to prototype parallel algorithms requires input from areas such as performance modelling [31, 32], program transformation [33, 34, 35, 36] and rapid prototyping [37, 38]. The selection of vision algorithms is also a thorny issue since very often there are several alternatives with no clear-cut means of choosing between them. Developing vision algorithms in parallel places additional resource and algorithmic constraints on this process. Prototyping vision algorithms in a functional language requires even more constraints. Although the utility of both functional languages and parallel programming is generally agreed, there have been few projects reported in the literature which comprise such diverse components.

Thus, despite the complexities facing this project from the beginning, it is a worthwhile exercise in its own right. In particular, there are two aspects in the foregoing discussion which have not previously been addressed in the literature:

1. Algorithmic skeletons have not been used in the construction of a system comprising the number of parallel modules needed here. There are plenty of examples of the development of small, discrete algorithms in which skeletons have been shown to be vital in the design and implementation phases [39, 9] but as a system development tool skeletons are relatively untried. Part of the reason for this is that no fully functional compilers are available which allow the automatic generation of parallel code using skeletons. There has been plenty of work in this area but the implmentations available are research tools rather than fully working compilers [40, 41, 42]. This places limitations upon what can be achieved in this research since all the implementations, including the parallel configuration and harness[1] code are, for the most part, generated by hand.

2. Parallel computer vision has likewise been concerned with individual algorithms rather than complete system.

This thesis therefore attempts to build a major parallel computer vision system which is intended to highlight the problems of system building in parallel for a substantial vision application. The chosen application is object recognition [43, 44] from registered range and intensity data which has many uses in industry and elsewhere. This requires a scene segmentation, a model description and matching algorithm.

The image data available to this thesis is in the form of registered depth and intensity data. A depth image is a mapping of the scene onto an image plane such that each pixel

[1]A parallel *harness* is the code and data required to handle the communications and topological aspects of a parallel implementation.

value represents the distance at that point on the image plane from the image plane to the corresponding point in the scene. The intensity image is a value representing the luminous intensity at the image plane under the perspective projection defined by the camera optics. This images has been subject to several investigations *in our research group*, including the capture of such data [45, 46], issues relating to the fusion of two data sources [47, 48], the filtering and smoothing of the data [49], in segmentation and model matching [50, 51] and in registration of two images. This thesis is complementary to such work and draws upon the experience gained there.

Other prior work includes the development of the functional prototyping methodology from early ideas [52] through several stages of development [16, 53, 54] to essentially the same methodology as used in this thesis [55]. These investigations lay down the groundwork for the system described in this thesis which continues and develops the ideas and experiences contained therein. Another distinct thread of work was the development of an automated parallelising compiler for Standard ML to Occam2. This started with a system for turning recursion into farmed functions called `PUFF` [56, 57] which was based upon a translator from Standard ML to Occam2 called `s2o` [58]. This work was developed into a full parallelising compiler for Standard ML based on algorithmic skeletons called `SkelML` [40, 59, 60]. Although the work in this thesis is not based on these systems many of the ideas and recommendations which arose from that work are implemented here. In particular the prototyping and translation from Standard ML implementation into parallel Occam2 follow closely the methods used in SkelML.

These, then, provide motivation and justification for building a parallel object recognition system using functional prototyping.

1.3 Objectives

There are two overall objectives of this thesis:

1. To specify a method of developing and implementing parallel model-based, multi-source 3D object recognition algorithms using functional language prototypes.

2. To build a complete parallel model-based, multi-source 3D object recognition system from a set of algorithms developed using this method.

Together these form the cornerstone of the thesis work but there are many peripheral activities which have to be taken into account and constrained if the objectives are to be met. A set of subsidiary goals to be used in achieving the major objectives is therefore:

1. Propose and specify a system for model-based, multi-source 3D object recognition from multisource scene data.

2. Review model-based, multi-source 3D object recognition algorithms to select those which are suitable for parallel implementation using algorithmic skeletons and are relevant to the functionality of the proposed system.

3. Review the algorithmic skeletons methodology to extract ideas which are likely to be useful in constructing the proposed system.

4. Develop a sufficient number and variety of model-based, multi-source 3D object recognition algorithms in Standard ML to justify the statement that functional languages are useful for the development of such algorithms.

5. Devise a methodology for the development of such algorithms from functional language prototypes. This includes several components:

   (a) Define a set of useful skeletons.

   (b) Implement these skeletons in Standard ML and in Occam2.

   (c) Develop a means of predicting the parallel performance for such skeletons.

   (d) Test the methodology.

   (e) Evaluate the methodology for ease of use, efficiency of parallel implementation and accuracy of prediction.

6. Use the methodology in the construction of the proposed system.

7. Complete the proposed system by adding *sequential* components and sufficient *interface* components to link the parallel phases together.

8. Test and evaluate the completed system for parallel performance.

9. Review and draw conclusions from the overall project.

This thesis presents the system which arose from these objectives. Some of this work has already been published:

- A single route intensity-based object recognition system was developed and published in Michaelson and Scaife [61]. Parts of this paper form the basis for Chapter 3.

- The partial classification of vision algorithms in Standard ML was published in Scaife, Michaelson and Wallace [62]. An improved version of this is in Section 3.4.

- The four skeletons used in constructing the system were presented at IFL '97 in Scaife, Michaelson and Wallace [63].

- The overall system, including the pose clustering component not covered in this thesis has been published in Wallace *et al.* [64].

In addition, technical reports include:

- The informal specification for the system was the basis for a technical report, Austin and Scaife [65].

- A report on the implementation of the segmentation component was published as Scaife [66].

- The development of the edge tracker is described in Scaife, Michaelson and Wallace [67].

Chapter 2

Literature Survey

2.1 Introduction

This chapter reviews the literature relevant to the main work in this thesis, i.e. the development of a parallel vision architecture through functional language prototyping and instrumentation and implementation using algorithmic skeletons. There is no scope to review all the vast literature that could be construed as relevant to this work so a subset of the most important areas is selected and presented here. In particular, *computer vision* is a large area of investigation and is the application for the work in this thesis. This is not reviewed here but there are many introductory texts [17, 68, 69] available. Instead, a more specific area of *parallel computer vision* is briefly presented. Other papers of interest are cited and described in the main body of the thesis as necessary.

Parallel vision algorithms are briefly described in Section 2.2. This is a vast area of study but some of the more relevant work is presented here as this is the main area of application in the thesis.

Algorithmic skeletons are fundamental to the proposed methodology and are quite comprehensively reviewed in Section 2.3.

2.2 Parallel Vision Algorithms

2.2.1 Introduction

Computer vision is one of the areas where parallel processing can make a significant impact. There are many problems, however, that have to be solved during the successful implementation of a parallel computer vision application. These arise both from the complexity of parallel programming and the wide range of algorithmic problems presented by computer vision work. Thus, according to Wang *et al.* [70]:

> *The computational characteristics of vision tasks are different from the number crunching computations that arise in most other grand challenge applications. A significant fraction of vision tasks consists of extensive symbolic computations, and therefore has irregular run-time data dependencies. Further, vision computations are heterogeneous, utilizing techniques from a variety of disciplines such as signal and image processing, advanced mathematics, and artificial intelligence.*

This has led to a huge variety of methods and architectures being developed for parallel computer vision and it is not possible to review them all here. This section presents a small number of examples to give the flavour of parallel computer vision, there are many comprehensive reviews which can be consulted [19, 29, 44, 71, 72].

2.2.2 Parallel Computer Vision

Image Processing on RISC Systems [73]

Baglietto *et al.* [73] present several low and intermediate image processing applications on a range of RISC machines. Although these are strictly sequential implementations it is claimed that the power of RISC systems allows them to compete with previous generations of massively parallel machines. The four applications cover low and intermediate processing and local and global computation. These are; the discrete cosine transform (DCT), full search block matching (FSBM) (video compression), convolution (CONVOL) and Hough transform (HT). Table 2.1 shows the relationship between these tasks.

	Low	Intermediate
local	CONVOL	FSBM
global	DCT	HT

Table 2.1: Low and Intermediate Image Processing Tasks

The five basic machines used were the HP 735/125, DEC AXP 300/800S, IBM RS6000 43P, SGI Indigo2 Impact and Sun Sparc 20/61. These cover several different architectures, memory types and cache operation. The applications were written in C and were compiled using both a native C compiler and the Gnu `gcc` version 2.6 compiler. The investigation concentrated on detecting and fixing inefficiencies introduced by the compilers. Several such inefficiencies are identified such as loop overheads, data dependencies in pipelines, inefficient data representations and suboptimal cache usage. Several source program solutions to these

Time (ms)	HP	DEC	IBM	SGI	SUN
FSBM	2854	2205	1764	1202	3586
CONV	230	220	190	225	430
DCT	77	86	45	73	110
HT	340	520	320	460	600

Table 2.2: Performance Summary for Image Processing on RISC Machines

problems were also given, these included loop unrolling to minimise overheads, data type optimisation to allow matching of problem data types to the most efficient machine data type and cache optimisation to reduce traffic between cache and main memory.

The best performance achieved for each algorithm and machine is summarised in Table 2.2, the image was a 512×512 byte image. Note that the speedups achieved for the optimisations range from no speedup up to about 5-8 for the more inefficient cases which have greater scope for improvement.

Three main conclusions are drawn from this work:

1. Most image processing programs are inefficient on RISC systems due to the inability of the compilers to select the most efficient schemes.
2. Programmers are required to use "tricks" to map high-level languages onto the given architectures requiring detailed knowledge of low-level architectural issues.
3. Applications are still limited by the amount of computation required and so a higher level of parallelism is recommended.

The overall impressions of this work are that, although RISC machines can compete with some parallel machines, they suffer from many of the same programming and developmental problems as parallel machines. As with parallel machines this could be solved with more advanced compilers which are not currently available.

High Performance Computing for Vision [70]

Wang *et al.* [70] propose a general purpose distributed memory model (GDM) for programming vision tasks on commercially available machines. This type of machine is contrasted with VLSI techniques which are good for structured computation but not for building complete systems and specialised machines which require considerable development effort. A method is proposed for programming general purpose machines, taking into account system characteristics and maximising locality by partitioning and transformation. These machines (for example; IBM SP-2, Meiko CS-2, Intel paragon, SGI Power Challenge, Cray T3D) are cost effective, flexible and portable due to standards such as MPI.

The method hinges on combining compiler optimisations, over which the user has no control and which work better for regular computations, with efficient algorithm design, which is needed for higher level computations. The basic algorithmic approach is to analyse computation and communication, identify bottlenecks and apply an appropriate algorithm to minimise effects of bottlenecks.

The machines are characterised according to memory partitioning (distributed or central), address space organisation (shared or private) and interprocessor coupling (loosely or tightly coupled). Various computational models are described, including PRAM (parallel random access machine), BSP (bulk synchronous parallelism), LogP, BDM (block distributed model) and GDM. GDM is claimed to be for general purpose message passing machines and can model low latency systems. It is based on startup time and transmission rate and can also handle overlapping communications and computation. Global operations such as broadcast and barrier synchronisation are handled with a global operation time. This model is proposed for vision applications on high performance computers.

The approach outlined comprises:

1. Characterise architecture using GDM model (calculate startup time, transmission rate).
2. Generic algorithms in computer vision at all levels are considered and used as the basis for scalable algorithm design. The performance of each algorithm is estimated by analysis.
3. The algorithm is implemented on distributed memory machines using the MPI model.

Several parallel algorithmic techniques are described including load balancing, communications overhead minimisation and data reorganisation. Other methods include reduction of message overheads by packing (vectorisation), reduction of node contention by communications scheduling and taking advantage of special communications hardware by software reorganisation. Asynchronous algorithm techniques are used since lack of synchronisation reduces idle time for irregular computation. Data re-mapping between phases allows matching of data structures to computation at each phase.

An example application is described which involves detection of buildings from aerial photographs. This is divided into several phases: line extraction followed by perceptual grouping and then rectangle selection and verification. A LISP implementation takes 75mins for 1K by 1K images.

Line finding starts with edge detection by convolution, this is easy to parallelise. Irregular dependencies in linear approximation are overcome by asynchronous algorithm development.

Written in MPI, this takes 0.065s on a 64 node SP-2 (compared with 3.45s on single node SP-2) and 0.1s on a 512 node CM-5.

Perceptual grouping implements grouping of lines, junctions, parallels and U-corners in a hierarchical manner. Grouping is performed as a scatter/gather operation over geometric decomposition. Communications overheads are minimised using message packing in conjunction with a 5 stage communications scheduling algorithm for many to many communications. Line grouping takes 0.633s on a 256 node CM-5.

This work bears some similarities to skeletons work, in that generic algorithms are specialised for specific applications. There is also a performance modelling component in terms of predicting the performance in order to identify and minimise weaknesses in the methods. The programmer is still left with a great deal of the responsibility for producing efficient parallel implementations, however, since many of the optimisations have to be selected and implemented at the source code level.

Parallel Image Processing with the Block Data Parallel Architecture [74]

Alexander *et al.* [74] describe a parallel programming model, the block data flow paradigm (BDFP), and an attendant architecture, the block data parallel architecture (BDPA), intended to reduce communications overheads and synchronisation constraints for digital signal and image processing algorithms. The model and architecture are described and several DSP/IP applications are given.

Here, the authors claim that the reasons for poor parallel performance include: mismatch between parallelism in the algorithm and the machine, synchronisation bottlenecks, resource contention, timing problems such as clock skew and the fact that programmers have to manually optimise code. The BDFP is intended to alleviate these problems and is based on data partitioned into large blocks where each block assigned to a processor module (PM). Processing starts on arrival of a block (no global start and stop control) and each PM outputs results as soon as they are available. Interprocessor communications are asynchronous and point-to-point.

The BDPA is designed to implement BDFP and consists of an input module (IM), output module (OM) and processor module array (PMA). A processor module (PM) can be multiple processor with features such as tight coupling and synchronisation. Each PM has direct connection to the input and output modules and there are two input and two output buses to improve bandwidth. The PMs are also connected in a unidirectional ring which allows pipelined operation and there are FIFO buffers between all components. The overall philosophy is that the input is assumed to be a data stream (since DSP/IP) and the IM takes input stream and generates and distributes blocks. The OM takes the processed blocks and generates the output stream but can also contain a (sequential) postprocessing module.

A mapping strategy from algorithms to the BDFP is given which is designed to minimise communications load and resource contention (for example access to the IM, OM and PM ring). An initial algorithm as finite difference equations or signal flow graphs (SFG) is transformed into a state-space representation. A good partition is then generated from analysis of the state-space model and scheduling derived from the partition. Synchronisation and communications requirements are also calculated, these use coarse grain parallelism, where possible, but resort to fine grain if coarse grain parallelism fails. Note that blocks can be rows, columns or subimages.

An example mapping is given for discrete linear shift-invariant systems (DLSI). These can implement convolution and other image processing algorithms, for instance averaging of pixels. This uses a modified version of Roesser's state-space model where state variables are assigned to the input of the delay elements for compatibility with the hardware implementation. The 2D state-space model is implemented as a linear finite state machine and the parallel implementation schedules row computations on separate processors. It is claimed that this scheme can be extended to cover nonlinear systems (for instance, a median filter) since data communications patterns are the same for nonlinear systems as for DLSI.

The basis for the evaluation of the system is a set of performance analysis tools. These include: *sogae*: a dataflow tool for fast approximate analysis of parallel algorithms, *erg*: a detailed simulator for validating the correctness of the mappings used and a prototype hardware system for demonstrating algorithms in practice. Several example applications are given, including several matrix algorithms (which are justified by pointing out that many image processing applications can be expressed in matrix form). Most speedups are estimated using the `erg` simulator but the authors are currently constructing TMS320-based BDPA hardware. The applications are:

- 2D infinite impulse response filter (IIR). This consists of a three-stage pipeline with two multipliers and three adders and is an instance of the DLSI mapping. Speedup is estimated at 26 for 28 processors for a fourth-order filter but is limited by the algorithm (no further speedup after 28 processors).

- 2D finite impulse response filter (FIR). This is a subclass of the IIR filter and speedup is almost linear for a sequence of six 64 x 64 images.

- 2D discrete cosine transform (DCT) [75]. This is computationally intensive ($O(n^3)$) and uses a hierarchical implementation with 8 processors per PM and a block decomposition. Almost perfect linear speedup is achieved.

- Cholesky factorisation [76]. Here, blocks are columns and are input and output from each processor. The same code is implemented on each processor apart from boundary processors. Linear speedup is found for up to 32 processors.

- QR decomposition. This method uses Givens Rotations [77] and has a column-based decomposition. Each processor eliminates subdiagonal elements for its column and passes along the state variables. Speedup is estimated at about 22 for 32 processors.

- Back substitution. This constitutes a back end for QR to solve linear systems of equations. This time a row-based decomposition is used and simulated speedup is about 22 for 32 processors.

The main conclusion from this work is that a wide variety of DSP/IP applications can be implemented with BDFP/BDPA. The authors are experimenting with a multiple DSP board but a major bottleneck is input/output. Another avenue being investigated is a hierarchical BDPA but problems have been found with global communications.

A Modular Massively Parallel Computing Approach to Image-related Processing [78]

Krikelis and Lea [78] describe a modular massively parallel computing approach intended to allow cost effective high-performance image processing which is upgradable for future developments. At the core of this approach is a series of multiple SIMD processors with attendant input/output provision and higher level control. A software environment for this approach is suggested.

The Modular-MPC approach comprises an MPP block which has multiple instruction streams operating on multiple SIMD processors (called MIMSIMD), a data stream manager (DSM) for image data handling and an instruction stream manager (ISM) providing independent control over over the individual processors (called task execution units TEU). Interprocessor communications are handled by an inter-task parallel data router and an inter-task sequential router. Each TEU is capable of handling different data sources, an example given is that one could handle intensity data whereas another could handle depth data. The core of a TEU is an associative string processor (ASP) where image data can be sourced from external IO, the host interface or from the inter-task parallel router. Each TEU has a parallel data buffer which is used to store both raw and processed image data. It also stores local data and accesses remote partitions via the inter-task parallel router. The parallel data buffer can be used in distributed mode or in shared memory mode using dynamic routing.

Mapping of image data onto ASPs can be regular (either block, column or row) or irregular. Patch shape is decided by three criteria: the number of pixels simultaneously processed should be maximised, communication paths should be minimised and the effects of patch overlap should be minimised. The most common patch shape under these criteria are rectangular patches across the height or width of the image. Spatial processing (window operations) are implemented by applying a series of scalar/vector and vector/vector operations in conjunction with appropriate data movement, an example is convolution implemented as n^2 scalar/vector multiply and accumulate operations. Point processing is similar to spatial processing but without data movement. One exception to this is image statistics (for instance counting) is effectively point processing but with global image transfers, this is implemented as a data movement operation. Connectivity algorithms (for example, connected components, skeletonisation and geometrical operations) are implemented as an associative search followed by a mixture of synchronous and asynchronous communications to establish the area of connectivity. Object information (such as area, position, size) is stored as a record in contiguous sequences of associative processing elements (APE) which are the components of an ASP. This information can be dumped out or can be "garbage-collected" to one end of an ASP module.

The software environment is purely hypothetical but the intention seems to be to provide libraries for common DSP/IP functions hosted in languages such as Modula-2/C++ and front ended by a data language such as IDL and a GUI interface. A prototype system called ASTRA has been built which can host 8 boards, each with 32 VLSI devices each of these containing 64 APEs. Three image processing applications are given:

- Near-earth object detection from astronomical images. These comprise $2048 \times 2048 \times 2$ images and are processed using a wavelet filter [79] followed by object description calculation (centroid and invariants). It is predicted that a 32768 APE machine organised as 8 ASP modules would be able to perform the calculation in "real time".
- Fingerprint classification from 8 bit grey-scale 512×512 images (these are stored in lossless JPEG format). After decompression the images are processed using three phases: ridge-valley feature extraction, direction finding and registration (which involves finding an orientation invariant point for features). Final classification uses a three stage backpropagation artificial neural network. The prediction here is that 4096 APEs in 4 ASP modules can classify one image in 0.8 seconds (including decompression).

- Volumetric visualisation (for example, medical images and molecular modelling). This occurs in 4 stages. Data formation involves processing of 2D slices which are then stacked into 3D volume data. Data classification in 3D using, for example, connected components or texture analysis. Data manipulation includes both geometric calculations and volume merging where voxels are assigned values according to position and content. Data viewing projects the 3D volume onto a 2D image plane with shading and compression. This time it is predicted that 65536 APEs in 16 ASP modules can process volumetric visualisation of $256 \times 256 \times 256$ data at 30 frames per second.

The architecture described here has considerably more flexibility than other SIMD systems and is able to exploit both control and data parallelism (although the emphasis is strongly on the latter). The hardware is also upgradable and expandable to suit new applications as they arise but the software environment does not exist yet. A wide range of computationally intensive vision problems have been addressed but require considerable knowledge on the programmers part for implementation.

Algorithmic Techniques for Computer Vision on a Fine-grained Parallel Machine [80]

Little *et al.* [80] describe the implementation of a number of computer vision algorithms using a vector model. These are claimed to be more useful for practical purposes than PRAM models although it is recognised that there is a performance tradeoff in using an abstract model which does not allow the use of the hardware-specific features of the target architecture.

The primitives used include:

- Elementwise arithmetic and logical operators.
- Permutation primitives (re-indexing of vectors). Non-unique indices result in *combining* operations where a binary associative operator is supplied.
- Grid permutation primitives. Only 2D grids are used here and restricted to *up* and *down* operations (rotations).
- Scan primitives. These are also called *prefix* operations and turn a vector into a vector of accumulated values:

$$[a_0, a_1, \cdots, a_{n-1}] \rightarrow [a_0, (a_0 \oplus a_1), \cdots, (a_0 \oplus a_1 \oplus \cdots \oplus a_{n-1})] \tag{2.1}$$

- Global primitives. Reduction using a binary associative operator.

- Segmented primitives. A vector is broken down at points marked by flags and can be defined for both permutation and scan primitives.

All of the above primitives have implementations on the connection machine (CM-2). Simple operations such as elementwise operations are easily supported, primitives requiring data movement are handled using the routing hardware of the CM-2 with only a $O(log\ n)$ performance penalty compared with the model. Scans are implemented using a binary tree algorithm in $O(log\ n)$ time.

In addition to the primitives it is possible to build up routines by combining primitives together. A number of routines are given which are used in constructing the vision applications. These are:

- Pointer jumping. These involve linked cycles of elements and can be calculated in $O(log\ n)$ time using two calls to *permute* and some elementwise operations.

- Ordering. These transform explicitly ordered lists of elements into contiguous implicitly ordered lists. This is a four stage algorithm using combine, scan and permute operations.

- Region summation. This sums elements in a rectangular region and is implemented using a constant number of scans and permutes.

- Outer product. Again this is calculated by fixed numbers of scans and permutes. The vectors are expanded by scans and then one is permuted to give the correct output.

- Histograms. Four methods are given for computing histograms using sorting, combine, counting and probing methods, each has different strengths.

Several image processing and computer vision applications are given.

- Gaussian filtering. This is implemented using a binomial approximation which reduces the calculation to an iteration involving summations of grid permutations (separated in each dimension). A different method involving iterated convolution with a uniform filter is suggested for large σ values.

- Canny edge detection. This is divided up into phases, the first is local image operations which are implemented as convolutions, non-maximal suppression can be implemented using grid permutations. The next phase is thresholding with hysteresis. Thresholding is based on histogramming by probing. The hysteresis is carried out by propagation, where all the values in a list are communicated to all other values. This is implemented using pointer jumping.

- Border following. This follows from the Canny edge detection by propagating labels along the connected lists of edges, again using pointer jumping.

- Hough transform [81]. This is implemented as a combination of an outer product of the set of rotation points and the set of image points, elementwise to produce a vector of ρ values. This is then histogrammed to produce Hough space.

- Connected components. This can be implemented using either $O(log\ L)$ permutations where L is the length of a list of elements using pointer jumping or, alternatively, using segmented grid scans where the complexity is dependent on the nature of the image.

- Convex hull. Several methods are considered, one is translated from a PRAM algorithm, another is a version of the QUICKHULL algorithm implemented using permutations and scans. The merge-hull algorithm is adapted using $O(log^2\ n)$ permutations and a version of the Jarvis-March algorithm [82] is mentioned which has a global reduce per point.

- Voronoi diagrams [83]. A method is described which uses a brush-fire technique. This is linear in the grid size but depends on the point density.

- Generalised Hough transform. This is the only high-level vision application given and is implemented using an outer product of the scene and model features followed by quantisation and a 3D histogram.

Although the parallel methodology is described and there are numerous examples cited from computer vision, implementation details are a bit thin on the ground. Very little information is given for the performance of the algorithms. The Canny algorithm is claimed to run in 50 ms for a 256×256 image on 64k processors on the CM-2. The Voronoi diagram takes 250 ms for 1000 points on a 512×512 image. Times for the convex hull operation take from 12 to 150 ms for the different methods.

PARADOX

Wang and Brady [84] describe a heterogeneous system called PARADOX intended for frame rate vision applications at the low and intermediate level. The machine consists of a network of 32 T800 transputers with variable topology and a Datacube pipeline processor which can store and process images at frame rate. These are connected via a BAe I/F card which has 512k of dual ported memory and one onboard transputer. The memory is shared between the onboard transputer and the Datacube and the transputers' links are connected into the transputer array allowing fast transfer of image data from the Datacube to the transputers. The transputers also have a graphics display and the whole system is bound together using a Sun 4 and VME bus.

Vision algorithms implemented on this system include the Canny edge detection scheme which, it is claimed, can process 256×256 images at frame rate. Another application is the Droid algorithm which is a structure from motion system intended for use in the guidance of AGVs. This works by tracking image features in a sequence of images and determining 3D locations from their image plane trajectories. The features used are corners and corner detection is parallelised using processor farming on the transputer array. The BAe card performs the partitioning. Gaussian filtering is built into the Datacube pipeline which also tags the images with timestamps.

The system can process the Droid scheme at three 256×256 frames per second which is a speedup of about 48 compared with a Sun 4 processing a prerecorded sequence of images. Various optimisations could be implemented for instance moving some of the corner detection onto the Datacube (convolution operations) or moving some of the sequential postprocessing onto the transputer network.

The Canny algorithm is implemented as smoothing with a 7×7 Gaussian kernel on the Datacube followed by edge enhancement using a 3×3 Sobel kernel and non-maximal suppression on the transputer array. Again, frame rate processing of this algorithm is claimed but no speedup figures are available.

MARVIN

Rygol *et al.* [85, 86] describe a system called MARVIN which has some similarities to PARADOX. Here, stereo intensity images are captured and processed using pipelined frame-rate hardware. At present, this allows frame-rate processing of convolution operations but edge and corner detection are also possible. This data is distributed via the MAXbus video bus (PARADOX also uses this bus) to a toroidal arrangement of 3×8 T800 transputers. Each transputer is configured with 1 Mbyte of video ram and 4 Mbytes of DRAM. Eight of the transputers are linked with a set of four video rate buses which are used for fast distribution of captured images around the network (two are used for input of the two cameras, one is used for an output display screen) the other transputers are 2 Mbyte transputer modules. The whole system is hosted by a Sun 3 which is connected via VME bus to the framestores and to a transputer on the network.

The system is programmed using Parallel C (3L Ltd.) and a local runtime environment called MRE. This implements a message passing system with a *nameserver* facility to allow addressing of specific tasks on remote processors and to allow enquiry and allocation of shared resources. A system of dynamic loading is used for task placement, this uses a configuration file for setup but can be altered by function calls once running.

Vision algorithms implemented include a stereo matching algorithm called PMF [87] which is effectively a 1D search along an epipolar line in the left image using gradient and high-level grouping constraints from the right image. While matching occurs the edges are also classified according to the type of feature they can represent (currently straight lines). Line fitting is implemented on top of the 2D edge strings using a recursive fit and segment method. Finally, 3D geometrical descriptions can be generated from the disparity information. This algorithm is front ended by the Canny edge detection scheme which is applied to both images.

Processing of the two images is carried out using pipelined system over a row-based geometric decomposition. Partitioning is handled by having servers for each column to minimise redundant image transfers. The entire image is processed in a partitioned state with recombination left until the last stage where the 2D data is joined, repartitioned for calculation of 3D data and then recombined for output. Processing times for the complete algorithm are about 15 seconds for 8 processors on a 512×512 image dropping to about 9 seconds for 16 processors.

A parallel model matching algorithm is also described. This is essentially a tree search optimised by various constraints and grouping matches into cliques. This occurs as a three level hierarchical setup. The lowest layer (*view matcher*) identifies data within the scene. The intermediate layer (*virtual matcher*) is effectively a communications layer which hides the parallel implementation from the matching and integrates data from the lowest level. A virtual matcher can be provided for each model giving additional parallelism. The top layer (*rootmatcher*) integrates the system and allocates processors to the virtual matchers. These layers are organised as client/servers using RPC for communication.

Processing times vary strongly with the number of features recovered from the scene, with a maximum of about 90 minutes for a scene with 110 features and about 10 minutes for a scene with 23 features. This has a maximum speedup of about 12 for 16 processors. MARVIN also has the ability to track moving objects and is intended for AGV applications.

Apply and Adapt

Apply and Adapt are two special-purpose languages for developing parallel image processing applications in an architecture-independent manner [88]. This is possible because the types of algorithms are *regular* and are not *algorithmically complex*. Parallelism is also desirable because of the need for fast processing times during applications. Apply is specific to *local window operations* whereas Adapt can handle a wider range of image proccessing applications using three different parallel models.

Apply is an architecture independent language for expressing local window operations. Wang, Dew and Webb [89] describe a transputer implementation of this language. Here, several low-level image operators are implemented in the language and run times are compared with times on a Sun 3 and CMU Warp machine. The Canny edge detector is used as an example of a multi-stage operation and used to make a comparison with hand-written code.

The local window operation is implemented as a statically distributed processor farm and has implementations on a number of architectures. This type of distribution is possible because the decomposition is known in advance.

The Apply language is syntactically similar to Ada but each procedure defines a local window operation from an input image to an output image. Data from outside the local window can be handled automatically by the compiler which generates object code for handling image data and performing calculations on the local windows. Communication and computation are overlapped.

Two topologies are considered, a quad-ring and a parallel one-dimensional array (PODA). The quad-ring causes a bottleneck at the host since the input and output images are stored there. The PODA splits the data up into multiple data streams and is less prone to the communications bottleneck. The PODA is thus recommended for Apply. An interesting point is that the host processor can be replaced by a frame grabber which can enable video-rate low-level processing of images.

The performance of Apply for several different edge detection algorithms on the Sun 3, CMU Warp and the two transputer topologies are compared. In general, the Warp machine gives the best times and the PODA outperforms the quad-ring by up to a factor of four. There is little performance improvement for more than 32 processors in either of the transputer topologies.

The Canny edge detector is used as an example of mulitphase image processing and an extension of Apply to allow load-balanced pipelines is described. This is compared with a non-pipelined implementation which takes twice as long as the pipelined version. It is also compared with a hand-coded Canny implementation and gives very similar run-times.

A comparison is made with Linda [90]. Linda is a general purpose system based on a *tuple space*, Apply is a special purpose language for low-level image processing. Both hide the parallel aspects of the system from the programmer but Apply is able to automate the partitioning better.

Adapt [91] is a more general purpose data parallel language for image processing which isolates the programmer from parallel issues in the same way that Apply does. Here, three basic models are presented to the programmer: split and merge, raster-order per-pixel processing and scanline/transpose. These can be used to implement histogram, convolution, run-length encoding, image warping, connected components and FFT algorithms.

Split and merge is a standard divide and conquer strategy to one level. Adapt has a specialised implementation of this where the division is row-based. A connected components algorithm can be implemented using this model, connected components within a row are labelled and then a table is built up linking components connected between rows, a final stage applies the table to the image. A compilation scheme for this model is described, the key points are in choosing an appropriate I/O module and managing specific types of variables. The row-based restriction is claimed not to limit parallelism based on the characteristics of the split and merge implementation. The split and merge model is said to be suitable for *local operators* (regions of the input image map to the same region in the output image) and *feature extraction operators* (global output variables are small compared to the computation required to generate them).

Raster order image processing is where intermediate values computed at a given pixel are used in processing subsequent pixels, in *raster order*. An example is a median filter. This model is implemented by retaining variables between phases, with an initialisation during the first phase. Two implementations are described on a CMU Warp machine, column-partitioned and row-partitioned. Column-partitioned required more communication but less storage but row-partitioned performed better overall for a range of algorithms including histogramming, minimum bounding rectangle, run-length encoding/decoding and median filter.

The scanline/transpose method is for algorithms where processing can be partitioned into two one-dimensional steps with a data reorganisation between the steps (the image itself is a global output variable). One example is FFTs which can be applied in each dimension separately, another is image warping, for instance an affine transformation defined as the composition of two skew operations. These operations can be implemented in Adapt using the transposed-order read/write facilities for input/output images respectively. The scanline implementation is trivial since this is already row-based. Implementing the transpose step on row-partitioned data is much harder since it involves communication between all pairs of processors. This can be done by sending the data to a central processor and redistributing or could use special purpose hardware to implement the exchanges.

An example of multi-baseline stereo imaging is used as an example of all three methods. This uses a raster order process to compute errors locally followed by a local window operation to sum the errors, these two steps are repeated for each depth value and then maximum correlation at each pixel value is used. This is compared with a hand-coded C implementation and achieves a very good runtime compared with the C version but, it is claimed, the Adapt implementation is more scalable due to fewer redundant computations.

In summary, Apply provides a method for performing local window operations using a specialised pipeline as the parallel implementation on transputers. Adapt provides three models for image processing, this time more varied and with more general applications in image processing. All four parallel models are effectively skeletons, the comments made concerning hiding implementational details from the programmer are typical of the claims made for other skeleton work and many of the issues in this thesis are also addressed by this work, particularly with regard to data placement and data reorganisation issues.

2.2.3 Summary

Although only a small cross-section of the field of parallel computer vision algorithms has been presented, this has given an idea of both the importance of this area of research and its relevance to industrial and scientific applications. It is also apparent from the work presented that there are many problems to be overcome before these systems can be designed and implemented with complete confidence. The main point here is that both the application domain and the characteristics of the parallel machine have to be studied and matched up if the requirements of the system, in terms of both performance and functionality, are to be met.

2.3 Algorithmic Skeletons

2.3.1 Introduction

The search for an abstraction mechanism for parallel programming has been going on for a long time. This is well-motivated by the success of abstraction as a means of developing and managing sequential programs [92, 93, 94] which has led to the development of full abstraction mechanisms for sequential programming languages such as object-oriented programming [95, 96, 97, 98, 99, 100, 101], the module systems for SML [15, 20, 102] and Haskell [103]. These abstraction mechanisms allow modular program construction, independent development of program modules by separate programmers and are influential in making sequential languages relatively independent of the target architecture.

These are highly desirable attributes for parallel programming but the development of abstraction mechanisms has proved much more elusive for parallel systems. The problem could be that parallel programming comprises activities which are time dependent, for instance synchronisation mechanisms and communications patterns. Another point is that parallel programming encompasses several fundamentally different activities such as; deciding upon suitable data structures, problem decomposition and mapping tasks to processors. All of these activities have to be included in an abstraction mechanism which places a great deal of complexity upon the designers of the abstractions and the programmers who use the abstractions.

One approach to parallel system development which has arisen in recent years is a concept called *algorithmic skeletons* [11]. Although there is no formal definition of this concept, all the current interpretations involve a *conceptual link between constructs in the source language and constructs in the parallel implementation.* The common structure between the two is known as an algorithmic skeleton. One immediate advantage of this situation is that a given skeleton can have implementations on several architectures and can be defined in more than one source language. This suggests a degree of both source language and architectural independence.

The idea derives its power from the vagueness of this link between source and target constructs. It allows all aspects of parallel computation to be encompassed within the source language in a reusable manner without grounding them in particular architectural features. Unfortunately, the vagueness of the definition allows a high degree of latitude among researchers in this area and so has spawned a number of different approaches which have had various degrees of success in implementing the idea in a practical and robust manner.

- *How should the concept be presented to the system programmer?*

  One idea is to give the programmer a finite set of system implemented skeletons and constrain the programmer to work with only these building blocks. This approach is guaranteed to give an efficient parallel implementation but places an onerous burden on the programmer to ensure that the right skeletons are selected and even to define the particular programming problem within the given constraints at all. Another possibility would be to view the skeletons as centres of parallelism within the source language that an automatic parallelising compiler can exploit. This is slightly easier on the programmer but there are no guarantees that the compiler can generate an efficient parallel solution from an arbitrary piece of sequential code and also assumes that all the necessary centres of parallelism are recognisable as such.

- *Relating source language constructs to specific groups of parallel constructs on various architectures.*

  For instance, should the source and target languages be the same, and if not how are the properties of the parallel implementation to be predicted by the process linking the two? Another difficulty is deciding which features of the parallel construct need to be parameterised to give the skeletonising process enough flexibility to express arbitrary algorithms sequentially but to allow the parallel construct implementor sufficient flexibility to guarantee efficient parallel performance. This is related to the original problem of defining abstraction mechanisms for parallel systems, although in a somewhat more constrained form.

- *How general is the approach?*

  One of the biggest unanswered questions in skeletons technology is: what constitutes a minimal set of skeleton definitions to guarantee both the ability to express the solution to any programming problem and the ability to generate an efficient parallel implementation of any sequential program written using algorithmic skeletons?

  Of these two requirements the parallel efficiency is the hardest to achieve. Any computationally complete host language can express any problem solvable by symbolic manipulation (with or without skeletons) but it is much harder to ensure that a particular program has enough skeletons in strategic positions to generate a parallel solution at all, let alone an efficient one.

Despite these obvious difficulties, the algorithmic skeletons idea has attracted considerable interest among researchers into parallel systems, functional languages and theoretical computer science. The reasons for this are due to the many advantages of the approach:

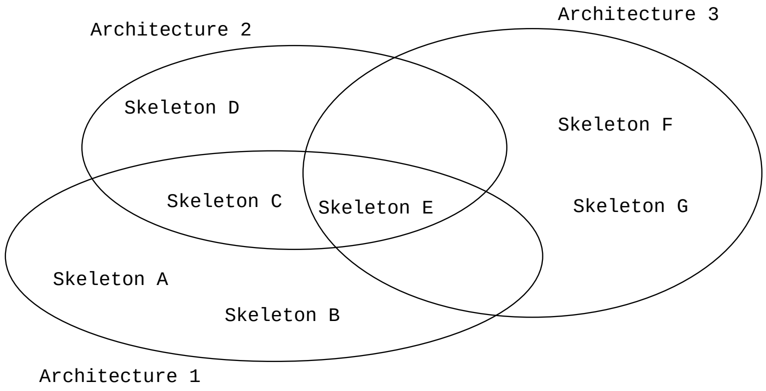


Figure 2.1: Applicability of Skeletons to Architectures

- The potential for *architectural independence* is one of the strongest aspects of algorithmic skeletons. In theory, any program written using a skeleton which can be implemented on more than one parallel machine instantly becomes transportable between those machines. The generality of this is open to debate, however, it may be that not all skeletons are applicable to all architectures which means that full architectural independence would not be possible, this situation is illustrated in Figure 2.1.

  It is quite often suggested that one of the reasons parallel systems have not been accepted into the computer industry more generally than at present, is because parallel programs are not easily transportable between architectures [104]. Parallel machines are evolving so quickly that no major software vendors are prepared to invest large amounts of programming effort into code which could become redundant in a few years.

- *Code reuse* is also quoted as being a barrier to the development and management of parallel code. Again, the skeletons idea has the potential to provide a good degree of code reuse, in the sense that once a skeleton has been defined and an implementation written (and debugged) the skeleton can be reinstantiated in the future.

- In some instances, it is *constraining the parallelism* which is the major problem. Any given machine only has limited potential for parallel implementation and it is often the case that there are *too many* centres of parallelism within the source program (at different levels of granularity) for them all to be realised on the target architecture.

  Skeletons effectively constrain the amount of parallelism in such programs to that which can usefully be translated to the implementation although there are still many unanswered questions about the granularity of the skeletons themselves. It is certainly true however, that searching for a small number of predefined constructs is a more tractable problem than parallelising arbitrary sequential code.

Finally, there are a number of peripheral issues which have to be considered before embarking upon research into skeletal programming or developing an application using a skeletal approach or a skeletal compiler:

- The *host language* or *mathematical formalism* obviously places limits upon what can be achieved. The concept of a set of predefined parallel constructs with efficient implementations is equally applicable to imperative, logic, functional or object-oriented programming paradigms. Functional languages are the most common due to their tractability to formal analysis and the ease of defining high-level abstractions.

- Since it cannot be guaranteed that the programmer or an automated compiler will provide a program with an efficient parallelisation, *program transformation* is often cited as a means of obtaining an efficient solution from an inefficient one by applying semantics preserving transformations. Unfortunately this is an area of research in its own right since program transformation has as many complexities and difficulties associated with it as does algorithmic skeletons. Attempts have been made (and are probably worthwhile) for small transformation systems, restricted either by the size of the program that can be handled or by the number of transformations considered.

- To allow the skeletons in a program to be most efficiently employed it is almost essential that some kind of *cost* or *performance model* be associated with each skeleton. This is to allow the compiler (or programmer) to decide where skeletal parallelism is appropriate and to decide between several competing sources of parallelism within one program. Yet again, however, this requires a major design and implementation effort although performance models for skeletal compilers have been extensively investigated [105, 8].

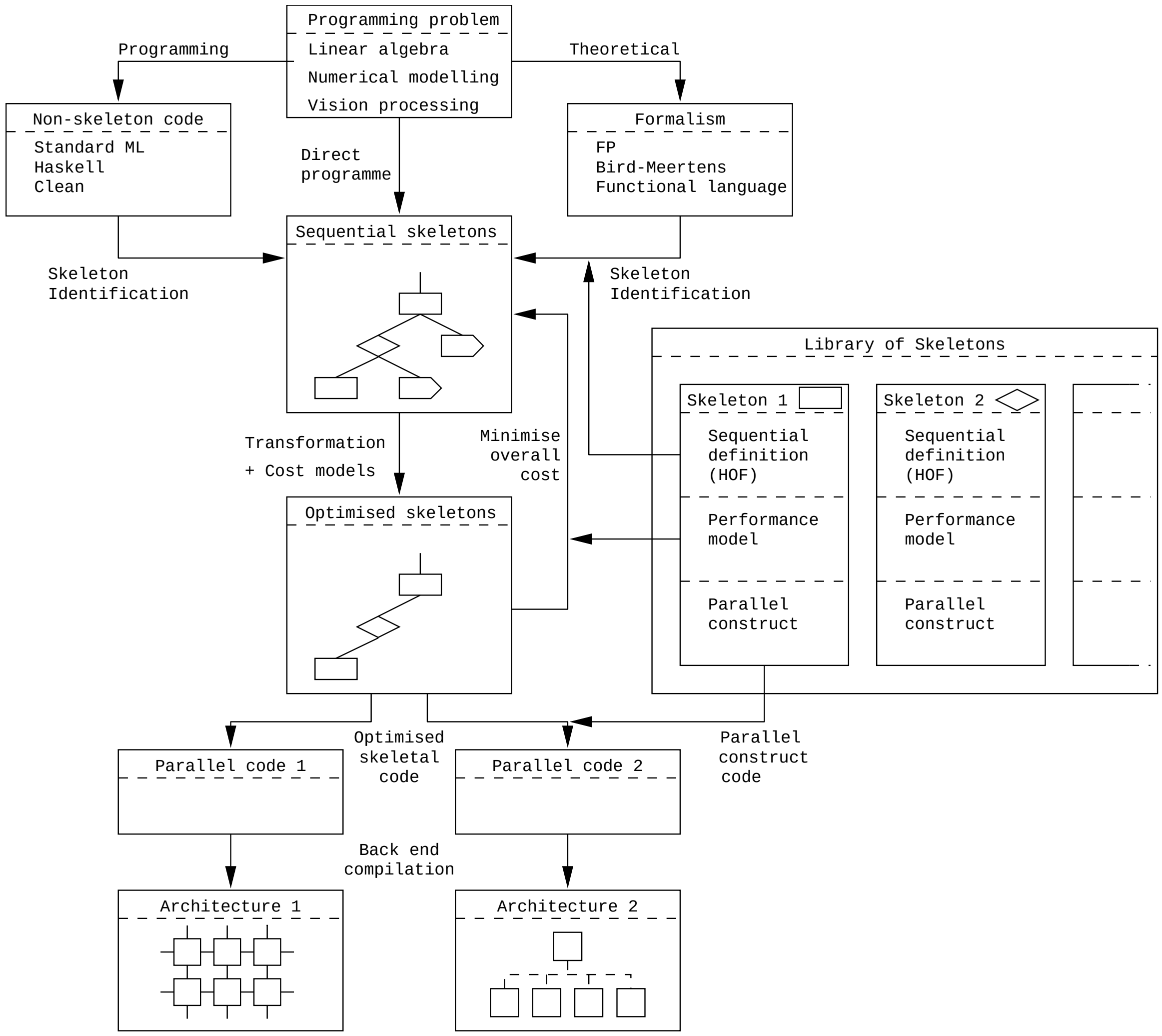


Figure 2.2: Components of a Skeleton-based System

Figure 2.2 outlines a possible generic scheme for using algorithmic skeletons to develop parallel programs. The programming problem can be tackled in a number of different ways;

skeletons can be written in a suitable host language as higher order functions or some equivalent construct, the problem could be analysed theoretically leading to a formally derived scheme which can be implemented using a suitable host language or skeleton-like constructs can be used directly in the target system. Once a skeletal solution is derived, a library of skeletal definitions would be called upon to translate this solution onto the target architecture. This would include a performance model for the skeleton which could be used to iteratively improve the skeletal solution by optimising for the target architecture, this may require excluding skeletons which do not lead to adequate parallelism or using a second library of transformations to generate new solutions. Finally, the skeletal library would provide the common code for the implementation which would be instantiated with the code provided for the particular solution generated by the optimisation routine.

2.3.2 Current Research

The skeletons idea is still relatively recent, although work on parallel abstraction mechanisms predates the name "algorithmic skeletons" (this was coined by Cole in 1989 [11]). The work is not mature enough for any review articles to be available at present and much of the research effort is fragmented, with no co-ordinated drive into particular areas. The following is therefore merely a summary of current papers in areas relating to skeletons, organised roughly according to a simple taxonomy. It is difficult to pigeon-hole some of the effort in skeletons work, there is sometimes a great deal of overlap between different approaches so the categorisations are only very loosely defined.

Based on Data Properties

There are a number of techniques for deducing parallelism from a program based on the properties of specific datatypes. Most of these are not explicitly skeletons-based but they generally share the idea of reusable functions implemented in parallel.

Nishimura [106] describes a typed lambda calculus which integrates a parallel evaluation mechanism into the operational semantics. The exploitation of data parallel programming from array-based data is extended by an evaluation scheme for recursively defined data. This successively transforms recursive definitions so that each recursion can be evaluated in parallel (similar to a "map"-like operation on lists). A typed lambda calculus is then founded on this idea (Data Parallel λ-Calculus) which is linked to an implementation scheme and a parallel operational semantics. There are some serious restrictions on the types of functions that the technique is guaranteed to work on and there is no working implementation of a compiler-based on the methods proposed, as yet.

Südholt [107] attempts to place the skeletons methodology on a formal basis using *data distribution algebras*. The work is algebraic in nature and instead of hiding the data distribution within the parallel implementation (as in most implementations), the programmer is given explicit control using *covers*. In this work a skeleton is defined as the combination of; 1) a data distribution, 2) a higher-order function in a sequential language and 3) a set of *cost measures* associated with the HOF to be used in predicting the performance in a parallel implementation.

A data distribution algebra is defined as a *cover* which is a set of data distributions, closed with respect to program transformation. It is claimed that the derivation of a parallel implementation directly from source language constructs has limitations and that transforming into intermediate skeletons (a form of intermediate representation) alleviates this. Another point made is that when skeletons are combined (or composed) the distribution of data is lost, which Südholt calls the *skeleton composition problem*. Using an algebraic representation allows a new distribution to be computed automatically from the existing skeletons.

Maaßen [108] proposes extending the application of list-based higher order functions to other data structures. The objective seems to be to minimise recursion in favour of iterative or divide and conquer style implementations. Most of this paper concerns the development of functional programs but the ultimate objective is parallel implementation. It is concluded that using the new data structures allows concise programs to be written and that most of the HOFs and data structures have efficient parallel implementations. Tree-based implementations are suggested as the most efficient.

Shafarenko [109] proposes the use of symmetry as a means of controlling access to array structures in data parallel programs. This work essentially combines the type theoretic approach with a type classification based on symmetry hierarchies, formalised as a lattice of access types. Symmetry classification is used in a subtyping system with coercion in the case of less general supertypes. A set of four skeletons is suggested as being sufficient for distributed data parallel programming, `Map` which is the familiar map function from functional languages (this work is essentially imperative in nature), `Juxtapose` (symbolised as `[[]]`) which is the `zip` operator, `Sel` is the selection operation and can be expressed as `Map` but is treated separately for efficiency and parameterisation reasons, and `Cat` which is concatenation generalised to handle multidimensional arrays. There is some similarity to Südholt's work in that skeletons are viewed as intermediates in an optimising compiler and are used to control data access.

Jay [110, 111] claims that the shape of datatypes can be used to devise communication strategies and load balancing schemes for parallel programming. The *shape* of a datatype is its size, for instance the shape of a list is the number of elements in the list and the shape of 2D array is a tuple consisting of the number of rows and the number of columns. Some functions are shape polymorphic, for example the `map` function could be applied to a tree or an array containing the same data so that the data is fixed but the shape can vary. The shape of a datatype is identified with a particular parallel computation strategy allowing reuse of parallel code by spotting other functions with the same shape, shapes would be handled much in the manner of types in programming languages although whether this should be explicit or implicit is not stated. This idea is not explored in any practical sense and it seems likely that there are severe restrictions on its use; many functions are not shapely, however it is suggested that shape theory allows some of the compromises made in generating parallel programs to be investigated.

More recently, this work has been developed into a language called `FISh` based on shapely types [112]. This is an array processing language combining ideas from both functional and imperative styles and has an associated semantics [113] and cost calculus [114].

Based on Formalisms

There are a number of mathematically-based formalisms that are used to reason about programs, particularly functional programs, and these have been used as vehicles for developing the skeletons idea. Functional languages have been used in this capacity for a very long time and also parallel functional languages have been used as basis for implementing skeletons-based programs.

Backus' FP [115, 116] was originally proposed as a means of reasoning about functional programs and proposes that functional programming should become a mainstream computing paradigm as an alternative to imperative programming. Being a condensed and symbolic language which is suitable for manipulating very high order functions makes it potentially useful for defining and manipulating skeletons.

Harrison [117] describes a transformation system for deducing the presence and the granularity of parallelism in functional programs specified in FP augmented with some APL style syntax. The main idea in this paper is to take a uniform source program and apply this to various types of parallel hardware using specific sets of transformations tailored to each type of hardware; pipeline and divide-and-conquer parallelism are discussed. Practical implementations are only mentioned and it is suggested that performance modelling and complexity analysis in combination with some means of mapping onto target platforms would be required.

Functional	Symbol	Definition
Composition	$\circ$	$f \circ g : x = f(g\ x)$
Constant	$\bar{s}$	$\bar{s} : x = s$
Constructor	[...]	$[f_1, ..., f_n] : x = [f_1(x), ..., f_n(x)]$
Insert	/	$/f :< x_1, ..., x_n > = f(x_1, f(x_2, ...f(x_{n-1}, x_n)))$
Apply to all	α	$\alpha f :< x_1, ..., x_n > = < f(x_1), ..., f(x_n) >$

Table 2.3: FP Functional Forms

Danelutto and Pelagatti [118, 119] also describe a skeletal version of FP (skeletons are called *templates* in their terminology), in this case concentrating on defining parallel constructs for distributed MIMD systems for the FP functionals. See Table 2.3 for some of the functionals treated as skeletons in this paper. In particular, the conditions for the application of the functionals and the parameterisation of the templates are discussed. Several templates are described for each functional, for instance the `apply-to-all` functional α and `insert` functional / both have three templates. A compiler is then described which consists of a library of templates and a library of valid transformations that can be applied to optimise parallel performance. The transformation library would also include the conditions for application of the transformations and the cost (in parallel performance) of applying them. The compiler would then optimise the program for parallel performance by applying the transformations, deduce which templates were the most useful for parallel implementation and then generate abstract code for the target machine. Interesting points are that separate optimisations are proposed for single run and continuous operation and a "dummy" farm functional is provided as a compiler intermediate to define processor farming. One potential limitation is that recursive functions are not allowed.

The Bird-Meertens Formalism [120, 121] is a method for reasoning about functional languages, based mostly on Backus' FP [115] but with some additional APL [122] style syntax. The original idea was to derive efficient operational solutions from declarative specifications for sequential systems but it has been observed [5] that the BMF methodology is also applicable to deducing parallel programs using the skeletons idea in conjunction with a cost calculus [8].

Skillicorn [5] gives a set of criteria for a successful parallel computation model, these are discussed from the point of view of programmers and implementors, between whom the parallel model forms a bridge:

Programmers require:

- *Architectural independence* of both the programming language and the methodology used to construct programs. This requires a high degree of generality in the model.

- *Congruence* meaning that the real processing cost should be a property of the model, allowing performance on any architecture to be predicted from the model.
- *Descriptively simple* so that humans can successfully manage programs of high complexity, this is necessary to give scalability to parallel programs (more processors require a greater amount of intellectual effort to manage).
- *A migration path* from one architecture to another so that programs do not have to be completely redesigned when an architecture is upgraded.

Implementors require:

- *Efficient implementations* on a wide range of architectures such that the performance of the implementation is asymptotically no worse than that of the (best) abstract model.
- *Transformation schemes* between target architectures which preserve efficiency and scalability.
- *A migration path* between architectures so that efficient implementations can be recompiled in an architecture dependent manner.

A large number of parallel models are reviewed, classified according to expressibility and degree of abstraction. It is concluded that the most promising type of models are data parallel models since they satisfy all the above criteria but that there are questions about which set of primitives should be included; BMF is proposed as a sufficiently powerful and general method of reasoning about data parallel models. It is suggested that BMF meets all the criteria given above, the main properties of BMF that lead to this statement are; the equational completeness guaranteeing a path between any two equivalent source or target solutions, the correctness of equational reasoning which allows correct implementations to be generated from correct specifications, the locality of communications induced by the locality of data constructors (for all of the constructors that have been considered so far), the encapsulation of communication/synchronisation issues in higher order functions, the natural decomposition over data types matching equivalent parallel topologies with further equational reasoning for migration of implementations.

Cole [123] examines the list homomorphism lemma [120]:

Lemma 2.1 The (List) Homomorphism Lemma:
A function h is a homomorphism with respect to $+\!\!+$ (the list concatenation operator)

iff $h \;=\; (\oplus/) \cdot (f*)$

for some operator $\oplus$ *and function* f *where* $/$ *is list reduction and* $*$ *is a list map.*

A homomorphism is a mapping from the range of one function to the domain of another such that applying the homomorphism mapping to the result of one function is the same as applying the homomorphism to arguments of the other function, this can be summarised:

$$\begin{aligned} h\ \mathbf{1}_{\otimes} &= \mathbf{1}_{\oplus} \\ h\ (x\ \otimes\ y) &= (h\ x)\ \oplus\ (h\ y) \end{aligned} \tag{2.2}$$

Note that homomorphisms preserve identities ($\mathbf{1}_{\otimes}$ and $\mathbf{1}_{\oplus}$) and if the homomorphism mapping is *one to one* and *onto* then technically it is an *isomorphism*.

Lemma 2.1 is important in that it specifies an entire class of (list processing) functions which can be implemented in parallel since any function proven to be a homomorphism can be broken down into the composition of a `map` ($*$) and an associative `reduce` ($/$) which can be implemented efficiently in parallel. Actually, any homomorphism itself contains useful parallelism since it defines a divide and conquer decomposition of the problem (($h\ x$) and ($h\ y$) can be implemented concurrently), however, for lists splitting is not a unit time operation so Lemma 2.1 is more useful. Homomorphisms have been proposed as a means of developing parallel programs [124]. Cole [123] addresses the problem of the fact that many functions are not homomorphisms and gives an example of a function (the "maximum segment sum" problem) which he rephrases with some additional variables (which he dubs "computational baggage") into a homomorphism. This is reminiscent of the use of *carry* variables that are used in numerical routines to allow parallel decomposition [4].

Grant-Duff and Harrison [125] also investigate the homomorphism lemma (Lemma 2.1) and in this paper propose proving functions to be homomorphisms by equational reasoning and matching with the homomorphism definition:

$$\begin{aligned} f\ [] &= id_{\oplus} \\ f\ ([\bullet]\ x) &= i\ x \\ f\ (xs\ +\!\!+\ ys) &= f\ xs\ \oplus\ f\ ys \end{aligned} \tag{2.3}$$

such that f is a homomorphism from one monoid $([\alpha], +\!\!+, [], [\bullet])$ over α to an unspecified monoid $(\beta, \oplus, id_{\oplus}, i)$ if Equations 2.3 are satisfied. In particular the last equation states that f is $+\!\!+ \rightarrow \oplus$ promotable. More generally, the *Homomorphism Promotion lemma*:

Lemma 2.2 The Homomorphism Promotion Lemma [125]:

A function $h : [\alpha] \rightarrow \beta$ *is a homomorphism between monoids* $([\alpha], +\!\!+, [], [\bullet])$ *and* $(\beta, \oplus, id_{\oplus}, i)$ *iff*

$$h \cdot +\!\!+/ \;=\; \oplus/ \cdot h* \tag{2.4}$$

can be used in conjunction with Lemma 2.1 to derive the *Homomorphism Partition Rule*:

$$(\oplus/ \cdot f*) \cdot +\!\!+/ \;=\; \oplus/ \cdot (\oplus/ \cdot f*)* \tag{2.5}$$

which shows how divide and conquer parallelism arises from the application of the homomorphism (the RHS can be applied concurrently to both halves of a list). Some existing skeletons such as `farm` and `ramp` are shown to be based on homomorphisms and a datatype transformation scheme `MT` is derived from binary trees to lists giving an equivalent scheme for SIMD parallelism, this is the least fixed point of:

$$g \;=\; p \rightarrow h \cdot var\,;\, \oplus/ \cdot g* \cdot split \tag{2.6}$$

where p is a predicate distinguishing nodes and leaves, var is the leaf element extractor and $split$ splits a node into a list from each branch. Note that `MT` is itself a skeleton.

Some simple examples are developed, the procedure is to first identify the types of the data elements by type inference, followed by some equational reasoning leading to the direct application of the homomorphism lemma giving divide and conquer parallelism. Finally, the `MT` datatype transformation is applied giving an SIMD form. A more complex example of fractal decompression is developed but no parallel implementations are given. The similarity between the several examples given might suggest the possibility of automated reasoning although this process will not be guaranteed to work on arbitrarily defined functions.

Skillicorn and others [5] have been quite vigorous in their support of BMF as a means of generating parallel implementations, particularly with regard to architectural independence and insulating programmers from the nuts and bolts of parallel architectures.

Kumar and Skillicorn [126] give an algebra of operations called *compound list operations* which operate upon pairs of elements selected from lists. These can be used to process a wide range of algorithms including sorting and FFTs. The algebra allows spotting of inefficiencies and has efficient mappings to hypercubes and 2D meshes. This work is not specifically BMF-based but bears some resemblance to parallel BMF work in that equational reasoning is used in conjunction with mapping higher order constructs onto the hardware.

Cai and Skillicorn [127] give an $O(\log n)$ scheme for implementing a general class of recurrence relations characterised by:

$$E \quad = \quad \oplus \not\rightarrow ([b_0, b_1, \cdots, b_n] \bigvee_{\odot} tails[a_0, a_1, \cdots, a_n]) \tag{2.7}$$

where:

$$b \odot x = \otimes \not\rightarrow_b \; x$$

and $\not\rightarrow_b$ is reduce to the right with base value b. The key element is to redefine the expression, assuming $\otimes$ distributes backwards through $\oplus$ and that both are associative, in terms of a reduction:

$$x \otimes /_{b_0} \oplus y \quad = \quad b_0 \otimes \pi_1 A \oplus \pi_2 A \tag{2.8}$$

in which; $A = \oslash/(x \bigvee_{\odot} y)$, $\odot$ and $\oslash$ are unit time operations and $\pi_1(a,b) = a$ and $\pi_2(a,b) = b$. This gives a divide and conquer implementation with more than one A being computed in parallel at each stage. A similar scheme is given for computing all the intermediate results, in terms of a prefix operation which also gives $O(\log n)$ complexity. A sample application is the maximum segment sum problem which is a straight instantiation of the above scheme.

Skillicorn and Cai [8] give a cost calculus suitable for their BMF equational derivations. These are based on coarse parameters such as data size and numbers of processors and the cost of the basic functions such as map are easily derivable, for instance the map function implemented in parallel has cost:

$$\begin{aligned} t_n(f *_n) &= t_n(PAR_n f) \\ &= t_1(f) \end{aligned} \tag{2.9}$$

where $t_n(f)$ is the cost of executing f on n processors, ($PAR_p f$ denotes p parallel applications of f). The costs of the basic operations `map` and `reduce` are derived and then used to derive the cost for more complex operations such as `prefix` and `inits`. These are then generalised to situations where the number of processors does not match the number of list elements, for instance, on p processors `map` has cost:

$$t_p(f*_n) = t_1(REP_{n/p} f_1) \quad (2.10)$$
$$= \frac{n}{p} \cdot t_1(f_1)$$

($REP_m f$ denotes m sequential applications of f). To derive the more complex cases, the functions are annotated with shape information, for instance:

$$^{[n,p]}f*^{[n,m]}$$

specifies that f takes a list of length n, with arguments of size p and produces a list of n elements of size m. The notation $[n, m, p]$ means a list of length n, maximum element size m and maximum sublist element size p, the maximum implies that the costs are overestimates.

Once the costs of the basic operations are derived, some typical BMF identities have their costs on the left and right hand sides evaluated. The identities are then annotated to indicate which direction (if any) results in a reduction of cost, for instance:

$$f * \cdot +\!\!+/ \rightleftharpoons +\!\!+ / \cdot f * * \quad (2.11)$$
$$\oplus/ \cdot +\!\!+ / \rightharpoonup \oplus/ \cdot (\oplus/)*$$

Although the treatment here seems general enough to cope with most situations, the analysis is very static, data dependencies or irregular data would require an order of magnitude more complex analysis. The calculus was, however, able to derive a new solution to the maximum segment sum problem.

Skillicorn [128] extends existing results for lists to binary and rose trees. A binary tree is defined by two constructors:

$$bleaf : A \to T(A, B) \quad (2.12)$$
$$bjoin : T(A, B) \times B \times T(A, B) \to T(A, B)$$

where A is the base type (of leaf values) and B is the type of a value stored at the $bjoin$ node (often these will be the same). A rose tree is defined similarly:

$$rtleaf : A \to RT(A, B) \quad (2.13)$$
$$rtjoin : B \times RT(A, B)* \to RT(A, B)$$

where $RT(A, B)*$ denotes a list of subtrees (rose trees). The analysis proceeds based upon homomorphisms defined for both these types of trees, for instance a homomorphism h for rose trees can be defined by a pair of functions $p_1 : A \to P$ and $p_2 : B \times P* \to P$:

$$
\begin{aligned}
h(p_1,p_2) \quad (rtleaf\ a) &= p_1(a) \\
h(p_1,p_2) \quad (rtjoin\ a\ l) &= p_2(a, h(p_1,p_2) * l)
\end{aligned} \tag{2.14}
$$

for instance, a `map` operation for rose trees could be defined by:

$$
\begin{aligned}
p_1\ f_1\ a &= rtleaf\ f_1(a) \\
p_2\ f_2\ a\ l &= rtjoin\ (f_2\ a, l)
\end{aligned} \tag{2.15}
$$

The basic homomorphisms studied are:

- `map`
- `reduce`
- `upwards accumulation`
- `downwards accumulation`

Implementation schemes are devised for cases where there is one node per processor and when there are more nodes than processors and several different schemes are considered in each case; the intuitive recursive scheme and also schemes based on *tree contraction* [129, 130, 131] which is a scheme for reducing the effect of the longest path from root to leaf (but places restrictions on the p_2 function). When there are fewer processors than nodes, there is a complication in that it is much harder to split a tree than a list (for example) and most of the effort here goes into partitioning the tree into subtrees. The same general method that was used for lists is applied, however, the identity:

$$dist_p \cdot flatten = id$$

is inserted between all of the functional elements in a given program and the *promotion lemmas* are used to shift the $flatten$ operations to the left (later in execution), cancelling with $dist_p$ where possible. The cost calculus is also extended to deal with the new results, this is a straight extension from the costs for lists. It is pointed out that many operations on text are succinctly expressed using tree structures.

Gorlatch and Lengauer [132] give a transformation scheme from a functional divide and conquer specification to an SPMD (Single Program Multiple Data) parallel implementation. The class of functions handled by this method is a generalisation of that found in [133] in that mutually recursive functions of the form $f = (f_1, \cdots, f_n)$ can be handled, where the f_i are defined as:

$$f_i(x) = if\ p_i(x)\ then\ b_i(x)\ else\ E_i(g_i, f, x) \quad (i = 1, \cdots, n) \tag{2.16}$$

where g_i is the non-recursive *auxiliary* function of f_i. There are also some slight restrictions on the form of the E_i functions.

A general schema is given for transforming the functional specification into a parallel functional implementation schema, this is is performed by instantiating the general result derived in the paper. The final imperative SPMD implementation is generated by implementing a derived function F (which combines the functionality of the f_i functions) uniformly on a message passing tree structured network. An interesting point is that the communications structure is abstracted separately from the instantiation code in the form of an evaluation tree with processors mapped to nodes and an abstraction function (div) which can be instantiated to derive the parallel schema from the initial specification (the abstraction function is used to derive the uniform function F which is implemented on the processor nodes).

It is claimed that the resulting target program can adapt itself to the number of processors available but there are a number of efficiency issues which are not addressed; controlling the amount of parallelism using the recursion depth of the div function, coping with communication trees which are not uniform, eliminating redundant computation (which is inherent in the schema) and dynamic load balancing (in the current scheme it is static). This work, however, is more general than other similar efforts at analysing divide and conquer parallelism [133] , for instance the scheme is not restricted to list processing (the BMF list functions are only used to describe the pattern of parallel computation, not the base data type).

Gorlatch [134] also presents a case study showing the relationship between transformational derivation and parallel programming methodology. The example chosen is polynomial multiplication and proceeds by representing the basic problem specification in BMF notation (polynomials are represented by lists of coefficients). Some transformational identities are derived which are used to develop a generic scheme for parallel implementation which consists of a distribution phase, a computation phase and a combine phase. A parallel implementation on 2D arrays of processors is suggested by the central computation (a $map^2 \otimes$ operation where $\otimes$ is multiplication of components of the computational decomposition). Several design decisions are made which leads to an asymptotically optimal solution in terms of both time and processor counts which, it is claimed, can outperform other algorithms for this problem although some further transformational development of the combine phase is required to reach the optimal solution. This work does, however, illustrate the role that methodologies like BMF might play in parallel system development. Another interesting point is that the central scheme is parameterised with both the number of processors and the size of the data.

General functional-theoretic work has also been the basis for some theoretical analysis of skeletons work. The objectives are essentially similar to that of more symbolic formalisms but the notation tends to be less condensed and possibly less readable as a result.

Axford and Joy [135] propose *append lists* as the basis for parallel programming using functional languages as opposed to the traditional *cons lists.* Cons lists are based on the list constructor :: which takes an element and a list and makes a new list with the element at its head. This is a bijective function which allows pattern matching to proceed deterministically in a functional language:

$$\begin{aligned} map\ f\ [] &= [] \\ map\ f\ h :: t &= (f\ h) :: (map\ f\ t) \end{aligned} \tag{2.17}$$

Append lists on the other hand are defined constructively using the append operation $+\!\!+$:

$$\begin{aligned} map\ f\ [] &= [] \\ map\ f\ [h] &= [f\ h] \\ map\ f\ l_1 +\!\!+ l_2 &= (map\ f\ l_1) +\!\!+ (map\ f\ l_2) \end{aligned} \tag{2.18}$$

The append constructor is not bijective since a list of more than two elements can be split at more than one place. This, it is claimed would give a parallelising compiler more freedom to decide upon data decompositions. A model of list processing is introduced based on six axioms; lists are stored as binary trees and a set of higher order functions are developed as the programming basis. These include the usual `map`, `filter` and `reduce` with a few more specialised ones such as `reverse` and `transpose` which transposes the rows and columns in a list of lists (in fact the full set of list functions in the standard Haskell prelude are defined [103]). A complexity analysis of theoretical performance on a CRCW PRAM model is presented and the complexities of cons and append lists compared. Three applications are developed; quicksort, ray tracing and convex hull calculation.

de Guzmán et al. [133] describe a method for transforming a class of divide and conquer algorithms into pipelines. The idea is to allow this type of algorithm to be run on static MIMD networks (divide and conquer algorithms cannot easily be implemented on systems with fixed numbers of processors and links). Static MIMD arrangements are more efficient for pipelines or process networks. Divide and conquer functions are first transformed into an iterative form where the number of computations increases at each divide step. The pipeline implementation is then derived by datatype transformations. A number of implementational problems are pointed out; control of granularity and synchronisation between pipeline stages are left as open questions. While this work is not explicitly skeletal in nature, it addresses some of the issues faced by a skeletons-based compiler, that of transforming a specific set of source constructs into parallel equivalents.

Boiten et al. [136] also view skeletons as intermediates for optimising compilers. A set of low-level skeletons with implementations on SIMD linear processor arrays is presented and an example of the trapezoidal rule is developed, including a very simple translation to parallel code by a straight transliteration of the transformed skeleton code. The usual `map`, `zipw` (≡ `zipwith`) skeletons are defined along with some architecture specific skeletons such as `shiftr` which rotates values round and `update` which updates a value on an array.

Some interesting observations are made about the skeletons concept. Firstly, it is stated that; "particular architectures are characterised by the sets of skeletons that represent the efficient computation structures on those architectures." This suggests that architectural issues may yet have an influence upon the development of sequential solutions to programming problems. Another point is that a comparison is made with the more conventional parallel functional programming route. This involves transformation to iterative solutions followed by parallelisation of iteration. It is pointed out that both of these activities are hard. On the other hand the skeletal route involves transformation from functional solutions to skeletons which already have efficient codings. The transformation is hard but the codings are predefined and reusable. Finally, a distinction is made between program skeletons which are defined as *"higher-order functions that can be expressed in terms of the available skeletons for some architecture"* and architecture skeletons which are defined as *"a higher-order function that has a straightforward interpretation as a typical elementary computation on that architecture"*. Architecture skeletons are lower level than program skeletons but again there is the suggestion that the sets of skeletons that apply to each architecture may only partially overlap, this is depicted in Figure 2.1.

Pepper [4] describes a methodology for developing provably correct parallel implementations from problem specifications. The method (which Pepper calls "deduction-oriented") is a hybrid between proving that an existing implementation conforms to a specification and the derivation of the implementation by correct transformations. The methodology proceeds by deriving a recursive sequential solution to the problem specification in the traditional functional style (the Bird-Meertens notation is used). A *data partitioning* is then performed on the sequential solution leading to a distributed solution which is mapped onto a network of processors. Note that the same comparison with the functional approach is made as in Boiten [136]. An example application is illustrated using Bird-Meertens style analysis but it is not indicated how automated this process could be, the example requires a fair amount of intuition from the programmer. Transformation rules for deriving parallel equivalents of sequential solutions are skeletons-based. The usual skeletons are defined; `apply-to-all`, `zip` and `reduce` along with variations on these, for instance `apply-to-all` with an additional higher order function. Table 2.4 lists the skeletons for which mappings to processor networks in terms of transformation rules are given. Several applications are given but the emphasis here seems to be on the correctness of the implementation rather than its efficiency.

Rule	Definition	Description
1	$h\ *$	`apply-to-all`
2	$(h\ x)\ *\ S$	`apply-to-all` with higher-order function
3	$(h/)\ *\ (inits\ S)$	`reduce` mapped over initial segments
4	$(h\ x)\ *\ (e \wedge (F\ S))$	map over recursion (e is identity)
5	$(g\ x)\ \wedge\ (F\ (h \circ g\ x)\ *\ S)$	map over recursion arguments

Table 2.4: Pepper's Skeletons

Chin [137] gives a method for deducing divide and conquer parallelism from linear self-recursive functions. The method involves unfolding the recursive function twice into what the author calls "pre-parallel skeletal forms". The computational structure of these forms is then extracted (giving a *parallel template*) and the two expressions are then matched by generalising at points where mismatches occur (this requires the introduction of unknown functions and so is a second-order generalisation), similar to pattern matching algorithms in machine learning. A small set of heuristics are used to guide the selection of transformations for generating the pre-parallel forms, for instance recursion variables are kept as shallow as possible in the expressions. The forms of the unknown functions are then deduced by inductive derivation and unification of both sides of the equations derived from the previous step. Finally, some tupling is required [138] to improve the efficiency of the parallel solution. The list *homomorphisms* often found in the Bird-Meertens formalism [120] turn out to be a subclass of the functions that can be handled by this method, however, several other classes of functions can be dealt with, including functions with accumulative parameters and functions with multiple or nested recursion and tail recursive functions. Although this method looks very powerful and general, it is very dependent upon the properties of the auxiliary operators and functions, most of the examples use addition (+) and concatenation

(#) which are associative and have identities. Again there is also the question of how automated the procedure could be made but this method is complementary to the constructive use of skeletons in that it effectively deduces the skeletons from a specific class of functions.

Category theory [139, 140] has also been proposed as a means of developing and controlling parallel programs [124, 141]. The problem with existing data type models is the *ad hoc* manner in which the basic operations are chosen. This is why *categorical data types* are preferred over abstract data types since equational completeness is guaranteed by CDTs. Although this work is not explicitly skeletons-based, there is a close relationship between data parallel skeletons and some of the parallel implementations suggested by results from category theory.

Banger and Skillicorn [142, 143] give a categorical treatment of array theory, this ties up some of the loose ends associated with existing array theory and suggests straightforward parallel implementations. Again the emphasis seems to be on finding array homomorphisms and using their properties to derive the parallel component of a given program. Plenty of applications are given but no performance figures for real implementations are presented.

Parallel Functional Languages

Parallel functional languages have also been used as vehicles for implementing skeletons-related ideas. Again the idea is to constrain the implicit parallelism in a functional implementation from swamping the machine with progressively finer parallelism. This eases activities such as mapping tasks to processors, deciding which centres of parallelism are viable and deciding upon patterns of communication/synchronisation. Again, the relationship with "pure" skeletons work is only conceptual but existing work on both parallel functional languages and skeletal parallelism are highly relevant to each other.

Wray et al. [6] develop vector and matrix programs in SML and use an automatic transformation scheme to generate parallel Fortran implementations. Centres of parallelism are parallel vector and matrix functions which are unfolded and expressed in terms of the primitive operation *generate*:

$$generate([1:n,1:n], fun(i,j) \;=>\; a(i,j)+b(i,j))$$

which consists of shape information and a function to be carried out on each value. The transformation scheme ensures that all `generate` functions appear on the RHS of assignments and specialised versions of `generate` are used to handle data locality, for instance $a(i+1, j) + b(i, j-1)$ may require data communications. This is handled by the transformation scheme which draws on "skeletal" code implemented on the target architecture. Details are a bit thin on the ground, here, but this is a good example of treating traditional parallel vector/array operations using a skeletal approach.

Kesseler [9] defines skeletons in the parallel functional language Concurrent Clean [144]. This is a lazy functional language with a small number of annotations to define parallelism; `{I}` creates a new process on the same processor, `{P}` starts a new process on the next available processor and `{P at` *processor* `}` starts a process on processor with the given processor-id. Note that arguments to remote processes are always evaluated before transmission, strict arguments are evaluated prior to forking the new process and lazy arguments are evaluated lazily only when the remote process needs the arguments. Communications other than those implied by these annotations are handled implicitly by graph copying. Only normal forms are ever directly transmitted; when a remote process requires a graph the graph is evaluated (lazily) to root normal form (RNF). This is called "Lazy Normal Form Copying" and it is claimed that this allows the programmer to predict exactly what will get copied. Apart from these annotations, some additional functions are provided which allow remote application of a function to its arguments and some low-level functions for accessing remote variables and copy-stopping.

Within this framework several skeletons have been defined. A data parallel `Map3` higher order function is defined which applies a function `f` to three arguments which are lists of lists of submatrices, returning a new double list of matrices. This is not parallel but defines a data decomposition which is utilised by an eager `Fold` skeleton which applies a function to remote arguments and then combines them with another function. These functions are used to implement a parallel matrix multiplication algorithm (known as Gentleman's algorithm [145]) and performance results are good on a transputer implementation which compares well with other parallel functional implementations of this algorithm, for instance DPFL [146]. A second example is also developed; plotting images in parallel, which is I/O intensive. This uses a similar structure to the matrix example, images are distributed like matrices and the same `map/fold` description is used but with some low-level differences. Again, parallel performance is acceptable, a ray tracing example is compared with SkelML [40] and has faster runtimes but similar speedup figures. (Clean is compiled to transputer assembly but SkelML uses Occam2 as an intermediate). The third example is different, this uses stream processing in a tree structured divide and conquer skeleton called `StreamTree`. This is used to implement a parallel mergesort algorithm but performance here is a bit disappointing. This is probably due to the simple buffering mechanism employed which introduces delays.

An interesting point made is that skeletons could be combined, for instance combining the data parallel skeletons with the stream processing skeleton. This is a potentially powerful mechanism but is not developed any further.

Developing skeletons where the host and target languages are the same (and both functional) appears to be a much cleaner and neater way of developing the skeletons idea. This could be particularly important when issues such as performance modelling and program transformation are taken into account, without the impedance mismatch between source and target languages seen in other systems.

Harrison [147] attempts to develop an abstract programming model called the gamma model [148] in the functional language Miranda[1] with transformation into the process network language Caliban [149]. The sequential implementation of the gamma model is transformed into a cyclic network and then annotated using the $\diamond f$ notation of Caliban which defines a network of processes. A performance analysis is based on counting the number of reductions at each node and indicates a considerable increase in the number of reductions required to implement a simple prime number generating example. Subsequent analysis revealed that moving from finer to coarser granularity resulted in a performance improvement giving speedups of about 30%. The conclusions here are quite pessimistic, the performance figures are very poor indeed but it is pointed out that the gamma model is designed for data parallel operations whereas Caliban is more suited to static process networks. An alternative scheme involving divide and conquer operations is proposed.

Sodan and Bock [150] use trace-based profiling to characterise patterns of parallel computation in large scale functional programs. Centres of parallelism are iteration and recursion which can be nested and analysis is based on a tree-structured representation of *parLisp* programs (a parallel extension to Common Lisp) although it is pointed out that the method is equally suitable for ML or C++. The main objectives seem to be mapping strategies and dynamic load balancing (which is difficult to predict using static profiling techniques). Profiling collects a high level of detail and a processing strategy is selected based on a set of heuristic rules and cost functions, for instance if a high variance of granularity is observed then a lower than mean granularity is employed. Parameters such as problem size (for a centre of parallelism), the critical path (which limits performance), depth of leaf nodes and number of processors are measured. This method looks quite powerful and general but the selection of strategies for mapping and load balancing could be difficult to automate.

Hartel et al. [151] describe a toolkit for parallel functional language. Applications are sequentially written in *Intermediate* which is a subset of Miranda. Development is done

[1] Miranda™ is a trademark of Research Software Ltd.

using an interpreter but, once debugged, is compiled into C which is in turn compiled onto the target hardware. Once the sequential program is ready, it is annotated by a transformation tool using a simple skeleton called a *sandwich*. This is a *fork-join* construct which allows more than one argument to be implemented concurrently and corresponds to strict evaluation of arguments at the centre of parallelism (Intermediate is otherwise lazy). For instance:

$$sandwich\ f\ (g\ x_1\ x_2\ \cdots)\ (h\ y_1\ y_2\ \cdots)\cdots$$

allows g and h to be executed concurrently. This is a medium grain form of parallelism and corresponds naturally to divide and conquer implementations. For other types of implementation; *communication lifting* [152] allows transformation to static networks for geometric parallelism and synchronous pipelines and *grain size transformation* [153] allows adjustment of the granularity. Parallel vector operations are supported by defining an auxiliary skeleton using *sandwich*:

$$\begin{array}{lll} parmap & :: & (\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta] \\ parmap\ \ f\,[] & = & [] \\ parmap\ \ f\ (z : zs) & = & sandwich\ (:)(f\ z)\ (parmap\ f\ zs) \end{array} \tag{2.19}$$

which allows $(f\ z)$ and $(parmap\ f\ zs)$ to be executed in parallel.

Once a parallel version exists this can be simulated at *task* level, *instruction* level and *bus cycle* level parallelism. These allow experimentation with both the sources of parallelism and architectural features such as numbers of processors. Finally, the system can output parallel code for a 4-processor Motorola Hypermodule. A substantial example is developed and performance values are given for several versions of the example on the different simulators and for the parallel output.

This technique offers a good compromise between purely hand-generated parallel systems which are efficient but difficult to program and fully automatic systems which usually are suboptimal in terms of performance. As tools such as this are developed they could push back the level at which programmers are required to make design decisions becoming more automatic while still providing the capability of programmer input where performance is paramount.

Herrmann et al. [154, 155] describe a compiler for HDC, a strict subset of the Haskell language. This compiler aims to integrate skeletons into the compilations process at all levels. Skeletons are specified by higher-order functions, the only restrictions on their instantiations are the types of the supplied functions. The divide and conquer skeleton $\mathcal{DC}$ is given special prominence and five versions of this skeleton are provided, optimised for different situations.

Other more common skeletons presented include `map`, `red` (fold), `scan` and `filter`. In addition, a number of low-level skeletons are provided for improved efficiency (including a `while` loop), evaluation control and input/output.

The compiler itself operates by eliminating all non-skeletal higher-order functions. List comprehensions are translated into combinations of parallel skeletons. The code is monomorphized, defunctionalized and features such as mutual recursion are removed by inlining. Once this is done, each function is translated into a directed acylic graph (DAG) to allow sharing, optimisations are performed on the DAGs. From this, abstract code is generated which leads to C code for which skeletons are generated using MPI for communications. A parallel run-time environment is described, this uses an SPMD model, with several different models of data distribution built in. A method of recursively populating skeletons with processors and an associated control mechanism are also mentioned.

Two examples are presented a divide and conquer implementation of Karatsuba's polynomial product [156] and a combination of simpler skeletons to solve Toivonen's frequent set problem [157]. The sequential code is compared with the GHC compiler but no actual parallel results are given.

Botorog and Kuchen [158] describe an imperative C-like language called *Skil* for implementing skeletons. This language has some functional features and a polymorphic type system. Several skeletons for operating on array structures including `array_map`, `array_fold` and various operations for permuting the arrays are given. Two examples, shortest paths in graphs and Gaussian Elimination, are presented. Performance figures are claimed to be much better than for functional skeletons and are claimed to be equal to or better than hand-coded implementations. More recently Richert [159] uses Skil in the implementation of more sophisticated numerical solvers of partial differential equations implemented as multigrid algorithms and proposes several methods of encapsulating the necessary control features. Again, speedups and efficiencies are claimed to be comparable with equivalent hand-coded implementations.

Pure Skeletons

Some skeletal work is not tied to a particular language or formalism. Most of the major contributors to skeletal programming work fall into one or other of the categories in this section, however, in some instances it is the skeletal component that is the object of research interest and the language or formalism is merely a substrate for this endeavour. The distinction is really one of emphasis.

Cole [11] coined the term "algorithmic skeletons". This seems to have been inspired by the observation that many algorithms have a similar structure (for instance divide and conquer) and that higher order functions capture exactly this kind of structure, although it is suggested that the idea applies equally well to imperative "templates". The closest thing to a definition of an algorithmic skeleton is:

> *"We will refer to each suitable higher order function as an "algorithmic skeleton", since it describes the computational skeleton of an algorithm without overspecifying the details."*

although a high degree of freedom of interpretation is specified. An important point that is made is:

> *"the actual specification of each skeleton as seen by the programmer need not (and ideally should not) be explicitly parallel."*

In other words the programmer should be insulated as much as possible from the nuts and bolts of parallel programming. Indeed, a high-level programmer interface is hypothesised in which the programmer is presented with a menu of available skeletons and, when one is chosen, the system responds with the languages and data structures for which that skeleton is applicable leading to an instantiated program in a base language which could be parallelised more or less automatically. Although this seems a bit farfetched it seems to be the panacea for which skeletal programming research is looking for.

The bulk of this work concerns the design and implementation of a number of algorithmic skeletons.

The *fixed degree divide and conquer* skeleton (FDDC) is a general divide and conquer solution to a problem which can be divided up into a constant number of subproblems at each phase. Binary trees are considered first, then for a divide factor of 4 and finally for a general factor. Asymptotic performance analyses are presented for H-tree implementations. Partial trees are also considered in which work is carried out at intermediate nodes which would otherwise have been sent on to leaf nodes. Finally, three applications are considered; the discrete fourier transform, numerical integration and matrix multiplication.

The *iterative combination* skeleton (IC) successively merges objects according to some criterion until no further merges are possible. An implementation on a square grid of processors is considered which utilises a Hamiltonian circuit of the grid. A single tour algorithm where the number of objects matches the number of processors is discussed which involves cycling objects round the circuit and has the disadvantage of transporting (potentially) large objects. A four tour algorithm is given which eases this drawback. A more realistic algorithm for general numbers of objects and processors is considered and broken down into a number of different cases, for instance when there is a surplus or shortage of objects or when there is a dramatic reduction after one iteration. Applications considered are the minimum spanning tree and connected components algorithms and the design of programmable logic arrays.

The *cluster* skeleton is an attempt at re-engineering a skeleton from an existing hardware implementation. This makes use of the observation that rectangular square grids with a Hamiltonian circuit can be divided up into smaller grids also with Hamiltonian circuits and implements a clustering algorithm for arbitrary objects. This is a very efficient skeleton but no applications are given.

Finally, the *task queue* skeleton is more explicitly parallel than the other skeletons discussed and concerns the situation where processing a task on one of a group of objects can generate more tasks to be processed. Again, implementation on a square grid of processors is detailed for a number of different queuing disciplines; stack, FIFO queue, unordered heap and strictly ordered queue. One problem with this skeleton is that it requires access to a shared data structure which is a potential bottleneck that has to be legislated for. The applications mentioned are; one to all shortest paths (a problem from graph theory), LU decomposition of matrices and solving band matrix linear equations.

Despite being the earliest formulation of the skeletons idea the approach adopted in this work is remarkably mature inasmuch as the methods proposed have been applied to a variety of skeletal constructs and have been applied to a (mostly random) selection of problems from general computer science. There are, however, many issues which have arisen in skeletal work which are not addressed here, for instance guidelines for selection of a particular skeleton for a given problem and although architectural independence is a common theme, nearly all implementations are on 2D grids of processors.

Darlington and others [7] have also developed the skeletons idea, in this case using Hope$^+$ as the source language in combination with performance models and program transformation. The target architectures are transputers and a Fujitsu AP1000, both using C as the target language.

Some preliminary results are sketched in [7, 160]. Several skeletons are given as an initial set. PIPE defines a pipeline of processes, FARM defines a `map` function as a processor farm, DC is a general purpose divide and conquer skeleton, RaMP (reduce and map pairs) processes a number of objects such that all inter-object calculations are performed and a value for each object produced and DMPA defines a general dynamic message passing architecture.

Examples are suggested for each of these skeletons, for instance ray tracing and N-body simulation using RaMP, a compiler using PIPE, a mergesort using DC and a database manager using DMPA. An example of what the authors term *application specific skeletons* is also given, implementing constructive solid geometry trees. This is similar to Coles's cluster skeleton [11] in that an existing application is abstracted and parameterised for potential reuse. A simple performance model is developed for the divide and conquer skeleton. Program transformation is also discussed and transformations from RaMP to PIPE or FARM and from DC to PIPE are given.

Darlington *et al.* [161] develop the idea further and provide performance models for the DC, FARM and PIPE skeletons. These are parameterised by average communications and computation costs, for instance the DC skeleton implemented on a binary tree of processors where intermediate nodes also perform work can be run in time:

$$t_{sol_x} = \sum_{i=0}^{\log(p)-1}(t_{div_{x/2^i}} + t_{setup_{x/2^i}} + t_{comb_{x/2^i}}) + t_{seq_{x/2^{\log(p)}}} \quad (2.20)$$

where t_{sol_x} is the time to solve a problem of size x, t_{div_x} and t_{comb_x} are the times to split and combine a problem of size x, t_{setup_x} and t_{comm_x} are the setup and transmission time for communications and t_{seq_x} is the time to solve a problem of size x sequentially.

The FARM and PIPE skeletons are then developed further by instantiating the performance model with values measured from transputer implementations. This requires some additional parameters such as garbage collection times and the times required to flatten and unflatten Hope$^+$ data structures. The ray tracing example is then implemented using the PIPE and FARM transformed RaMP specification. The parallel performance results are then presented for both of these implementations and compared with predicted results. The FARM implementation predictions were accurate but the PIPE implementation predictions were complicated by the garbage collector.

Darlington *et al.* [162] propose a structured parallel programming scheme which they call SPP(X), where the "X" refers to an unspecified base language. This is intended to be based on skeletons at several levels of program specification and compilation. Firstly, a distinction is made between several different types of skeleton; *parallel abstract datatypes* which provide the basic data structures for programming, *configuration skeletons* which abstract commonly used patterns of communication and data distribution and *computation skeletons* which abstract commonly occurring patterns of computation. This seems to have been inspired by work into coordination languages [163, 164]. Secondly, programming is specified at two levels; the *skeleton configuration level* (SCL) which gives the skeletal structure of the program where skeletons can be nested and composed and the *base language level* (BLL) which is the "X" language and consists of fragments of (closed) code which can be called by the SCL. The upper layer is tractable to program transformation and can be manipulated to give an optimal solution for both computational and communications aspects.

A specific instantiation of the SPP(X) scheme is proposed, defining a skeletal system for Fortran called Fortran-S. A *parser/generator* front end will feed into a *transformation* and *optimisation* stage giving an intermediate skeletal form which can be translated into one of three host Fortrans (VPP, HPF and Fortran-90 + (PVM or MPI)). A simple example program of Gauss-Jordan elimination is given which clearly shows the separation between the two layers and the use of separate configuration and computational skeletons. This is one of the first skeletal compilers to be specifically targetted at an imperative language while still retaining the useful properties of program transformation.

This system is developed further in [165]. More details are given for the SCL language and a more substantial application given with performance results. The parallel abstract data type is a parallel array which has type:

`ParArray index` α

where α is the base type and `index` are the indices for subarrays. Several configuration skeletons are shown, for instance the combination of `partition` which defines a decomposition of an array and `align` which keeps coincident data together. These lead to a distribution of an array:

$$\mathtt{distribution\,(f,p)\,(g,q)\,A\,B} \;=\; \mathtt{align\,(p} \circ \mathtt{partition\,f\,A)\,(q} \circ \mathtt{partition\,g\,B)} \qquad (2.21)$$

This type of distribution can be generalised and then instantiated into different decompositions. Redistributions during calculation can be defined in a similar manner.

As well as configuration and computation skeletons, another type of skeleton called an *elementary skeleton* is defined. These are low-level skeletons defining the basic data parallel operations upon which the parallelism in the current implementation is founded, for instance `map`, `fold`, `scan`, `rotate` and `brdcast`. Computation skeletons are also given, these are the `farm` skeleton, the `SPMD` skeleton which defines a general mode of single program multiple data processing and the `iterUntil` skeleton which is a simple iterative calculation.

The example given is parallel matrix multiplication and two algorithms are studied; *row-column-oriented* and *block-oriented*. An SCL program is given for each algorithm and then a generic algorithm is abstracted from the two (all of these use the `SPMD` skeleton). The translation into Fortran-S along with the base level Fortran instantiation is given. Several points have to be made about the base level code; the call semantics are call by value, there are type checks to be taken into account (it is a type error to pass a parallel array to a Fortran construct) and the Fortran routines have to be written without side effects. If these conditions are guaranteed then the translation into parallel Fortran is relatively simple and results for the parallel matrix multiply program on the AP1000 are very encouraging, with a speedup of 70 for 100 processors.

Compared to High Performance Fortran, the Fortran-S system gives a higher degree of abstraction (HPF configurations are relatively static) and more flexibility in terms of program transformation. Several transformations are mentioned but the means of application and the effect on parallel performance are not discussed.

The techniques adopted by SPP(X), particularly the abstraction of communications patterns, are related to similar ideas from more general parallel computing. For instance, abstraction of communication patterns has been suggested for concurrent object-oriented programming [166] or the research into coordination languages [163, 164]. This work suggests that the skeletons idea is applicable to such abstraction, in tandem with the more familiar abstraction of computation patterns.

Two additional examples are given in [167]. The Gauss-Jordan elimination is presented as an SCL program and a sorting algorithm known as *hyperquicksort* is also implemented [2]. The Gauss-Jordan is an instance of the `iterFor` skeleton (a variant of `iterUntil`) and hyperquicksort is an instance of the `SPMD` skeleton. The hyperquicksort algorithm was implemented on the AP1000 in Fortran and MPI. The original SCL program was transformed from a recursive divide and conquer into a linear iterative scheme. This gave a speedup of about 14 for 32 processors.

To [42] presents the first thorough investigation into the combination of algorithmic skeletons where the skeletons can have more than one parallel implementation and parallelism is preserved across skeleton combinations (as opposed to previous attempts which assume a single sequential data stream between skeletal compositions [40, 41]).

The methods described consist of three layers; the *parallel abstract data types* (PADTs) which form the basic data structures, the *primitive skeletons* which are simple computational skeletons operating on the PADTs and *combining skeletons* which take skeletons as arguments and build more complex applications using primitive skeletons as building blocks.

A PADT consists of an aggregate data structure and the parallel operators required to manipulate it, these are the primitive skeletons. Two examples are given; `rlists` (restricted lists) and arrays.

The `rlist` PADT is like a conventional list but lacks explicit constructors and destructors which force sequential order on the elements (they are generated by *unfold* operations, for instance `rUnfold` where element i is generated by applying a function i times (f^i) and `rConstList` which gives a list of identical elements). The usual list operators are defined such as `rMap`, `rFold` (in several varieties) and `rFilter` and it is claimed that defining these functions using restricted lists makes them more flexible for parallel implementation. In addition, other skeletons are defined for reordering elements and for handling divide and conquer style parallelism.

Arrays are also built using unfold operators such as `aUnfold` and `aConst` and have a similar set of primitive skeletons such as `aMap`, `aFolda` and `aScana`. In addition, the `aLNO` (*local neighbourhood operations*) skeleton is defined which allows operations which depend on neighbouring elements in the array to be defined for instance rotation or permutation of elements around the array.

Some binary skeletons are also defined; those with the same data type for both arguments (`rZip` and `aZip`) and those which have two different types for example `ramp`. Other examples are matrix multiply (`mm`) and vector-matrix multiply (`vm`).

Finally, a skeleton combination language is defined with rules for forming skeletal programs (WFSP rules) from primitive and combining skeletons. The combining skeletons include compose ($\circ$) (also generalised to binary skeletons ($\circ\circ$)), iteration over skeletons (`iterFor` and `iterUntil`) and split ($\triangle$) which applies two skeletons to a single data structure.

Skeleton combinations can occur at several levels, the result of one can be used as data for the next:

$$(\texttt{rFold}\,(+)\ \circ\ \texttt{rMap}\,(*\,2))\ [1,2,3,4]^r \tag{2.22}$$

first order:

$$rMap\,(aMap\,f)\,[\ll 1,2,3 \gg, \ll 3,1,2 \gg]^r \tag{2.23}$$

or higher order:

$$rMap\,rMap\,[f,g,h]^r\ ==\ [rMap\,f, rMap\,g, rMap\,h]^r \tag{2.24}$$

The last level requires runtime generation of topology and so was not considered. Another restriction that is imposed is that when one PADT is distributed over another, for instance the `rDistl` function:

$$rDistl\,k\,[a_1,\ldots,a_n]^r \;=\; [(k,a_1),\ldots,(k,a_n)]^r \tag{2.25}$$

it is assumed that only sequential data structures are distributed, which leads to some skeletons (such as `mm`) having to be defined from scratch rather than constructed from other skeletons.

Several example programs are presented which include sorting, convex hull, solving PDEs and perspective projection.

A model for reasoning about parallel implementations of skeletons is presented, these are based around Hudak's [168] parallel functional annotations to describe parallel behaviour. Two basic modes of parallelism are considered; data distributed and data pipelined along with simple combinations of skeletons.

For instance, the `rMap` skeleton in data distributed mode can easily be implemented in parallel by processing element n on processor n:

$$\begin{aligned} rMap'\,f\,((x,p):xs) &= ((a,p):b)\;on\,p\;sched\,Da\mid Db \\ where\,a &= f\,x \\ b &= rMap'\,f\,xs \end{aligned} \tag{2.26}$$

(Single dashes are used to denote distributed skeletons ($rMap'$) and double dashes to denote pipelined skeletons ($rMap''$)). Note that this is defined in normal lists rather than `rlists` which requires a conversion layer between the `rlist` definitions and the Gofer syntax used here. This also requires a (hypothetical) list constructor (`:`) for parallel spawning with zero cost.

Several other `rlist`-based data distributed skeletons are considered, for instance compose ($\circ$):

$$(f\;\circ'\;g)\,x \;=\; m@(f\;(n@(g\,x)))\;sched\,Dm\mid Dn \tag{2.27}$$

retains parallelism between combinations of `rMap'`. `rFilter` is complicated by the need to construct the output list sequentially so the predicates are evaluated in parallel followed by barrier synchronisation which reduces performance. `rReverse` is a communications only skeleton but can be handled using selection functions. It was not possible to express `rFolda` using the parafunctional notation but transformation was claimed as a means of getting the correct ordering. `rFoldr` and `rFoldl` are sequential and lead to barrier synchronisation.

Data pipelined behaviour was also considered for several skeletons. `rMap` was easily defined but needed revision to work with compose correctly. Compose was also easy to define:

$$(f \circ'' g)\, x \quad = \quad m@(f\, n@(g\, x\ on\ (self\ +\ 1)))\ on\ self\ sched\ Dm \mid Dn \tag{2.28}$$

This definition, however, has no parallelism when used with `rMap` due to the strictness with respect to the head of the list so a mechanism is required to request the next element of the list while the head element is being processed (`oneAhead`). `rFilter` is simpler than data parallel but uses the same lookahead strategy. `rReverse` is sequential and stalls the pipeline. A strategy for divide and conquer is also discussed which includes skeletons for partitioning data (`rEqualSplit` and `rPartition`).

Array-based skeletons are slightly complicated by the need to impose ordering on elements for pipelined behaviour. In general, elements are identified by the element's index and the pipelined annotation needs the look ahead as for data parallel `rlists`. The `ramp` skeleton is slightly more complex since it can be based on mixed data types and if one is `rlist` and the other array then there are several combinations of sequential, dataparallel and pipelined behaviour not all of which are useful.

Several other methods of combining skeletons are considered, for instance compose simply stalls if the two functions are not of the same behaviour but nested parallelism can cope with different behaviours of the inner and outer skeletons. The iteration skeletons `iterFor` and `iterUntil` simply behave as compositions of each iterative step. Transformation is briefly discussed as a means of minimising the effects of stalls and barriers in combinations of skeletons. Transformations are divided into; moving skeletons around, reducing synchronisation and removing reverse computations.

Implementation strategies for some of the skeletons are given and these are all targetted on distributed memory MIMD in a 2D toroidal arrangement. The array-based skeletons match this type of system and are developed in detail.

Array implementations are discussed in terms of block distributions (where contiguous blocks of data are split up among processors) and cyclic distributions (where the data is allocated to processors in a round robin manner). Some issues associated with distributing arrays across processing elements are addressed, for instance there are different ways of mapping higher dimension arrays onto 2D processor topologies. The `aLNO` skeleton (block distributed) is used as an example and two basic processing methods are considered; the *sequential loop* method where each processor loops over its local data and the *concurrent processes* method where a process is forked for each data element. The first allows coalescing of messages and has lower memory overheads and the second hides communication latencies better but forks far too many processes for practical implementation. A combined method is also possible and cyclic distributed is considered but is much poorer in performance.

Some other skeletons are also considered. `aMap` is trivial to implement but depends on static allocation of data and is not efficient if the data is not not regular. It is suggested that pipelined may perform better than distributed if the data is irregular. `aFolda` can be implemented block distributed on a virtual process tree and again, cyclic distributed performs worse both in communications and computationally. `aScana` block distributed is similar to `aFolda` but with local folds. Cyclic distributed is again much worse than block distributed.

Pipelined array implementations are also discussed in terms of a general pipeline stage instantiated for each skeleton (*generalStage*). This involves transformation into nested arrays to allow aggregation of communications. A single buffered algorithm is given but suffers from communications latencies so a double buffered algorithm is proposed which is generated using virtual stages. Instantiations of *generalStage* include; `aMap` which is trivial, `aFolda` which is easy to implement but exhibits no parallelism, `aScana` which is effectively a combination of `aMap` and `aFolda`. `aLNO` is also considered but care has to be taken to ensure that one output element is produced for each input element in the steady state, otherwise no parallelism results.

`rlist` implementations are also given but mostly concern mapping of `rlist` data onto the toroidal arrangement. Binary tree representations are not as useful for `rlists` as normal lists so embedding of a ring topology onto the 2D torus is used (Hamiltonian circuit). Data distributed rlist primitives are mostly as expected. `rMap` is again trivial. For `rFold` there are several possibilities based on reduction along columns or rows but needs slightly different behaviour if the number of columns is odd or even. `rFoldl` and `rFoldr` are sequential and simply follow the Hamiltonian circuit. `rFilter` is implemented in two phases (map the predicate then restructure the list). Data pipelined `rlist` skeletons are virtually the same as pipelined array equivalents and are not mentioned further. An implementation of the `ramp` skeleton also uses the Hamiltonian circuit.

Note that the performance of these implementations is given in terms of a performance model parameterised according to the number of processors, number of data elements and the various computation and communications unit times. These are intended to be used in an optimising scheme for combining skeletons where the overall performance is predicted using these models.

Given this basis, a compiler is proposed based on behavioural inference for primitive skeletons (for deciding the data distributed or data pipelined implementation for each primitive) followed by partitioning and scheduling according to the cost models. At present this is only semi-automatic, the programmer has to annotate the data structures as distributed or pipelined (both input data and structures generated during computation).

Behavioural inference is similar to type inference (based on behavioural variables rather than type variables) and are deduced using a subset of Milner's type inference algorithm $\mathcal{W}$ [169]. Thus, a program plus suitable annotations leads to a behaviour for each skeleton.

An algorithm is given for optimising linear combinations of data distributed skeletons. This should be simple but is complicated by different performance under different distributions (for instance block versus cyclic distributions).

The algorithm works by splitting the list of skeletons into sublists with the same block/cyclic distribution (the linear combination of skeletons is represented by a list). This is then optimised by balancing redistribution costs against suboptimal distribution for skeletons. Skeletons are first marked as "improvable" if the difference between the two distributions is less than the performance penalty of implementing a redistribution phase before and/or after the skeleton. Different analyses are required for isolated improvable skeletons and for sequences of improvable skeletons (where a dynamic programming approach is used to optimise the sequence). Other cases arise depending on whether the input or output data distributions are fixed or not although the algorithm is generic in this case. The complexity of this algorithm is $\Theta(m^3n)$ for m distributions and n skeletons but there is a less complex $\Theta(mn)$ prephase which reduces the complexity of the final algorithm.

The iterative skeletons also cause problems. `iterateFor` is handled by unravelling the loop three times and considering the initial, middle and final sections using the previous algorithm. Again, several cases arise depending on whether the initial, loop or final skeletons can be improved. If there is a sequence of improvable skeletons in the loop then a suboptimal approximation is required. In general an optimal solution may require different distributions for each iteration. `iterateUntil` is also too complex for an optimal solution but uses the same approximation as `iterateFor`. Some slight modifications are also required for binary skeletons.

Combinations of pipelined skeletons are handled by splitting at skeletons which cause a stall and implementing Bokhari's algorithm for pipeline balancing [170] on each segment. The iterative skeletons also cause problems for pipelined combinations and two cases are considered; one stalls after each iteration, the other merges iteration streams. The first case is easy to handle but the second needs modifications to the sequential functions to allow labelling of elements with the iteration number and is not applicable to all skeletons.

Nested skeletons are briefly considered and there are numerous cases generated by implementing the inner and outer skeletons as sequential, pipelined or distributed. Cases where there is sequential behaviour can be inferred from more general cases and $Dist(Pipe)$ is essentially the same as $Pipe(Dist)$ so only one is considered.

$Dist(Dist)$ can be handled by flattening nested `rlists` into a single `rlist`. This results in a loss of synchronisation but transformations are proposed to flatten prior to computation followed by reconstructing the nested list on output. The same transformations can be used to flatten $Pipe(Pipe)$ into a single pipeline. The flattening cannot be performed on $Pipe(Dist)$ so the synchronisation between the outer stages has to be minimised by fusing them together.

Although this thesis is the first major attempt to address the issue of combining skeletons in such a way that parallel performance is retained, there are a number of flaws in the proposals put forward. The parallel model is mostly discussed in the context of `rlists` which are easier to analyse but the implementation is mostly based on arrays which match the target architecture better, this rather avoids the issue of implementing a general algorithm in a transportable manner. The analysis is also restricted to one topology and it is not clear how different architectures would be handled by the methods suggested.

It is also not clear how extensible to other parallel data types (such as trees or graphs) and other data distributions the techniques are. There would certainly be a large amount of effort in incorporating a new PADT into the system and the effect on generality and transportability is difficult to predict. In common with most related work in this field, the high complexity of the algorithms given mean that any expansion would lead to the widespread use of suboptimal and heuristic solutions which could lead to poor parallel performance or even no parallelism at all. Finally, there is no implementation work in this thesis, the true utility of the methods given can really only be proven in a successful parallel compiler.

Rabhi [171] has also conducted a thread of investigation into algorithmic skeletons, which is termed the "paradigm-oriented" approach. This seems to be a slightly higher-order form of skeleton since the skeletons given here are slight generalisations of skeletons seen elsewhere. A *paradigm* is simply the broad computational structure of a class of algorithms and can be expressed as a specification which is termed a "template" in a functional language (Haskell is used here) but can be instantiated with other higher order functions to create a skeleton capable of further analysis. A sequential, executable specification is used to validate the algorithmic details and program transformation is used to generate an annotated parallel functional program. Four skeletons are described.

Recursively partitioned algorithms are a generalisation of the more familiar divide and conquer skeleton. Here the definition (`rp$`) is restricted to the statement that any problem which can be divided into subproblems can be specified by this skeleton, given functions for division, combination and solution of subproblems and an indivisibility predicate.

The *process network* skeleton generalises the streams/dataflow mode of computation where process nodes continuously pass data between each other and are connected in a graph structure. The specification given here (`pipe$`) is a linear pipeline of processes specified as a list of functions.

Distributed independent algorithms are similar to recursively partitioned in that a problem is divided into subproblems which are concurrently executed. In this case, however, there is a central controlling process and the number of subdivisions can be defined by the programmer. This is embodied in the `di$` skeleton which is a simple `map` implemented as a processor farm.

Iterative transformation algorithms are a generalisation of Cole's iterative combination skeleton [11], they are sometimes called *geometric parallelism*. These algorithms operate on a group of objects and process these objects in discrete steps. At each step, local operations can be performed on each object, global values can be computed from all the objects or objects can combine to form aggregate objects. A termination condition is also required. This skeleton is specified by the `it$` skeleton and two applications are developed; solving Laplace's equation and a topological sort. An instantiation of the `it$` skeleton for the Laplace problem is given and transformed into an executable specification using a transformation scheme $\mathcal{S}$ leading to a working Haskell program. This is suggested as being suitable for an implementation scheme using *distributed graph reduction* [172].

Schwarz and Rabhi [39] give some more details on the implementation of a static version of the iterative transformation skeleton (SIT). The proposal is to use a specialised form of graph reduction in parallel. A set of virtual processors are assumed to be mapped one to one with the set of physical processors and each virtual processor behaves as an abstract machine. The abstract machine consists of a lazy reduction machine acting on a local copy of a graph. Threads are handled by a thread pool and communications are buffered and include two lists for each thread; *dwait* is a list of processors waiting for the reduced value from the thread and *dreq* is a list of processors that will request the value, if needed. The thread terminates if both queues become empty. Only normal forms are transmitted and it is claimed that this results in only local garbage collection being required. No implementations are discussed but it is proposed to target a PVM [173] port of GRIP [174] using Haskell [103] as the reduction engine compiled into C.

Rabhi and Schwarz [39] develop the same system still further. A programming environment called "POPE" (Paradigm-Oriented Parallel Programming Environment) is proposed for the generation of implementations of SIT skeletons. The same $\mathcal{S}$ transformation is proposed for sequential implementation but the transformation scheme $\mathcal{P}$ is proposed for the generation of parallel implementations. A dialect of parallel Haskell is used as the target language, this contains low-level annotations for forking threads (`parlet`), requesting and updating remote values; (`request`, `updateW` and `updateR`) and for broadcasting (`broadcast`) and blocking (`wait`). There are actually two of these schemes, the first one $\mathcal{P}_1$ uses a *request-update* scheme. Here, references to external variables are replaced by calls to `updateR` and `request` for both local references and also the host's global references. The second scheme $\mathcal{P}_2$ uses an *update-only* method. This is slightly less general than $\mathcal{P}_1$ in that additional constraints have to be satisfied and the programmer has to supply some extra functions, but is likely to be more efficient than request-update. The same two examples of Laplace's problem and topological sort are treated with the $\mathcal{P}_2$ transform giving annotated parallel Haskell programs. One further transformation scheme $\mathcal{M}$ is specified which maps the logical references to physical references on the target system. Again, no actual performance figures are given but an overall development system is outlined.

Rabhi [175] hypothesises some further applications and implementational details for the four skeletons (or paradigms) given earlier. An implementation of the SIT skeleton using PVM and a graph reduction mechanism has also been presented [176].

Rangaswami [41, 105] has developed a comprehensive set of skeletons with associated cost models in a system called HOPP (Higher-Order Parallel Programming). The main motivation seems to be to address the problem of architectural independence since each program is intended to be implemented on a number of different architectures.

The base language is functional in nature, founded on list processing and the notation is based on the Bird-Meertens formalism with some additional functions. These are called *recognised functions* in HOPP terminology since parallelism only stems from the direct implementation of one of them. The use of a functional language means that program transformation is possible, although in the approach adopted here transformations are only applied by the user at the source level. The programmer is also expected to input some data dependent values such as estimates of the size of the lists. This is due to the lack of a profiler for the analysis phase.

Being list-based limits the applicability of the system but extensions to other data structures such as arrays [177] should be possible. The use of lists also implies some implementational overheads such as packing and unpacking lists for transmission down links and computational overheads such as arbitrary list access which is $O(length\,of\,list)$. Another limitation is that since the analysis is essentially static in nature, only regular problems can be handled. This precludes algorithms such as quicksort which are dependent upon the data for the pattern of computation.

A HOPP program is viewed as a sequence of phases. Each phase is evaluated to completion before the next phase can commence but pipelining of phases could be a possible extension. If a phase contains no recognised functions then it is implemented sequentially. One of the most advanced features of this system is its ability to handle nested recognised functions down to three levels (any recognised functions below that are implemented sequentially). All combinations of sequential/parallel implementation of the recognised functions at each level are considered (giving eight possibilities in all, including a sequential implementation) and a search tree is constructed to find the most efficient solution. This grows exponentially with the number of recognised functions so pruning would be required to limit the search space for large programs, at present this is not implemented so only small programs can be handled.

The cost model is parameterised according to the target architecture. This implies that the models are at a relatively abstract level, for instance the costs of unit operations such as multiply and divide are not taken into account. This possibly limits the accuracy of the models. The cost for a complete program comprises the sum of the costs for each of the phases plus the cost for any data rearrangement communications between each phase:

$$Cost = \sum_{i=1}^{n} C_{p_i} + \sum_{i=0}^{n-1} C_{i,i+1} \qquad (2.29)$$

where C_{p_i} is the cost of phase i and $C_{i,i+1}$ is the cost of rearranging data from phase i to phase $i+1$.

At present there is no back-end compiler for this system so implementations are essentially hand-coded. Some support is provided by libraries of functions, however.

The list of recognised functions is extensive and these are divided up into a basic set and an extended set. Some of them can be expressed in terms of others but are present for efficiency reasons, also some of them have different variants determined by the nature of the data they are applied to. These are specified in an ML-style syntax.

The complete list is, a) the basic set;

- `map`
- `s_fold` (static)
- `g_fold` (growing)
- `scan`
- `filter`

- `inits`
- `tails`
- `r_cross_product` (row major)
- `c_cross_product` (column major)
- `composition`

and b) the extended set;

- `map2`
- `len` (length of list)
- `select` (element of list)
- `zip`
- `apply_select` (map with mask)
- `copy` (distribute element over list)
- `iterate_up` (iterate f over list for count)
- `iterate_down` (same downwards)
- `iterate_cond` (iterate until cond)
- `reverse`
- `split` (list at point)
- $\mathcal{R}^k$ (pipe list through functions)
- `get_neigh` (get neighbours within list)

Some of the functions such as `fold` are split up into *static* `s_fold` where the data size remains the same and *growing* `g_fold` where the size of the data changes as the fold is applied. This is to allow communications costs to take changes in data sizes into account.

Three example programs are presented. **Matrix multiply** is a combination of `map`, `s_fold`, `r_cross_product` and `map2`:

```
fun mat_mult times plus A B =
  map (map (s_fold plus 0)) o
  r_cross_product (map2 times) A (transpose B)
```

P	Parse tree	
M	Parallel machine characteristics	
		Topology
		Number of processors
	K_0, K_1	Link characteristics
D,I_s,F_t,S	Input list characteristics	
	D	Level of nesting
	I_s	Lists sizes at each level
	F_t	Size of elements at each level
	S	Relationship between list sizes and processor counts
C_f,F_s	Specifications for sequential functions	
	C_f	Sequential function costs
	F_s	Function output sizes

Table 2.5: Input to a HOPP Program

Mode	Description
Nearest neighbour	one packet to neighbour
Broadcast	single packet to all processors
Scatter	different packets to all processors
Gather	packets from all processors to one
Total exchange	all packets between all processors

Table 2.6: HOPP Data Rearrangement Modes

Merge sort is composed of `g_fold`, `map` and `split`:

```
fun sort n xs =
  g_fold (merge []) o map msort o split (n div 2) xs
```

The third example involves the solution of partial differential equations using Jacobi iteration and is much more complex but comprises; `map`, `map2`, `get_neigh` and `iterate_cond` in various combinations nested to the full three levels.

A HOPP program is specified by a number of program and hardware specific parameters. Table 2.5 shows the major items, some can be deduced directly from the program such as the parse tree, list lengths and element sizes, others are related to the target architecture such as the topology, communications characteristics and the number of processors. Two algorithms are outlined, one ($ANALYSE$) defines the search space of parallelisations and the other ($COMPUTE\_COSTS$) finds the most efficient way to split processors for a single phase.

The major part of this thesis is the derivation of the costs for each of the skeletons on each of the four topologies; hypercube, 2D torus, s-ary tree, and linear array. Firstly, the costs for rearrangements of data in between phases are derived, five basic modes of communication are considered, Table 2.6 describes these modes.

The costs for each recognised function are calculated for sequential operation and for parallel implementation on each topology. For instance `map` has the sequential cost:

$$C^s_{map} = nC_f$$

and has the same cost on all four topologies:

$$C_{map} = \frac{n}{p}C_f$$

Most other functions have different costs for each topology. In general the costs for linear and tree implementations were too inefficient and were not studied further. The 2D torus was outperformed by the hypercube, this was attributed to the fact that it is difficult to split a torus up without causing variations in the costs on each of the subdivisions, whereas a hypercube can be split up into uniform sections. An implementation scheme for the torus arrangement was sketched, this involved embedding a logical hypercube onto the torus.

Implementations are supported by a parallel library in C, using CSTools as the communications harness. The library consists of; C prototypes for the recognised functions and the list data structure, implementations for the basic communications modes and auxiliary functions for activities such as packing/unpacking of lists. In addition, some optimisations have been incorporated into the support routines; memory allocations are aggregated, where possible, some function calls are expanded inline, the scatter mode is implemented using a more efficient *shuffle* scheme and recognised functions involving communication can be more efficiently implemented by running to completion prior to communication.

Performance results are presented for the three applications and the actual performance compared with that predicted by the cost model. For matrix multiply the best implementation is the outer `map` and `r_cross_product` in parallel. Actual performance is less than predicted but the trends are accurate. For merge sort the best implementation is all functions in parallel for all phases, this gave a more accurate prediction than matrix multiplication but still less than the theoretical prediction. The Jacobi iteration required adjustment due to limitations of the analyser. The best implementation is implementing (mostly) the first level of nesting below the top level `iterate_cond` in parallel. This still resulted in an overestimate of the actual performance and the correction seemed minimal.

Feldcamp et al. [178, 179] propose skeletons as an environment for parallel programming. Two skeletons are developed [178], *processor farm* and *divide and conquer* along with performance models for speedup, throughput and startup costs. Implementation on transputer networks is described and an application is developed using the processor farm skeleton, Cepstral filtering for motion analysis. This is "tuned" for performance optimisation which involves optimising the task distribution along a linear topology of transputers and controlling the granularity of the tasks. The program development environment Parsec is also described [179]. This consists of a set of tools for the construction of a skeletons-based program and includes visualisation tools for message passing systems. The system is built on top of a virtual machine for message passing systems.

Deldarie et al. [10] derive performance models for two skeletons with applications in image processing. The first is a local window operation (LW) which maps from images to images such that the pixels in the output image are derived from a (square) window of pixels in the input image. The second skeleton is a *split and merge* model (SAM) in which images are split into slices, a function (*solve*) is applied to each slice and then the results combined hierarchically using a *merge* function. Performance is predicted for execution on the WPRAM model [180], which is a variant of the PRAM model incorporating ideas from BSP [181].

Complexity results are derived for both LW and SAM skeletons, these are parameterised by the size of the image (n), the number of processors (P) and the costs of the various instantiation functions. The SAM skeleton also includes estimates of the changes in data sizes induced by the *merge* function. The theoretical results were compared against simulations for edge detection applications of the LW skeleton and histogram, median filtering and 1D FFT instantiations of the SAM skeleton. The simulator was setup with parameters based on the T9000 transputer and the comparison was claimed to be good.

An interesting aspect of this work is the use of the *isoefficiency* function [182] in the asymptotic analysis. This is defined as the rate at which the problem size must grow with the number of processors in order to maintain constant efficiency and is perhaps a more easily visualised measure of the scalability of an algorithm than simple speedup behaviour. The isoefficiency was calculated from the WPRAM complexity expressions and experimentally measured (using an augmented tree search) for the Sobel edge detection algorithm. This gave values scattered around and with a slight trend relative to the theoretical isoefficiency, a fact which is blamed on the costs of forking worker processes which become relatively less important as the number of processes increases.

The use of asymptotic analysis as opposed to direct performance prediction is interesting and has not been applied widely to algorithmic skeletons. Another unusual aspect is that the skeletons are written in C as opposed to a functional language, this is one of the very few instances of imperative skeletons and the results look promising for the flat compositions of skeletons used here.

Constructor	Description
`pipe`	process pipelines
`loop`	iteration and recursion computations
`farm`	processor farming
`geometric`	vector processing and data parallel
`tree`	tree structured computations

Table 2.7: P^3L Constructors

Compiler-based Approaches

These have been used in conjunction with ideas from skeletons technology. A skeletons-based compiler is probably the ultimate goal in skeletons research but there are still no complete and reliable implementations of such parallel languages either incorporating skeletons into a high-level abstraction layer on top of a language or using skeletons themselves as compiler intermediates during the compilation process. There are two basic approaches; develop an existing language, for instance SML, incorporating skeletons into the evaluation scheme for that language or devise a special purpose language which has the facilities to support skeletal programming and compilation.

Danelutto and others [183, 184, 185, 186, 187] have developed an explicit parallel programming language P^3L which is based on a restricted set of parallel constructs and a set of compilation and optimisation tools aimed at providing portability without sacrificing performance or efficiency. The target hardware is the general class of message passing systems, attributes such as topology, communications strategy and scheduling strategy are intended to be the responsibility of the compiler.

The language P^3L is based on a sequential component and a set of constructors for building parallel programs. The sequential component could in theory be any sequential language (one version uses a subset of C++ [184]). The constructors are present as language primitives, these are shown in Table 2.7.

In addition, processor farms are divided up into three cases. *Pure* farms apply a single function to each value of an input stream. *MISD* farms apply a different function for each processor to each value of the input stream and return all the resulting values. *Dedicated* farms are like pure farms but with an additional guard associated with each processor. The `tree` constructor can be defined as `combining` where data flows from the leaves to the root or as `distributing` where data flows from the root to the leaves. The `geometric` constructor includes a `topology` definition (for instance `array`, `mesh`) although it is claimed that this can be deduced by the compiler if none is given.

The compiler consists of a front-end performing syntax analysis of the source code, a middle section which optimises the intermediate representation for a specific architecture and a back-end which outputs code in the P^3M abstract machine [188]. This abstract machine provides the compiler with an abstraction over the target hardware, mostly defining memory limits, topology, basic operations and also provides costs for basic operations such as interprocessor communication. Note that the *type* of parallelism is specified by the programmer using the constructors, the compiler's job is then to provide an efficient implementation of that parallelism on the chosen hardware.

Each constructor is associated with an *implementation template* which comprises a *mapping template* which describes the mapping of processes and channels and a *process template* which is the actual code required to implement the construct. These are successively used by the compiler which operates in two passes. Also associated with each construct is an *analytical performance model* which allows performance predictions for the P^3M abstract machine, the models for the parallelism constructs are based on costs such as communications costs and routing costs (the network may contain nodes which only distribute or collect data). The costs for the user supplied `sequential` code are calculated by profiling an implementation of the P^3L program but a static analysis tool is also under development.

The core of the compiler consists of three elements; the *process graph handler* which performs local optimisation on the process graph when it is first generated or when new ones are created by optimisation, the *mapper* which selects a mapping strategy from a library of costed strategies and the *load balancer* which attempts to statically balance processing element loads. The overall strategy is that the mapper generates an optimal arrangement of processors assuming sufficient resources to achieve the optimum. The load balancer then reduces this down by allocating multiple processes to processors and adjusting processor counts for the parallel constructs until the resources of the target architecture are matched.

The first prototype compiler was written in Prolog and P^3L programs were handled as Prolog data structures. This was presumably to facilitate the search procedures used during optimisation and mapping. This compiler also made extensive use of libraries, the *mapping* library, the *process template* library and the *optimisation* library. This was presumably to allow new constructs and architectures to be added without altering the compiler. Early work only produced code for the Proteus parallel simulator [189] and for the Meiko CS1 based on transputers.

More recently, a template-based compiler for P^3L targetting the Fujitsu AP1000 has been reported [190]. This describes a compiler called *ANACLETO* in which skeletons are organised into a *process graph* and mapped efficiently onto the parallel machine using *implementation* *templates.* Each template is provided with an analytic performance model allowing performance predictions for the target architecture. Parallel harnesses are developed in a subset of MPI and the process farm template is used as an example although it is claimed that all the P^3L skeletons can be handled.

Danelutto *et al.* [191] describe a set of data parallel skeletons and the methods needed to handle them. The data parallel skeletons are `map`, `reduce` (fold) and `comp` (composition), all applied to array structures. The *game of life* program is used as an example, the implementation and compilation are described and results are presented for the Fujitsu AP1000. Two implementations are described, one using an implementation template, the other using compilation of data-parallel subtrees (this is only partially automatic). Parallel performance is indicated as being dependent upon optimisation techniques, much better results are achieved for optimised as opposed to non-optimised versions. Performance predictions are claimed to be accurate, however.

Bacci *et al.* [192] summarise the experience of developing the P^3L compiler. The compiler itself is described and the parallel performance of five applications on four parallel machines (Cray T3D, Meiko CS-2, CM5 and a T800 mesh) are presented. This work shows how a hierarchical structuring of skeletons can be used to exploit different types of parallelism. In addition, the performance prediction errors are all below 11%.

Busvine [56, 57] developed an SML-based compiler called PUFF (parallelisation using farmed functions) in which arbitrary linear recursion is implemented as a processor farm. A compilation scheme (**FarmFns**) is presented which extracts the farm control functions from lambda-lifted functional language source code based on a simple grammar. This scheme is used by the PUFF compiler to generate Occam2 code from a pure functional subset of SML using profiling and performance modelling to guide a heuristic determination of which functions can usefully be implemented in parallel. An example is presented which is a ray-tracing program and produces useful speedup for several scenes, although the parallel performance seems to be data dependent.

Bratvold [40, 59, 60, 193] developed the first working compiler based on algorithmic skeletons. This is called SkelML and is based on a pure functional subset of Standard ML, including datatypes, polymorphism and pattern matching but excluding local function definitions, abstract datatypes and records. The idea behind this compiler is that as much of the burden of deciding on parallel performance as possible should be placed on the compiler. The programmer thus provides a working SML program with representative sets of data and the compiler deduces the best implementation from there, including profiling the input data sets, transformations for optimisation and back-end output of Occam2, the target platform

Skeleton	Description
`map`	apply function to list elements
`filter`	filter list elements according to predicate
`fold`	fold elements with associative operator
`filtermap`	`(filter p) o (map f)` optimised
`mapfilter`	`(map f) o (filter p)` optimised
`foldmap`	`(fold f) o (map g)` optimised

Table 2.8: SkelML Skeletons

is a transputer system.

The compiler is front ended with an ML interpreter coded in C++ allowing debugging of SkelML programs in the usual way. This is based directly on the structural operational semantics of the Standard ML definition [15] and indeed the profiler also operates by counting semantic rule firing. The profiler is built into the interpreter along with the *process network extractor* which builds a graph of skeletal parallelism and the *mapper* which maps the process network onto a physical network of processors and links.

The back-end is based on Busvine's SML to sequential occam translator [58] which is a stack-based SECD machine style evaluation. Additional support is provided for memory management, garbage collection and list representation (which have to be emulated in Occam2's arrays). Support is also provided for generic communications so that types such as tuples and constructors can be packed and unpacked for transmission down Occam2's links. Skeletal code is also output by the back-end in the form of generic harnesses for each skeletal construct along with wiring tables and processor placements.

The skeletons provided by SkelML are shown in Table 2.8. `map` is implemented as farming on a linear pipeline of processors. `filter` is implemented in a similar manner but with some additional indices to allow reconstruction of the output list. `fold` is implemented as a logical process tree mapped onto the linear pipeline topology. The composite skeletons are implemented in the same manner as their components but with communications optimised by performing both operations on the same worker.

Performance modelling is based on information gathered from counter-based profiling. The major points of interest within the subset of SML are function calls and branches in the form of pattern matching. One complication is that functions may be called from more than one point in the program or even called via different routes from the same point. It is for this reason that profiling counts are tagged with the source of the function call to allow the correct counts to be used in context. The entire program is profiled rather than points which affect parallelism, however, this is stated as being useful should deduction of parallel constructs be implemented in future. The main problem is relating the semantic rule firing to the Occam2 executions costs. This is achieved by attaching a cost for each rule which is added to a global clock when a rule fires although there is an additional complication for some rules. An example of this is that function application could be top level or recursive, which requires additional information for disambiguation given as extra formulae attached to the rule. This allows the program graph to be annotated with the profiling costs in a simple way and, it is claimed, allows separation of the profiling mechanism and the target architecture (this is captured by the table of costs and formulae). With suitable costs this method can be quite accurate, comparison of estimated costs from profiling with measured sequential Occam2 are typically within about 5% to 10% of each other. Communications costs are handled by a similar method of relating average data object sizes to the target hardware using tables of cost information, these reflect overheads such as packing and unpacking of tuples and lists.

Skeletal performance is modelled using coarse parameters such as execution times for sequential components, data sizes (plus list lengths) and some additional considerations such as startup and shutdown costs and a simple modelling of latency. The analysis is simplified by the fact that the only topology considered is a linear pipeline of processors. For instance, the `map` skeleton is modelled by assuming a linear distribution of work across the pipeline and taking the maximum processing time for the first and last processors. This gives a very simple relationship between the parameters and the parallel performance. In addition, although average execution times are used, the processor farming approach gives a limited robustness in the presence of irregular workload. Similar analyses are given for the other skeletons with minor complications, for instance the percentage of pass/fail results for the `filter` predicate has to be taken into account. Again assuming suitable values for the various architecture dependent constants predictions of performance are about 10% of the measured values.

The process network uses a representation similar to dataflow graphs [194], nodes can only process when data is present on the input arcs and only produce output upon evaluation of input data. Arcs are also typed which allows deduction of communications costs. The whole network is attached to a terminal node which acts as the external interface to the program and an additional SML expression is used (called a *CallNodeExp*) to allow handling of skeletons within sequential constructs, this allows the production of Occam2 code to evaluate the arguments in parallel and return the results of the skeleton. Nodes are classified as skeleton nodes (which are produced for each skeleton present in the program) and sequential nodes which are kept as large as possible to minimise network complexity. The process network extractor is described in terms of a scheme $\mathcal{N}$. This takes an SML expression and a list of identifiers and returns a 4-tuple consisting of an expression or a node, a list of identifiers, a set of nodes and a set of arcs. This is applied recursively to subexpressions. The resulting network contains entities which are all covered by the performance models except the *CallNodeExp*

#	Pre		Post
T1	`map f (map g l)`	==	`map (fn x => f (g x)) l`
T2	`map f (filter p l)`	==	`mapfilter f p l`
T3	`filter p (map f l)`	==	`filtermap f p l`
T4	`fold f a (map g l)`	==	`foldmap f a g l`
T5	`filter p2 (filter p1 l)`	==	`filter (fn x => (p1 x) andalso (p2 x)) l`
T6	`filter p1 (filter p2 l)`	==	`filter p2 (filter p1 l)`

Table 2.9: SkelML Transformations

expressions which have to be handled separately. Finally, some additional structure can be imposed on the network by considering pipelines of nodes and pipelines of skeletons. These are intended to help reduce the overall complexity of the network for large programs.

Due to the fact that mapping of processes to processors is, in the general case, NP complete [195] and that there are several approximations to the problem, SkelML implements a simplified version of a scheme $\mathcal{M}$ which itself has several limitations. Nodes are assigned to processors based on penalties such as; allocate communicating nodes to the same processor, keep arcs on the same links to allow greater mapping flexibility and having a penalty for each connected processor link so that when all links are allocated, only nodes with matching communications can be allocated. The scheme **AssignProc** is given which allocates sequential nodes according to the penalty costs and skeleton nodes according to the performance model predictions. A second scheme **Balance** is given which iteratively improves the load balance by reallocating resources from the fastest completing processes to the slowest, this scheme uses $\mathcal{M}$ to find the optimal allocations for subtasks. $\mathcal{M}$ itself is divided up into a separate algorithm for each type of node, for instance $\mathcal{M}_{PAR}$ deals with the PAR constructor and is simply a recursive call to the **Balance** scheme, the other components (SEQ, SKELPIPE and PIPE) are more complex and additional components are provided for SML expression evaluation $\mathcal{M}_e$ which also includes *CallNodeExp* expressions. The major problem with this algorithm is that it is exponential in the depth of recursive calls to $\mathcal{M}$ although it is suggested that nest depth may not be a good measure of program complexity. Another point is that the algorithm is not optimal, for instance parallelism within tuples is not exploited and nodes are allocated in a sequential manner.

SkelML incorporates a simple transformation scheme for optimising the overall performance of linear pipelines of skeletons. These are given in the form of equational rules and are shown in Table 2.9. The transformation process is handled automatically by the compiler and is simply a depth-first exhaustive search of the tree defined by the above transformations and is therefore limited to small programs and small numbers of transformations.

Several example programs are presented. The first is a ray-tracing problem and results in a network with one sequential node and one skeleton node (`map`), plus the terminal node.

This gives speedups of between 6.7 and 13.6 for 23 processors for different complexities of the input scenes. The performance prediction varies from about 16% to about 48% but this is attributed to the fact that the input data was scaled to cope with compiler limitations. The second example is a best match algorithm applied to lists of polygons and is inspired by work into the Homomorphism lemma (Lemma 2.1). This results in a network with one sequential node and two skeleton nodes (`fold` and `map`) in a skeleton pipeline. The transformation scheme, however, notes that this can be transformed (using T4 in Table 2.9) to combine the skeleton nodes into a single node (`foldmap`). This gives a speedups of 7.3 for 200 objects and 9.5 for 1000 objects using 24 processors. The prediction accuracy is about 11%. The last example involves calculating the areas of a list of polygons, filtering out only those that are symmetric. This gives a process network of four skeletons in a pipeline (two `map` and two `filter`). In this case the transformation system identifies 14 possible implementations (using combinations of T2, T3, T5 and T6). The best one turned out to be a two node pipeline with `filtermap` with a combined predicate and a `map` node giving a speedup of 4.0 with 9 processors.

The main limitation of SkelML is the complexity of the compilation schemes, particularly the mapping and transformation schemes for which heuristic solutions or solutions for restricted domains would be required for a more general compiler. The performance models are also quite limited in their sophistication, more detailed models would be required to give better predictions but the profiling method is very successful. Another point is that only a limited set of skeletons are supported, compared to other works, for instance [41, 42], although a complete set of skeletons for SkelML would increase the complexity of the compilation schemes to a point where it is was unusable. The subset of SML that is implemented is also quite restrictive and expanding this would have the same effect on the computational complexity of the compiler. Finally, there is no support for nested parallelism although this could be added without changing any of the basic methods. Although none of the components of SkelML are optimal or completely general it presents a fairly complete implementation of a parallelising compiler based on algorithmic skeletons.

Application-based Approaches

Developing real applications is a much less common approach to the development of skeletons technology but some work has been carried out in the field of computer vision. The idea is to place skeletons in the context of systems development to evaluate how the methodology could be used as a system development tool rather than a means of devising specific parallel algorithms or as a compilation mechanism. Unfortunately, results in this area are rather vague. There is no doubt that skeletons can contribute to systems development but how it compares with more conventional methodologies is difficult to assess.

Wallace et al. [52, 55] present examples of the use of SML prototypes and Occam2 implementations for visual scene labelling [52] and model-based matching [55]. The first example presents a (sequential) system for scene labelling in which the ML prototypes are used to guide the Occam2 development. The model-based matching is implemented using a pipeline of farm constructs which is balanced using profiling information and leads to acceptable parallel performance (a speedup of about 12 for 14 processors). A primitive attempt at transformation was also used to optimise performance.

Waugh [16, 53] also developed Occam2 programs coded from SML prototypes but considers the mechanisms of deriving the parallel implementation from the prototypes in more detail. Several versions of a perspective inversion model matching algorithm are presented along with data placement and optimisation strategies.

Michaelson and Scaife [61, 62, 196] have continued this theme of hand coding Occam2 implementations from SML prototypes. An attempt to build a complete system from existing SML/Occam2 components is presented [61] although there are severe limitations as to what can be achieved without breaking up the individual modules. A wider range of vision algorithms developed in SML with a view to parallel implementation is also presented [62] although no parallel performance is presented. One of these algorithms (edge tracking) is analysed in detail [196] and is in fact highly problematical in terms of parallel performance, a combination of program transformation and brute force was used to get decent parallel performance but the prototype effectively became dissociated from the implementation during this process. One conclusion from this could be that the skeletons methodology breaks down when there is too great a mismatch between the source language and the parallel hardware.

Serot [197] has developed image processing applications using a skeletal approach. Several vision applications were developed using functional languages.

Ginhac [198] presents a system called SKiPPER (A SKeleton-based Parallel Programming Environment for the Rapid prototyping of vision applications). This system takes executable specifications in ML and provides a set of functional skeletons for data farming, task farming and geometric decomposition which are parameterised with sequential C functions. Vision applications are then built by composing these skeletons, two examples given are connected component labelling and a road-tracking algorithm based on mark-detection.

2.3.3 Related Work

Algorithmic skeletons does not really have any direct competition in the field of parallel programming but the idea intersects with some of the current research and effectively competes with others. The closest thing to an algorithmic skeleton in imperative parallel programming is the idea of *templates* found in languages such as the various dialects of parallel Fortran. The idea of relating source language constructs to parallel constructs in the implementation is quite widespread but often occurs in a much more restricted capacity, for instance vector or array processing. The ability to transform source code and the accuracy of performance models are much more difficult to achieve in an imperative context, however, although attempts have been made [199].

Parallel abstraction

Such mechanisms have been hypothesised for a long time but have proved elusive in practice. This work has spawned a number of ideas, however, that are similar to some aspects of algorithmic skeletons.

Crowl and Leblanc [200] propose the use of control abstraction in parallel programming. This is not a new idea but the version presented here is algorithmic skeletons in all but name. Programming proceeds by defining new user defined control constructs by constructively building upon a small number of primitive predefined control constructs. These include *statement sequencing*, *operation invocation* (recursion), *first-class closures* (similar to a procedure), *conditional execution* which can be used to implement *while* and *repeat* constructs and *condition variables* for synchronisation.

New control constructs are defined by specifying a *syntax description*, a set of *precedence constraints* and at least one *implementation*, verification proofs are also possible. Using the user defined control construct mechanism allows the definition of common control constructs such as *cobegin* for parallel execution of closures or a *forall* construct similar to the parallel `for` loop in many parallel languages. The separation of definition from implementation allows more than one implementation of a given control construct. These are annotated when they are defined to allow selection at invocation, for example `cobegin $SEQUENTIAL` and `cobegin $PARALLEL`. This allows "tuning" of particular programs to meet new circumstances such as a new architecture.

It is pointed out that control abstraction can be used in conjunction with data abstraction and it is claimed that control abstraction encourages representation-independent programming provided the designers of the abstractions give a rich enough set of operations for the implementors to derive sufficient parallelism from the program.

Although this work uses something like the skeletons idea, the method is mostly to facilitate parallel programming by providing code reuse and a library of parallel implementations for a given construct to allow tuning for new situations such as different numbers of processors or different architectures. A language called Natasha is described which operates on control abstraction principles.

Foster and Taylor [201] present a transforming compiler for the PCN language (program composition notation). This language is founded on four concepts; *monotonicity* provides communication and synchronisation through single assignment variables, *concurrent composition* provides parallel execution and in conjunction with monotonicity allow segregation of mapping and partitioning from the other compilation activities, *choice* which is a form of alternation and *separation of sequential code* so that the sequential components can be manipulated as first class objects. The language is block structured (similar to C) but a block has the form $\{\texttt{operator}\ P_1, P_2, \ldots, P_n\}$ where `operator` can be $\|$ for concurrent execution, ? for choice or ; for sequential execution. In addition, programmer defined `operators` can be used which are implemented by calling user defined transformations during compilation. Mapping of procedures to processors is defined using the `@location` annotation.

The compiler operates as a series of source code transformations. Firstly, programmer defined abstractions are transformed out leaving primitive PCN code which is in turn transformed into Core PCN which is a high-level representation of a fine-grained concurrent programming model. This representation is then translated into an abstract machine which incorporates the facility to call sequential code from other compilers. There are a number of ways of implementing this abstract machine depending on the particular architecture. An interesting point about the transformation system is that it is implemented in PCN itself which means that the compiler can effectively compile itself in parallel. The transformations themselves are defined using *set* operators (programs can be represented as sets of procedures, variables etc.). Another point is that the mapping operation is completely separate from the abstraction mechanism allowing different physical mappings for the same control abstractions.

This work levels criticism at an early version of algorithmic skeletons (Cole's thesis [11]). The main argument is that PCN allows the construction of user defined abstractions (which are similar to skeletons) whereas skeletons only allow a small fixed set of constructs. The problem here is that it would be difficult to perform load balancing and performance modelling when there are arbitrary algorithmic constructs involved. This criticism is also less applicable to more recent skeletal work where compositions and nesting of skeletons have been investigated. Where the PCN approach does score, however, is the ability to incorporate existing sequential code from other compilers which, it is claimed, is influential in adapting large volumes of existing sequential application code for parallel execution.

Automatic parallelisation techniques

These represent a significant proportion of research into parallel systems. There are a wide variety of different methods but none operate on arbitrary code, for instance a great deal of research has gone into parallelising iteration loops in sequential languages.

One mature example of this approach is the Stanford SUIF compiler project. This is based on C and Fortran and is capable of a bewildering variety of parallelism detection and optimisation techniques. These include activities such as data placement optimisations [202] and data and code transformations [199]. In addition, these optimisations can span procedural boundaries [203], in effect this looks upon a program as a single linear algorithm but analyses based on actual inlining of procedural code are likely to be limited in the size of program they can handle. Although this is not in any way skeletons-based it does effectively compete in that it attempts to deduce specific types of parallelism from arbitrary sequential code, for instance replacing `for` loops with parallel `forall` equivalents where appropriate. This certainly has the great advantage of allowing the analysis of existing sequential code for parallel execution and compilers such as these are becoming adept at discovering parallelism in the presence of array operations and nested iteration loops. These are the basis for a wide range of existing software, for instance linear algebra and matrix computations, [203] presents results for a large range of benchmark programs.

Pinter and Pinter [204] give a method for extracting *idioms* from sequential loop structures. The program is parsed and loops are normalised and then unfolded three times to give initial, middle and terminal sections. This is expressed in the form of a graph structure (called a *computation graph*) which is used to optimise loops by eliminating communication and synchronisation steps. The idea here is to use subgraph isomorphism to detect known patterns of nodes in the computation graphs which relate to known efficient parallel structures. These include structures such as *reduction*, *scan*, *recurrence equations*, *transposition* and *reflection*. Composites of these such as convolutions and permutations can also be generated and filtering can be handled at a basic level. Several examples are given which, it is claimed, can deduce parallelism where other similar methodologies in loop optimisation have failed.

This technique is interesting in the context of skeletons work since it effectively deduces skeletons from arbitrary sequential, imperative code. It may even suggest another role for algorithmic skeletons in terms of automatic parallelisation and parallel program optimisation techniques. The only major problem here is the computational complexity of the subgraph isomorphism problem which will worsen if a larger list of idioms were used.

Fisher and Ghuloum [205] present another method of parallelising iteration loops, this time adopting a more semantics-based approach as opposed to the syntactic pattern matching of Pinter and Pinter. This method uses the observation that function composition is associative and that the body of an iteration loop can be modelled as a function application. After a process of logic minimisation for conditionals the loop modelling functions are extracted in two passes by assigning recurrence variables to flow dependencies in the data dependence graph followed by elimination of non-loop carried instances of the recurrence variables. This is followed by a technique called *templatisation*. This involves building a composition of functions such that symbolically constant terms are abstracted out, thus shifting computational load from a sequential operation (the evolution of loop variables) into a parallel one (the application of the composite functions which can be implemented as a parallel prefix operation). The costs of generating the composite functions has to be balanced by the parallel implementation benefits.

Unfortunately there are still some high complexity operations associated with this method, for instance a structural isomorphism detector is required to unify conditional nests which uses ordering heuristics to keep computational complexity down. The importance of this method is its generality, it is not dependent upon a set of predefined parallel constructs and although at present the method only applies to programs with quite simple semantics (arithmetic operators and conditionals) the method should be equally applicable to more complex loops such as packing loops (where the data changes size as the loop progresses) or pointer jumping loops.

Only a very small proportion of the work into automatic parallelisation has been presented here. This actually comprises a much more mature and more thoroughly investigated idea than the skeletons method and has already borne fruit in terms of real applications. The main advantages it has over skeletons is that existing code is more easily adapted using automated compilers and that most of the work into automatic parallelisation (particularly iterative loop parallelisation) has been done on imperative code in which the bulk of scientific and engineering applications are written. A few selective examples were presented simply to give the flavour of this work and to give a comparison with the algorithmic skeletons methodology.

2.3.4 Summary

A substantial number of papers have been reviewed in this section. The idea has progressed substantially since the term “algorithmic skeletons” was coined. None of the work is at the stage where the use of algorithmic skeletons can be shown to have a beneficial effect on real applications, either in terms of the development of the application or upon the parallel performance. This cannot be far away, however, if some of the systems described in this section were to be completed satisfactorily and live up to their initial results.

One problem is that there are a large number of different interpretations of how the concept of algorithmic skeletons can be employed usefully; either as an abstraction mechanism for parallel systems, as an intermediate for a parallel compiler or as a means of automatic parallelisation of sequential code. There are also a number of different mathematical and programming language choices that have to be made in the development of skeletons ideas. Formalisms such as BMF, category theory and FP have been used in the theoretical analyses along with more conventional approaches such as graph reduction and other functional language theory and several languages have been used as a host for skeletons-based compilers, for instance SML and Clean as well as dedicated languages such as P^3L.

A concise summary of the present state of skeletons methodology is that there is a great deal of ongoing work into this method, as well as many interesting and important results, but a great deal of work must be done before any commercially viable systems can be produced.

2.4 Summary and Conclusions

The review in this chapter was intended to complement the major work of this thesis which concerns the design and implementation of a parallel vision system.

Section 2.2 reviews other approaches to building systems specifically for running parallel vision applications. Some of the algorithmic problems were outlined and illustrated with several examples. Distributed memory MIMD systems feature prominently in this area justifying the use of a transputer-based system for the implementation work in this thesis.

Section 2.3 reviews the algorithmic skeletons approach and grouped this work according to a simple taxonomy. Vision systems are a rarity in this work so our application to computer vision systems is an original approach. It appears that most of this work is still experimental and no mature and robust support is available in terms of compiler support or automated programming tools. This is why the investigations in this thesis are somewhat hampered by being restricted to hand-coded parallel implementations in Occam2. The work into algorithmic skeletons covers sufficient ground and enough results are available to confidently advocate this method as the basis for the development work in this thesis.

Chapter 3

Functional Prototyping

3.1 Introduction

This chapter plus Chapter 4 lay down the theoretical foundations for the work which forms the bulk of this thesis. The system proposed in Section 1.1 requires theoretical support from a number of areas. These are *functional programming*, *parallel programming*, *computer vision* and *compilers*. Each of these is a vast subject in its own right so there is not enough scope in this thesis to cover all of them in detail. Several key ideas are introduced, however, which relate to the system described in Chapter 7, these are:

Functional languages form the basis for the prototypes that are specified in the methodology. A brief review of functional programming languages is given in Section 3.2, including parallel functional languages. A brief note is given in Section 3.3 to show how higher order functions in the chosen functional language Standard ML can be implemented in parallel.

Computer vision is described in detail elsewhere [17, 68, 69]. It has many real applications in industry [1] and there is a substantial community of researchers working on parallel implementations of computer vision [72]. Parallel computer vision algorithms were briefly covered in Section 2.2. One problem at the core of this thesis is the development of computer vision algorithms in functional languages, so an attempt is made in Section 3.4 at a partial classification of vision algorithms using working Standard ML code as examples.

Compilers again cover a vast range of research activities from parsing and language definition to the semantics transformations that define the conversion from source code to executable code [206]. The main compiler issue in this thesis concerns the translation of Standard ML prototypes into Occam2 implementations. This is essentially a compilation process although it is performed by hand. An attempt is made to validate this process in Section 3.6 by examples of code in SML and Occam2.

Parallel programming is another vast area of endeavour that this thesis relies upon for inspiration and guidance. Again, there is not enough room to cover the fundamentals. Performance modelling, however, is a specialised activity associated with parallel program construction which is covered in some depth in Chapter 4 since the proposed methodology relies quite heavily upon performance predictions to guide program development.

This is combined in Section 4.2 with the other sections into a complete example of the methodology at work.

3.2 Functional Languages

There are many general introductions to functional languages [12, 40, 57, 207], their implementation schemes [208]. Their importance has also been stressed [209].

Functional programming has its roots in the λ-calculus developed by Alonzo Church in the 1930s to develop a theory of computability [21]. The λ-calculus has remained a highly influential formalism in theoretical computer science [210] spawning many variants such as typed λ-calculus [211]. Indeed, this calculus can be argued to occupy a central rôle in many theoretical investigations, for instance many systems are proven to have computational completeness simply by showing they can emulate λ-calculus [212].

To base a programming language upon a formalism such as λ-calculus requires an evaluation scheme for reducing a general expression into a form where no further operations are possible (called a normal form for the expression), which constitutes termination. There are several evaluation orders for λ-calculus:

Normal order (or call by name) substitutes entire argument expressions for bound variable names in function bodies. This has the disadvantage of potential multiple evaluation of substituted expressions but can possibly terminate where other evaluation orders fail (the second Church-Rosser property [213]).

Applicative order (or call by value) evaluates argument expressions prior to substituting in function bodies. This has the potential for redundant computation of arguments which are discarded but does not result in multiple evaluation and so is generally more efficient than normal order. In addition, the arguments need not be reduced completely to normal form, usually the outer abstraction of the argument is reduced leaving potential redexes (reducible expressions) in the argument's body.

Lazy evaluation (or call by need) operates like normal order but multiple copies of the argument expression are tagged for sharing so that when one copy is reduced, all other instances are automatically reduced. This combines the benefits of normal and applicative order evaluations.

Normal order is generally too inefficient for programming language implementation but variants of both applicative and lazy evaluation have been used in many functional programming languages. Applicative order languages include Standard ML [15] and Hope [214] and examples of lazy languages include Miranda[1] [215] and Haskell [216]. Combinations of the two are also possible, for instance Hope provides both strict and lazy list constructors.

The evaluation order, however, only provides the core semantics of the language. Functional languages are eminently suitable for many sound programming language features which aid the development of correct software. These include the following but note that these are not restricted to functional languages but have become an integral part of functional language technology:

Functions as first class objects come naturally to functional programming languages since this stems directly from λ-calculus. This allows functions to be passed as arguments and returned as results of functions facilitating highly abstract programming.

Partial application is also a natural consequence of functional languages which include currying of functions and allows new functions to be built from existing ones by selective instantiation of arguments, encouraging reuse of existing code.

Pattern matching is a more succinct syntax for conditional execution than, for instance, nested conditionals and also allows functions to represent the data types they operate on more naturally.

Type systems which are powerful and flexible can be built into functional language compilers. These include *polymorphic* type checking [217] which allows functions to operate on abstract data, again promoting the use of highly abstract programming. *Type inference* [218, 219] is also possible which supports the programmer by deducing types of identifiers from their context. In general, strong and static type checking is preferred for functional languages.

[1] Miranda^{TM} is a trademark of Research Software Ltd.

Automatic memory management is also a natural consequence of the evaluation schemes used in functional languages [208]. This requires the use of a garbage collector [220] but insulates the programmer from memory resource issues.

Side-effect free programming is provided by *pure* functional languages but others provide mutable variables such as Standard ML's `ref` constructor for reference variables. The advantages of programming with side-effect free languages are obvious, program objects are insulated from what is occurring in other parts of the program and are thus less likely to be accidentally altered.

Denotational programming is possible in a limited sense in functional languages, although the separation of declarative and operational semantics is not perfect, the denotational meaning of most functional languages is tied up with the evaluation scheme and other operational considerations. Functional languages can also provide algebraic programming [221, 222] in a limited capacity but fall short of being fully algebraic languages.

The absence of side-effects is often referred to as *referential transparency*. There is actually no clearly defined semantics for this term [223], it was originally used by Quine in the 1950s [224] but the formal definition in terms of substitutivity and the informal definitions in the literature in terms such as definiteness, absence of side-effects or determinacy do not coincide and are not equivalent to each other.

The operational semantics of functional languages can likewise be divided up into several categories:

Stack-based evaluation schemes were originally proposed by Landin in 1964 [225] and was a major breakthrough in functional language technology since it allowed the use of functional languages on traditional von Neumann architectures. This method, called the SECD machine, is based around four stacks and can be used for both applicative and lazy evaluation.

Graph reduction arises from viewing lambda abstractions as nodes in a program graph and some sophisticated methods of evaluation have been developed borrowing techniques from graph theory [208]. This is particularly useful for lazy evaluation since the graph structure provides a natural method for representing subexpression sharing. Problems with free variables in function bodies can be overcome using *combinators* [226] which are simply functions with no free variables.

Term rewriting [227] based on the direct evaluation of structural operational semantics is also possible [22] and is the basis for the evaluation scheme of Standard ML [15].

3.2.1 Parallel Functional Languages

The properties of evaluation order independence and the lack of side effects make functional programs ideal for parallel implementation [228]. The main problem is that parallelism is present throughout functional code and the main obstacle to efficient parallel performance is deciding where within the program the most suitable parallelism is present. Parallel implementations of functional languages fall between the two extremes of *implicit* parallelism where the source is an ordinary sequential functional program and the natural parallelism in the program is exploited and *explicit* parallelism where the programmer tells the compiler where parallelism is to be exploited. Both have drawbacks, implicit parallelism suffers from the high computational complexity of search algorithms needed to extract parallelism and explicit methods do not sufficiently insulate the programmer from architectural and other low-level issues.

Methods of evaluating parallel functional languages include:

The dataflow model views programs as directed dataflow graphs [194]. The nodes perform computation and are *fired* when data is available on all their input arcs which transport data. Parallelism arises from several nodes firing simultaneously which can operate at all levels of granularity. The graph structure gives a simple mapping to the graph reduction model for functional languages. Languages which operate on this principle include VAL [229], Id [230] and SISAL [231] which are all fully implicit functional languages but implement different functional language features and use different sources of parallelism, all are executed in a dataflow model, however.

Parallel graph reduction is based around concurrent reductions of a program graph and is probably the most common method of executing a parallel functional language. Since performing a reduction can produce more reductions this method is very dynamic requiring some method of controlling task spawning such that some processors are not starved of work while others are overloaded. There are many languages and systems based on this idea including; ALICE [232, 233] which is based on Hope and has subsequently been upgraded in the Flagship system [234], the $<\nu\text{-G}>$ machine [235] which compiles Lazy ML programs down onto the G-machine abstract machine, GRIP [174, 236] which comprises specialised MIMD hardware for evaluating Haskell programs.

All of these are fully implicit approaches. Concurrent Clean [144] is a more explicit approach requiring programmer annotations as does Concurrent Haskell [237].

Other more explicit approaches include paraML [238, 239], Caliban [149] and Hudak's parafunctional programming [168], all of which place the burden of efficient parallel performance on the programmer. The alternative method of using algorithmic skeletons has already been reviewed in Section 2.3.

3.3 Some Simple Skeletons in Standard ML

Section 2.3 reviewed many interpretations of the algorithmic skeletons concept. This is visualised in this thesis as higher order functions (HOF) in Standard ML relating to specific parallel constructs in Occam2 on a transputer surface. The code and data required to instantiate the HOF appears in the parallel implementation as sequential components, executed outside of the parallel construct or within it as part of a decomposition. This section gives some simple examples of such constructs which form the basis for the following section where several computer vision algorithms are constructed.

3.3.1 Mapping a Function Over a List (`map`)

The `map` function is one of the most basic higher order functions. This maps an arbitrary function over a list:

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

In this form there is no parallelism since the recursion order is fixed. If the recursion is unfolded, however, the form of the calculation can be seen as:

```
(f h1)::(f h2)::(f h3):: ... (f hN)::[]
```

Provided the order of the elements is preserved, which satisfies the semantics of the list constructor (`::`), the function `f` can be applied to the elements in any order which is a form of parallelism that can be exploited by many types of parallel machine. On an SIMD machine, for instance, the `f` function would be the shared instruction steps for the processing elements and the list elements would be the distributed data. On MIMD machines the function `f` would be programmed into each distributed processing element and the list elements distributed for processing, there are several schemes for doing this.

3.3.2 Filtering a List (`filter`)

Another simple function common to many functional languages is the `filter` function which applies a predicate to each element of a list and retains only those elements which are `true` under the predicate:

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

This is essentially similar to the `map` function but with the complication of the conditional which is associated with the list constructor. Unfolding this is less informative due to the influence of the predicate:

```
(if p h1 then h1::(if p h2 then ...) else (if p h2 then ...))
```

This function can, however, be transformed into an equivalent form:

```
fun pmap p [] = []
  | pmap p (h::t) = (p h,h)::(pmap p t)

fun gettf (tf,value) = tf
fun getvalue (tf,value) = value

fun filter' p l = ((map getvalue) o (filter gettf) o (pmap p)) l
```

In this form the `p` predicate is applied to the list using a modified `map` function which tags the elements with the results of applying the predicate and can be parallelised as before. There is then a sequential overhead where the output list is constructed from the tagged elements. The efficiency of the parallel scheme, here, depends on the bulk of the work being carried out by the `p` predicate.

3.3.3 Folding the Elements of a List (`fold`)

Folding a list can be viewed as replacing the list constructor with another (binary) function. If the function is non-associative then for a list of n elements this can be carried out in $(n-1)!$ ways. There are two extremal ways corresponding to always folding from the left and always folding from the right:

```
fun foldl ff def nil    = def
  | foldl ff def (h::t) = foldl ff (ff def h) t

fun foldr ff def nil    = def
  | foldr ff def (h::t) = ff (foldr ff def t) h
```

These functions are intrinsically sequential since the order of function application is fixed. If the `ff` function is associative, however, the order of application is unimportant and parallelism exists:

```
fun folda ff (h1::h2::nil) = ff h1 h2
  | folda ff (h::t) = ff h (folda ff t)
```

This is quite often envisaged as a divide and conquer tree. Assuming an auxiliary function `split` which divides a list at some suitable interval (usually the middle) the list can be recursively divided up until in the base case there are only two elements:

```
fun split _ [] = ([],[])
  | split 0 l = ([],l)
  | split n (h::t) =
    let
      val (l1,l2) = split (n - 1) t
    in
      (h::l1,l2)
    end

fun pfolda ff (h1::h2::nil) = ff h1 h2
  | pfolda ff l =
    let
      val (l1,l2) = split ((length l) div 2) l
    in
      ff (pfolda ff l1) (pfolda ff l2)
    end
```

In this divide and conquer scheme the `ff` function is both the base case and the "conquer" operation. A similar scheme to that devised for `filter` can also be carried out for `folda`. In this case the fold is partially executed by mapping the fold function `ff` over pairs of list elements. This can then be carried out iteratively until only one element remains:

```
fun fmap ff (h1::h2::t) = (ff h1 h2)::(fmap ff t)
  | fmap ff l = l

fun pfold ff (h::nil) = h
  | pfold ff l = pfold ff (fmap ff l)
```

There could be several parallel implementations of this, for instance the `fmap` function could be treated as for `map` above, although it might be desirable to merge the `pfold` iterations, this function is very similar to a `map` operation and successive iterations could be viewed as a concatenated list, once time dependencies within the list are sorted out.

3.3.4 Function Composition

Function composition is often viewed, in the above schemes, as a means of pipelining processes together. Thus two functions can be composed to create a new function:

```
fun filtermap f p l = ((filter p) o (map f)) l
```

These could be physically implemented on separate processing elements or on the same processing element sharing a common set of workers. Using the map scheme described for `filter`:

```
   (filter p) o (map f)
== ((map getvalue) o (filter gettf) o (pmap p)) o (map f)
==  (map getvalue) o (filter gettf) o (pmap p) o (map f)
==  (map getvalue) o (filter gettf) o (pmap (p o f))
```

This shows how the combined function could also be implemented in a single parallel process by combining the predicate `p` and the map function `f`.

3.3.5 Representation Issues: Lists Versus Arrays

One of the problems in prototyping vision algorithms in functional languages is that most images are two-dimensional in nature. One possible means of handling this data is to represent a 2D array of pixels with nested lists:

```
type IMAGE = int list list
```

The complication here is that not all the basic operations on lists can be nested, only those that can be partially or fully applied to give a value of the same type as one of their arguments, such as `map`, `flatten` or `fold`:

```
- map (map I) [[1,2],[3,4]];
val it = [[1,2],[3,4]] : int list list

- flatten (flatten [[[1],[2]],[[3],[4]]]);
val it = [1,2,3,4] : int list

- foldl (foldl add) 0 [[1,2],[3,4]];
val it = 10 : int

- foldr (foldr add) 0 [[1,2],[3,4]];
val it = 10 : int
```

More complex list operations have to be built from the non-nested list equivalent. An example of this is the `member` function which tests for list membership:

```
fun member _ [] = false
  | member m (h::t) = (m = h) orelse (member m t)

fun memberll _ [] = false
  | memberll m (h::t) = (member m h) orelse (memberll m t)
```

Most of the vision algorithms discussed in this thesis have been implemented using either explicit decomposition of iconic data or a small set of higher-order functions over nested lists, the most salient being `mapll`, `map2ll`, `foldlll`, `foldrll`, `filterll` and `unzipll`[2].

An alternative, however, is to use the array facility which is provided by most functional languages. New Jersey Standard ML provides an `array` type:

```
- Array.array (10,0);
val it = [|0,0,0,0,0,0,0,0,0,0|] : int array
```

This is an imperative array with routines to extract and update array elements, create arrays by size and default value, by calling a function to initialise values and by converting lists into arrays. There is also a `length` function. It is thus possible to build up the same set of functions for arrays as for lists.

From the experience of developing vision algorithms in SML in this thesis, it is actually quite easy to substitute arrays for lists to achieve the same algorithm. This could be formalised by developing two abstract data types, one using lists as its internal representation, the other using arrays.

The main problem is that the space and time complexities of the two ADTs are completely different. Lists admit a constructor so that adding one element to the head of a list can be done in unit time, for arrays this operation is $O(size\ of\ array)$. Arrays admit a unit time update whereas for lists this operation is $O(length\ of\ list)$. This creates problems since the methodology requires a close correspondence between the space and time complexities of the prototype and the implementation.

This is compounded, in this thesis' work, by the fact that prototypes are list-based and implementations are array-based. This is justifiable, however, on the grounds that the prototypes are intended to sort out algorithmic details as much as performance issues and the list-based equivalents are far more tractable to analysis and transformation than array-based versions. Conversely, arrays are necessary in the implementation because of the need for efficiency in processing algorithms which are both data and code intensive.

To relate the performance of arbitrary list-based and array-based algorithms is difficult because the number of each type of basic operation has to be known to make a correction

[2] The suffix `-ll` is used here to mean the equivalent of a higher-order function on lists generalised to lists of lists.

based on the times or complexities of these operations. These counts will be compounded by numerous nested conditionals which will individually have to be counted or summarised and even all possible combinations of each choice which define a tree of conditional choices.

There are conditions under which an accurate comparison can be made, however. Consider mapping a function over a list and an equivalent array:

```
map ff list1
```

```
ff ** array1
```

Where `**` is a BMF style (see Section 2.3.2) infix `map` operation on arrays. Taking the time to process the identity function as the list or array overhead:

```
fun I x = x
```

So that $T_{list}(f) = time(\texttt{map f list1})$ and $T_{array}(f) = time(\texttt{f} ** \texttt{array1})$. If $T_{list}(I) \ll T_{list}(ff)$ and $T_{array}(I) \ll T_{array}(ff)$ then the computation is dominated by the application of the ff function and the list and array representations will have the same processing times. Although this only works in very simple cases it is a general principle that if the loop (or recursion) overheads are small with respect to the time to process the loop (or function) body then differences between the overheads can be ignored. In practice the differences between array and list access can be significant, particularly if either insertion or access dominates the computation.

3.4 A Selection of Vision Algorithms in Standard ML

Applications in vision systems involve a wide range of operations from low-level, pixel-based operations to high-level operations such as heuristic search. In the past, functional prototyping techniques have been applied to a small set of algorithms in computer vision [52, 55, 61, 240, 241] and some success has been achieved in these limited domains. In order to claim that these methods are more widely applicable, a set of algorithms is presented which are a fair cross-section of classical methods in image processing and computer vision. Although not state-of-the-art, these algorithms are representative of wide classes of algorithms in terms of the way they manipulate data. A small example of each algorithm is given which runs in Standard ML and some comments are made about how parallel implementations could be derived from these prototypes. These algorithms have already been published in Scaife [62].

The traditional role of prototyping in software engineering is in sorting out requirements analysis and user requirements before the main implementation work can begin [242]. The

```
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
```

Figure 3.3: Image Negation in SML

role of prototyping in parallel systems development can be viewed as similar to this in that many of the issues in the application can be resolved before going to a parallel solution. This can be important since the development and maintenance of parallel code is far more costly than its sequential equivalent. In this sense the prototype could be considered as a specification for the parallel implementation.

Although the algorithms presented here are not all the basis for parallel implementations, they were developed as prototypes suitable for conversion into parallel form. Thus they allow testing of the algorithmic structure, determination of suitable parameters, experimentation with different versions, all without the complexity of writing code for parallel machines.

3.4.1 Operations on Single Pixels

In a single pixel operation, each pixel value is modified independently, for example when creating the negative by reversing the intensity of all the pixels in an image. Another widely applied example is global thresholding, applied for example to edge maps to create binary images (edge, no edge) for further processing. Other examples include type conversion and scaling, for example by a logarithmic function.

The SML implementation of this is the application of nested `map` operations to a `list list` representation of the image, an example of this is illustrated in Figure 3.3.

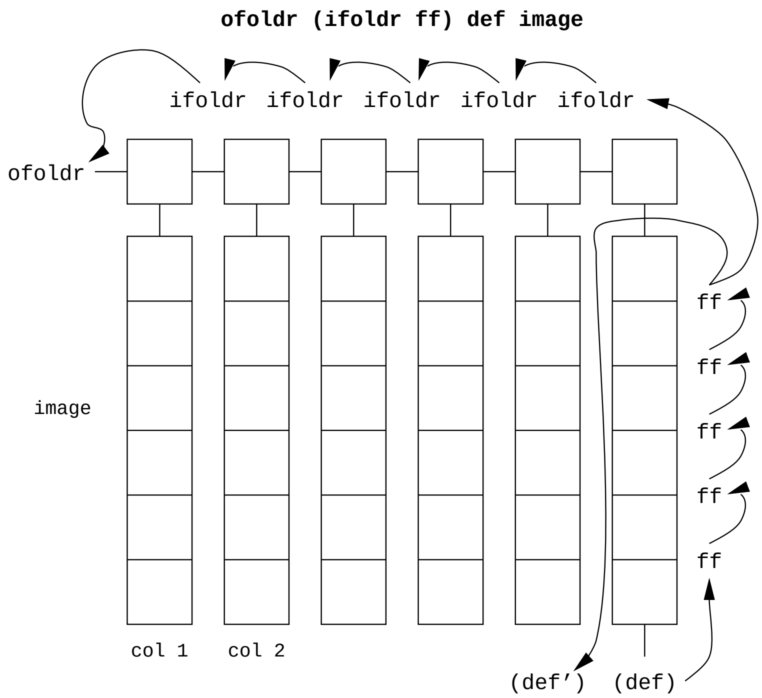
Here, a higher order function is created (`local_pixel`) which encapsulates the structure of the required operation but not the details of the calculations. The particular operations for a given pixel are then defined, (`negate` and `quantise_pixel`), and then used to instantiate the higher order function to create the desired operations. This is a representative example of code development in functional languages. One is always looking for abstractions, commonality between different parts of the code to maximise the recognised benefits of code reuse, such as reducing the amount of time spent debugging.

Parallel implementations of this type of process are easy to devise, for example the outer map could be parallelised as indicated in Section 3.3.1. Some arrangements do have pitfalls, however, for instance if too fine a decomposition is used such as decomposing over individual pixels the parallel operation quickly becomes communications bound. It is generally advisable to decompose over geometric regions rather than individual pixels.

3.4.2 Operations Which Collate Groups of Pixels

In this case, statistical or other values are computed on either the whole or a subsection of image data. Widely used examples include the computation of the mean and standard deviation of groups of pixels, or the production of a histogram of grey levels of an image. Since these operations involve data collection and combination, the application in SML demands a fold function in some form. This can be encapsulated in a HOF which `fold`s over the pixel values in a rectangular image, this is illustrated in Figure 3.4:

```
fun fold_image ff def image = foldr (foldr ff) def image
```



The default value is the starting point for the computation. The `ff` function is applied by the inner fold (`ifoldr`) along a column (or row depending on the order of the lists). This then becomes the default value for the outer fold (`ofoldr`) which uses (`ifoldr ff`) to fold along the outer row (or column). Thus each successive row is folded until the result value is returned by the outer fold.

Figure 3.4: Illustration of the Operation of `fold_image`

This can be used, for example, to compute the mean value of an image from an incremented accumulation of the intensity values and number of pixels respectively:

```
fun inc (s,c) (h:int) = (s + h,c + 1)
```

```
fun calc_mean (s,c) = real s / real c

fun mean image = calc_mean (fold_image inc (0,0) image)
```

Similarly, the number of pixels of each intensity value in an image can be accumulated as a list of tuples (intensity value, number of pixels), the ordering of the intensity histogram is maintained by the `update` function:

```
fun update [] (v:int) = [(v,1)]
  | update ((v1,c1)::t) v =
    if (v = v1)
      then ((v1,c1 + 1)::t)
      else
        if (v < v1)
          then ((v,1)::(v1,c1)::t)
          else (v1,c1)::update t v

val makehist = fold_image update []
```

If the `fold` operator is associative then parallelism can be extracted using a divide and conquer strategy [132, 133, 243, 244]. If the operator is not associative then the algorithm is virtually sequential in nature. In the examples above the operators `inc` and `update` are both associative and commutative and so could easily be implemented in parallel.

Many applications in image processing, however, can be specified in terms of `fold` functions but with non-associative operators. In these cases it is difficult to see how useful parallelism could be achieved where the `fold` function is the centre for parallelism. Some algorithms may have enclosing constructs or contain constructs which could be parallelised with the `fold` implemented sequentially, for instance the `update` function above could be implemented in parallel, or the `fold_image` function may be part of pipeline processing multiple images.

Other algorithms may be transformable into an equivalent form with more attainable parallelism or may be related to another method which is more easily parallelised. At least, for the purposes of the prototype, this situation would be spotted quickly and efficiently during prototype development and not during more difficult parallel system development.

3.4.3 Geometric Transformations

In vision, as in graphics, much use is made of geometric transformations in 2D and 3D space. Transformation of lists of co-ordinates is straightforward, since this is equivalent to an independent pixel operation. A transformation function (matrix multiplication) can be mapped onto each co-ordinate independently to produce a new list of co-ordinates. Transformation from one 2D image into another is more difficult, since this leaves the values unchanged, but effectively alters the indexing function. This could also be viewed as a transformation of a list of co-ordinates since the iconic image could be converted into such a list, transformed and then the transformed list could be converted back into an iconic image. It is this last step which is harder since the transformed co-ordinates may not map exactly onto a 2D grid. Another point is that a distinction has to be made between transformations which do not alter the values of pixels and transformations which do. Here, the former type of transformation is examined.

The simplest example is to reflect an image about a horizontal or vertical axis by reversing the row or column lists:

```
fun hreflect image = rev image
fun vreflect image = map rev image
```

Another simple example is to rotate in 2D by 90 degrees by transposing rows and columns (in an array structure the x and y indices would be swapped):

```
fun transpose (nil::_) = nil
  | transpose image = (map hd image)::(transpose (map tl image))
```

Rotation by arbitrary angles requires a mapping for each individual pixel. Calculating the co-ordinates of a transformed pixel is easy but arranging for the transformed pixels to be retained in iconic form requires more or less random access to the image. The usual method is to scan over the co-ordinates of the output image, back calculate the co-ordinates in the input image that pixel came from and interpolate the pixels in the input image to give the value for the output image. A simplified version of this, which does no interpolation in the input image is shown in Figure 3.5.

In this code the HOF `mapixyll` is a `map` over lists of lists, maintaining the `(x,y)` index of the value being passed to the `map` function. The application function `transpix` is applied to each co-ordinate in the output image (which for convenience is taken to be the same as the input image). This function calculates the equivalent co-ordinate in the input image using the (inverse) 3 by 3 affine transformation matrix in homogeneous co-ordinates `mA` and the matrix vector multiplication function `Mmatvec3`. The input image is then scanned row by column using `getxy` to retrieve the pixel value at that co-ordinate. The image can thus be rotated, scaled or translated within the original frame by generating suitable transformation matrices although aliasing is a problem when there is no interpolation.

The HOF is a simple `map` in this case and so is suitable for processor farming over geometric regions. This corresponds to a decomposition over the output image, with each processing

```
fun mapixyll _ _ _ nil = nil
  | mapixyll ff x y (h::t) =
    (mapixyll' ff x y h)::(mapixyll ff x (y + 1) t)

and mapixyll' _ _ _ nil = nil
  | mapixyll' ff x y (h::t) =
    (ff (x,y,h))::(mapixyll' ff (x + 1) y t)

fun getxy (x,y) img =
  if (x < 0) orelse (x >= cols) orelse (y < 0) orelse (y >= rows)
    then 0
    else nth(nth(img,y),x)

fun transpix image mA (x,y,v) =
  let
    val (imgx,imgy,_) = Mmatvec3 mA (real x,real y,1.0)
  in
    getxy (floor imgx,floor imgy) image
  end

fun transform mA image = mapixyll (transpix image mA) 0 0 image
```

Figure 3.5: Geometric Transformation in SML

element transforming the co-ordinates in the input image that land in that portion of the output image.

Sometimes, the basic `map`, `fold` and `filter` functions and their `list list` equivalents have to be "embellished" with additional features for particular applications. Two were commonly found in image processing. One was an index for the list elements, for instance `mapll` might need to keep track of the `(x,y)` index of the pixels for geometric calculations, these equivalents are labeled with an `-ixy` suffix. The other is the extension to operate over two or more lists simultaneously, these are suffixed with the number of lists so `map2ixyll` is a `map` function extended to map over lists of lists, keeping track of the `(x,y)` index of the elements and mapping over two lists at the same time.

The implications for parallel versions are minimal. Mapping over two lists could be viewed as mapping over a list of tuples and the indexing function has very little overhead.

3.4.4 Operations Using Local Neighbourhoods

Another fundamental set of operations with many applications in vision systems are local window operations. These are operations where the region surrounding each pixel, usually a square neighbourhood, is operated on to give a new value at that pixel. These include gradient calculations, filtering and smoothing functions such as Gaussian filtering [245], pattern matching [246] and local function fitting routines [247]. Figure 3.6 shows the basic process.

A large subclass of these algorithms concerns the 2D convolution operation in which a mask

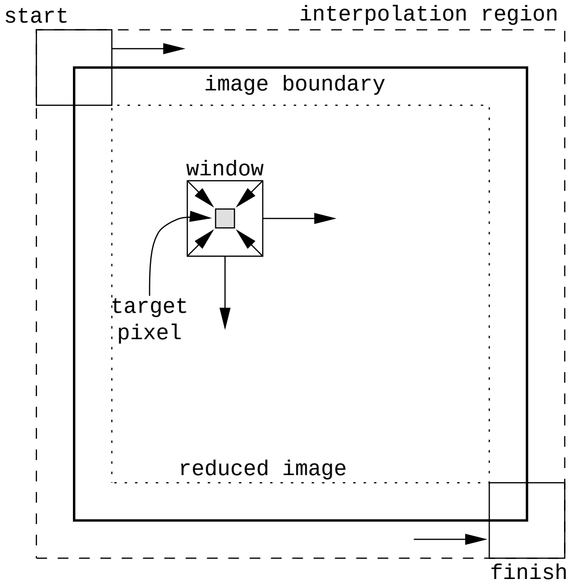


Illustration of general local window operation. The window is scanned over the image and at each point the pixels within the window are processed to give a value. These values form a new image, this coincides with the original image if the value corresponds to the centre of the window, otherwise there is a shift. As can be seen from the start and finish windows values are required from outside the image if the output image is to be the same size as the original image, these can be extrapolated from within the image or a default value used. An alternative is to only map the window so that it always lies within the original window but in this case there is a reduction in image size.

Figure 3.6: Illustration of Local Window Operations

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 [248], an edge enhancement regime that includes Gaussian filtering and edge thinning giving good localisation of edges. The Canny edge detector involves several phases, however, so the process is illustrated in SML using the Sobel edge detector [245], which 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. The code to do this is in Figure 3.7.

The procedure is the same as before. A HOF is developed which has the basic, top level structure of the local window operations we want to perform (`local_window`). The specific operations are then defined that make up the intended operator (`sobeloper`) and instantiate the 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`).

What has been defined here is a completely general local window operation function and although the Sobel edge detector is a single operation based on combining two convolutions,

```
(* 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 convol ff def flt img = foldr2ll ff def flt img
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
```

Figure 3.7: Sobel Edge Detector in SML

any similar operation can be defined such as Gaussian filtering or the Canny edge detector by instantiating with the relevant operators and composing successive operations together.

The top level of a Canny edge detector in SML is illustrated in Figure 3.8. In this example the three phases of the Canny edge detector are defined separately and then glued together using function composition to give the overall functionality. Note that this implementation of the Canny edge detector does not have full hysteresis thresholding but simply performs local non-maximum suppression.

The Canny edge detector has been studied in combination with hand-coded implementations in Occam2 [240]. A number of issues are addressed such as the decomposition of the input image, which could be pixel, row or geometric. The best parallel implementation produced by this work was based on preloading of the entire image onto each worker followed by farming of geometric area definitions.

3.4.5 Image-to-feature Transformation Using Hough Transformation

The Hough transform is a technique for detecting features in images by transforming into a space in which the features are mapped onto points in transformation space [81, 249]. Histogramming techniques can then be used to identify peaks in transformation (or Hough) space corresponding to features in the original image [250]. This technique is a special case of

```
val smooth_x = convol gaussianX
val detect_y = convol dgaussianY
val smooth_detect_y =
  (local_window detect_y) o (local_window smooth_x)

val smooth_y = convol gaussianY
val detect_x = convol dgaussianX
val smooth_detect_x =
  (local_window detect_x) o (local_window smooth_y)

fun smooth_detect_xy img =
  zipll (smooth_detect_x img) (smooth_detect_y img)

fun non_maximum_suppression img =
  local_window non_max_operator img

val canny =
  non_maximum_suppression o smooth_detect_xy
```

Figure 3.8: Canny Edge Detector in SML

the Radon transform [251] and can be applied to straight lines [252, 253], ellipses [254, 255], arbitrary shapes [256, 257] and 3D surfaces [258]. The main disadvantage is the amount of memory required to store the transformation space which can be of high dimensionality. Adaptive methods have been used to limit the amount of required memory [259, 260].

Note that this is one example of a range of transformations that can be performed on images, another example is the discrete Fourier transform. For the purposes of this partial classification of computer vision algorithms, the common structure that these transformations have is that the entire image is converted into a transformation space such that, potentially, the entire input image may be required to calculate a value in transformation space. The Hough transform is used here since it is a technique which has been investigated thoroughly by computer vision researchers.

The simplest example of a Hough transform is a straight line detector [253] which works by mapping detected edges in an image according to their angle and the length of the normal from the line the edge lies on to the origin:

$$\rho = x \cos\theta + y \sin\theta$$

This gives a 2D transformation space (indexed by (ρ,θ)) in which straight lines in the original image are mapped to (ρ,θ) points in transformed space. A quantized transformation space is represented by a 2D histogram of (ρ,θ) values. This histogram can be thresholded to give the ρ and θ values for straight lines in the image. The core of a Hough transform for straight lines in SML is illustrated in Figure 3.9.

This code works by mapping the transformation equation as a function over the image giving

```
(* 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)
```

The function `map2ixyll` maps a function simultaneously over two lists of lists, the function is passed a tuple `(x,y,e1,e2)` consisting of the co-ordinate of the elements in the lists along with the two elements. These routines also use a general purpose ordered histogram `abstype` that stores a list of tuples against a single slot tuple and accumulates a cost value for each slot, `addHist t1 cost t2` adds tuple `t2` to the slot labelled `t1` and adds the `cost` value to the accumulator.

Figure 3.9: Core of a Hough Transform in SML

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.

The above code applies for the special case in which the angular information can be used to reduce the transformation equation into a one to one mapping. The same method could be used where a range of points have to be plotted in parameter space for each image point. The same first step would apply, `map2ixyll`, except that instead of creating an image of $(x, y, \rho, \theta, edgeval)$ tuples, the (ρ, θ) component would be replaced by a list of (ρ, θ) values generated by that (x, y) position:

$(x, y, [(\rho_1, \theta_1), (\rho_2, \theta_2), ...], edgeval)$

This is equivalent to a list of $(x, y, \rho, \theta, edgeval)$ tuples which could be histogrammed as before. It would be prudent in this case, however, to separate accumulation of the ρ and θ parameters so that the first accumulation gathered ρ values giving a histogram of ρ versus a list of $[(x, y, \theta, edgeval)]$ values. A second accumulation could be run for each unique value of ρ giving the sparse list of $(\rho, \theta, \sum edgeval, [(x, y), ...])$ values as before. Actually generating parameter space would be needed if, for instance, Gaussian filtering were required in Hough space prior to peak detection but this is a highly inefficient process where arrays would almost be essential for reasonable performance.

The Hough transform has been studied in a more complete form and has been implemented in SML and Occam2 [61]. The basic structure of the SML Hough is identical to the code in Figure 3.9. Two versions were written, however, to emulate the behaviour of different parallel versions, one based on a geometric decomposition of the original image and the other based on a geometric decomposition of parameter space. The latter was found to be more efficient after prototyping and performance modelling and was used as the basis for the parallel implementation.

Note that these two alternatives are equivalent to implementing the `map2ixyll` and `foldrll` HOFs in parallel. The map is easy to visualise but in this case since the function being folded by `foldrll` is both associative and commutative, similar techniques for implementing `fold` and `map` can be used. See Bratvold [40] for a more complete discussion of this point.

3.4.6 Image-to-feature Transformation Using Local Continuity Measures

Parallel implementations of the preceding classes of algorithms are relatively straightforward, though not trivial if optimum performance and load balancing are to be achieved. However, operations which involve local continuity, such as edge tracking or region-growing using local homogeneity measures are more difficult.

Because of this difficulty, prominence is given to a discussion of the SML prototyping of an edge tracking algorithm, parallel implementation is discussed in Section 7.2.3. This admits a highly efficient *sequential* $O(N)$ algorithm, where N is the size of the image and involves determining seed points in an edge enhanced image and following lines or curves by examining the neighbourhoods of the seed points for compatible pixels. Compatible neighbours then become candidates for further growing and the process continues until no more compatible neighbours can be found.

Due to the fact that processing an edge tracking algorithm requires more or less random access to the iconic image, list-based processing becomes highly inefficient compared with array-based implementations. It is still possible, however, to develop edge tracking algorithms using SML and, potentially, to infer parallel implementations from the SML prototype.

Edge tracking presents problems for a distributed memory parallel system because it is difficult to ensure that each processor is working on a unique portion of the algorithm, without shared parameters which have expensive communications costs or global merge processes for which parallel implementations are non-trivial. In the past high communications costs algorithms have been avoided and *gathering* operations have been viewed as sequential overheads to the parallel processes.

Three implementations of an edge tracking algorithm are presented which work on the edge strength and orientation maps produced by the Canny edge detector. The merits of each algorithm are discussed and likely parallel implementations are indicated.

List-based Processing of Iconic Data

An *imperative* expression of the edge tracking process is defined in Figure 3.10. Edge tracking is based on a similarity measure between pixels and the accurate detection of lines is a balance between the ability to reject noise points but to effectively negotiate discontinuities. Popular measures include metrics such as strength and angular constraints, segment curvature or on proximity to existing contours. The algorithm presented uses a very simple measure, all points over a given strength threshold are treated as seed points and a simple angular threshold is used to guide the growing process. This does not address problems such as those

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

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

Figure 3.10: Imperative Description of Edge Tracking

of lines that cross over or meet at vertices but the basic structure of the algorithm will serve for much more advanced methods.

This algorithm can be directly encoded in SML, the core of this algorithm is shown in Figure 3.11. 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. For efficiency, all the neighbours for each pixel are checked at each point by `fold`ing over a list of neighbouring point offsets and the resulting list of compatible neighbours is then added to the list of pending neighbours.

One point to note about this algorithm is that, in general, the order of points in the final lines will not be contiguous across the lines. This will affect later processing in some systems where point order is required, for instance finding the end points of the lines. There will, however, be a great deal of structure within the point lists derived by this algorithm which could be exploited by a subsequent sorting process. For instance straight lines are guaranteed to be ordered up to branches and intersections. The ordering is set by the fixed order in which the neighbours are checked and this could itself be used to improve the final ordering using a more rigorous regime. This might have implications for any parallel implementation, however.

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.

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

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

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

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

Figure 3.11: Edge Tracking Algorithm in SML

Having such a high complexity operation at the very lowest level of the algorithm causes the prototype to be very inefficient, this algorithm will process a 256 by 256 test image in about 232 seconds of user time. 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. The handling of the `V` list is a similar source of inefficiency.

Array-based Processing of Iconic Data

The above algorithm is, however, easily converted into array-based processing using an array facility such as New Jersey Standard ML's [261] `struct Array2`. 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, a similar method is used to construct a `V` array. The `a2img` operations are shown in Figure 3.12. This version processes the same image with the same threshold values as the list-based version in about 4 seconds on a Sun Sparc 10 machine.

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 and that when an error occurs in an array-based operation it can be much harder to track down

```
(* 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 _ = map2ixy seta2 0 0 edges angles

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

Figure 3.12: Array Based Operations for the Edge Tracking Algorithm

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

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

Figure 3.13: Alternative Description of Edge Tracking

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 for this reason.

List-based Processing of Sparse Point Lists

The final version of the edge tracking algorithm is based on an alternative description of the process, see Figure 3.13. This is much closer to a declarative description of the problem which states that the set of tracked edges is the set of sets of pixels such that each pixel is compatible with and a neighbour of at least one other pixel in the same set and with no other pixels in the other sets.

This algorithm works by starting from a list of singleton lines derived by thresholding the

```
fun genlist ll l =
  let

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

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

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

  in
    (l @ llc)::llnc
  end

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

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

Figure 3.14: Line Merging Version of the Edge Tracking Algorithm

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

Again, this algorithm can be directly encoded in SML, this is shown in Figure 3.14. The scanning of point and line lists is carried out by a series of nested `fold` operations. Overall, this program has a much more succinct source code than the previous versions showing the declarative nature of the specification.

The sequential performance of this algorithm is much faster than the list-based version but much slower than the array-based imperative tracker, the test image can be processed in about 15 seconds. The problem with this algorithm is that it is effectively $O(M^2)$ complexity, where M is the number of thresholded edges in the image, (each edge has to be compared with all other edges) which means that the parallel performance of the system will have to be very good to overhaul the fast imperative sequential algorithm.

Note, however, that the list and array-based algorithms are likely to be much more difficult to express in parallel, whereas this version could be parallelised simply by decomposing over all pairs of edges. Parallel implementations of edge tracking algorithms are discussed in Section 7.2.3.

3.4.7 Image Interpretation Using Non-iconic Structures

At the top level of a vision system, the necessity for parallelism is determined not from the volume of data, which can be relatively small, but from the complexity of the algorithms for high-level image interpretation, commonly exponential in the number of features. The data is stored in a variety of forms, including relational databases, relaxation grids, interpretation trees, association graphs and so on [1]. Parallel implementation is not difficult *a priori*, since general algorithms already exist to perform many of these functions. Instrumentation and evaluation is still paramount however, as the parallel performance is dependent critically on the nature of the data, to a much greater extent than in low-level processing of images.

As a meta-language SML is capable of handling these structures in an efficient manner since parsing and evaluation problems are very often expressed as tree search or graph matching algorithms. Here, we show one widely applied example of an algorithm to match features extracted from an image with features in the objects expected to be found in such images. The chosen example is the maximal cliques algorithm applied to a graph which associates image and model features. The nodes in the graph are based on single feature similarities. The arcs are based on pairwise geometric similarity between features [1].

A cliques finding algorithm is illustrated in Figure 3.15. In this code we represent a graph as a `SET` of nodes, `SET` is a set `abstype` with all the usual set operations such as `union` and `difference`, augmented with routines more familiar to list processing such as `hd` and `filter` operations; `sethd` and `setfilter`. A graph is therefore a `SET` of `node` values, for example the graph `G1` is built using the `SET` operations `create` and `insert` on a list of predefined `node` values:

```
val G1 = insert na (insert nb (create nodeeq)) : node SET
```

which are in turn tuples consisting of a node name (`nodename`) and a list of connected node names:

```
type node = nodename * nodename list
val na = (a,[b,c,d,e,f,g,h]) : node
```

The node names are simply strings:

```
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
```

Figure 3.15: Clique Finding in SML

```
type nodename = string
val a = "a:1" : nodename
```

The algorithm itself is well-known [262], 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.

Parallel implementations of this algorithm could simply be based on parallel implementations of set operations [263]. Alternatively, the algorithm is structured in a manner highly suitable for a divide and conquer skeleton. The split phase would be the `union` and `difference` operations and the merge phase would be the append (`@`) operation.

3.5 A Partial Classification of Vision Algorithms

The SML algorithms in Section 3.4 represent a cross-section of some very simple image processing and computer vision algorithms. Here, an attempt is made at a classification of these algorithms.

Figure 3.16 shows a possible division of the presented algorithms. Initially, they are divided into two sets, those based on *iconic* images and those based on *non-iconic* images. This thesis concentrates on the former so only one example of the latter is given (a *set-based* maximal cliques algorithm). Algorithms based on iconic data are further subdivided according to whether their output is an *iconic image*, a set of *features* or some *other type of data*.

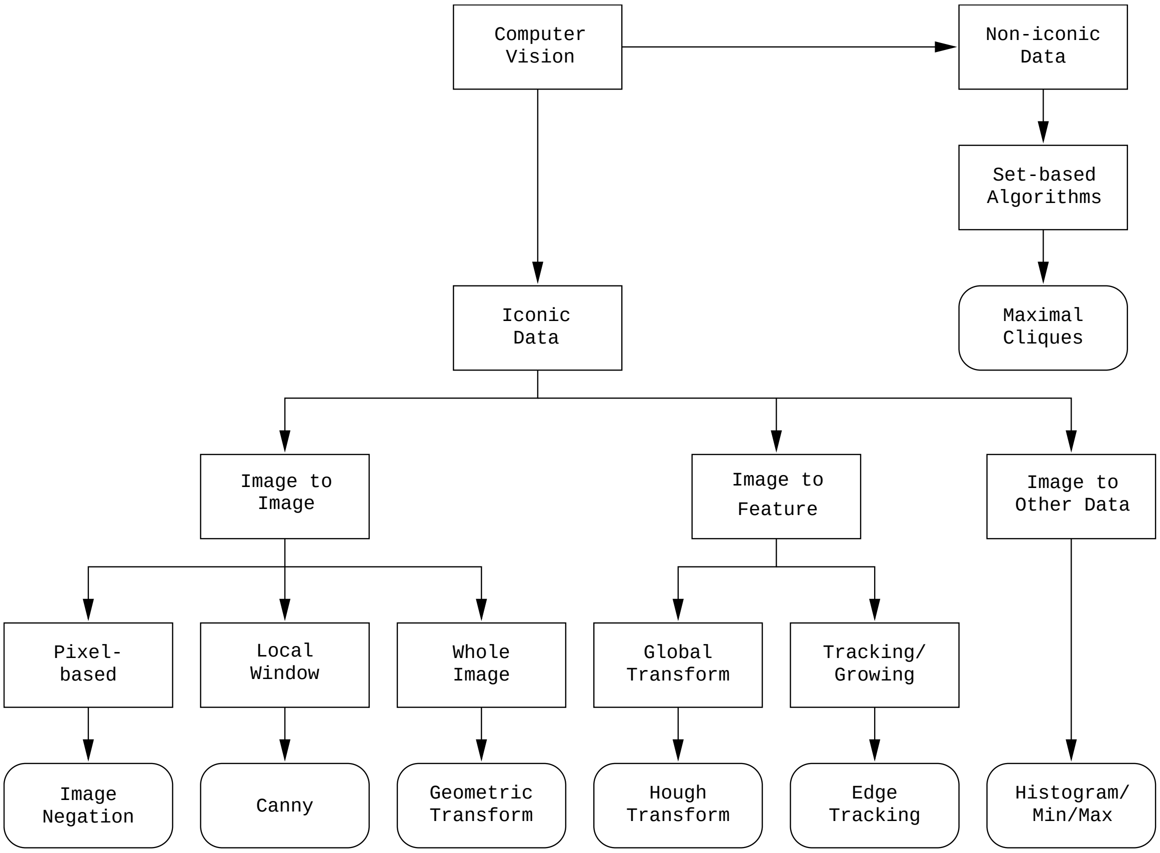


Figure 3.16: A Partial Classification of Some Vision Algorithms

Transformations between images are divided into those based on *pixel-wise* operations (image negation), *local window operations* (Canny edge detector) or *whole image transformation* (geometric transformation), for which one example is provided in each class.

Feature detection algorithms are represented by *global transformatio* (Hough transform) and *growing and tracking algorithms* (edge tracking). This is not a complete subdivision since other types of feature detection algorithm are possible.

Finally, images can be converted into any type of data (histograms, maximal/minimal values). Again, there are many possibilities under this class but these are not explored further.

This classification represents only a tiny fraction of the methods used in image processing and computer vision. Neither does it include combinations of the above methods which can be used in constructing complete systems. The algorithms included are those starting from a single iconic image using either local or global processing to generate either processed images or features from the input image. All of the algorithms in this these fall into one of these categories.

3.6 Translating SML to Occam2

3.6.1 Source Language Translation

One potential problem with the methodology suggested here is the *impedance mismatch* between the source and target language constructs. Previous work has included an SML to sequential Occam2 translator called `s2o` [58]. This, however, only implements a highly restricted subset of the full SML language and in any case cannot handle parallel constructs. A full parallelising compiler called `SkelML` [40] was written based on algorithmic skeletons which used `s2o` to generate the sequential Occam2 code and incorporated a library of parallel Occam2 harnesses to implement the skeletons. This compiler was not in a usable state for the purposes of this project, the limitations of the particular subset of SML implemented meant that it was not possible to implement the required vision algorithms. The variety of simple skeletons provided for parallel implementation were also not sophisticated enough to handle many of the complex resource issues that arose during the construction of the system described in this thesis. Instead, a set of informal rules were followed such that arbitrary SML could be converted into efficient Occam2 by judicious recoding of complex sections. This gave a sufficiently close correspondence between the two languages for the methodology to work but placed a considerable burden on the programmer.

3.6.2 Conversion of SML into Occam2

This is illustrated by showing excerpts from the Hough transform example discussed in Section 3.4.5.

Linear, sequential sections of code can be translated at the syntactic level. Consider the section of SML code in Figure 3.17 which can be directly converted into the Occam2 code shown in Figure 3.18.

There are a few pitfalls in this process, however. This sequence of `let` constructs in SML can be transformed into a single expression by a process of substitution but it is important to maintain the same pattern of single/multiple evaluation in the translation process. For instance, if the `ri` expression were substituted into the `rhostep` expression this would result in a double evaluation which would have to be duplicated between the prototype and implementation.

Another problem is that there are two real number representations in Occam2 whereas only one is provided by New Jersey SML. In terms of precision, `REAL64` is identical to the New

```
let
  val ri = real rinc
in
  let
    val theta = real angleval * 360.0 / 255.0
  in
    let
      val beta = (90.0 - theta) * pi / 180.0
    in
      let
        val rho = real x * cos beta + real yy * sin beta
      in
        let
          val thetastep = floor(theta * real tinc / 360.0 + 0.5)
        in
          let
            val rhostep = floor(rho * ri / 720.0 + ri / 2.0 + 0.5)
          in
            (x,yy,thetastep,rhostep,angleval)
          end
        end
      end
    end
  end
end
```

Figure 3.17: SML: Hough Transformation Code

```
INT thetastep, rhostep:
REAL32 theta, beta, rho, ri:
SEQ
  ri := REAL32 ROUND RhoInc
  theta := ((REAL32 ROUND frame[1][y][x]) * 360.0(REAL32)) / 255.0(REAL32)
  beta := ((90.0(REAL32) - theta) * pi) / 180.0(REAL32)
  rho := ((REAL32 ROUND x) * COS(beta)) + ((REAL32 ROUND yy) * SIN(beta))
  thetastep := INT ROUND ((theta * (REAL32 ROUND ThetaInc)) / 360.0(REAL32)
  rhostep := INT ROUND (((rho * ri) / 720.0(REAL32)) + (ri / 2.0(REAL32)))
```

Figure 3.18: Occam2: Hough Transformation Code

Jersey SML reals although in terms of processing time, `REAL32` runs (very approximately) 25-30% faster than `REAL64` for arithmetic operations on T800 transputers. Due to the influence of iteration using residuals as convergents the transcendental functions run about twice as fast for `REAL32` (although this is highly data dependent).

This represents a thorny problem since the space efficiency of the `REAL32` representation would be preferable (vision algorithms generally do not need the precision of 64-bit real numbers) but New Jersey SML supports only the 64-bit representation. The 32-bit representation is adopted in the Hough transform due to the large amount of intermediate data which limits the size of image that can be processed. An estimated discrepancy of about 49% in the time to process the core of the Hough transformation in 32-bit as opposed to 64-bit real numbers is found and the assumption is made that this does not affect the validity of the prototype.

Minor differences between the prototype and implementation are tolerable, however, since it is the ratio of computation between different parts of the program that is important, not absolute processing times. Provided both prototype and implementation are consistent within themselves these ratios should be preserved to within acceptable limits.

This is an important observation since there will inevitably be differences of one kind or another between two equivalent programs in different paradigms, for instance the `angleval` value in the SML version is provided as a function argument by mapping over a list whereas the Occam2 version has to explicitly index an array within an iterative loop. The code implementing the map function is the logical equivalent of array indexing but they occur at different points in the computation.

Note, however, that it is often the case that the amount of computation in the differences is outweighed by the processing in equivalent code sections, for instance, in the above code, the time to index an array is far smaller than the time to process the floating point operations.

The direct equivalence between sequential code segments can be extended by preserving function (`PROC`) boundaries. This does not apply to recursive functions in SML which translate into iteration over suitable variables, but programmers will generally use function abstraction as a mechanism for structuring code and this can be carried across directly from the prototype to the implementation. A modest example is illustrated in SML in Figure 3.19 and its Occam2 conversion in Figure 3.20. In this example, the `rshdst` function contains a nested function `shproj`, which in turn contains `within`.

The external function calls are not important, this code is given to exhibit how structure is preserved in translation. Here, tuples are translated into separate variables, resulting in unwieldy syntax in the implementation and conditionals are translated across directly, due to the identical semantics of conditionals in each language.

```
fun rshdst ((l1 as (l1p1,l1p2)):rline) ((l2 as (l2p1,l2p2)):rline) =
    let
        fun shproj
            ((l1 as (l1p1 as (l1x1,l1y1),l1p2 as (l1x2,l1y2))):rline)
            ((l2 as (l2p1 as (l2x1,l2y1),l2p2 as (l2x2,l2y2))):rline) =
            let
                fun within t = t >= 0.0 andalso t <= 1.0
                val omc1 = romc l1
                val pp1 = proj_pt_to_line omc1 l2p1
                val pp2 = proj_pt_to_line omc1 l2p2
                val wi1 = rptwithinln pp1 l1
                val wi2 = rptwithinln pp2 l1
            in
                if within wi1
                    then
                        let
                            val dp1 = rdistfn pp1 l2p1
                        in
                            if within wi2
                                then rmin dp1 (rdistfn pp2 l2p2)
                                else dp1
                        end
                    else
                        if within wi2
                            then (rdistfn pp2 l2p2)
                            else mmd
            end (* shproj *)
        val md1 = rmin (rdistfn l1p1 l2p1) (rdistfn l1p1 l2p2)
        val md2 = rmin (rdistfn l1p2 l2p1) (rdistfn l1p2 l2p2)
        val mmd = rmin md1 md2
    in
        rmin (shproj l1 l2) (shproj l2 l1)
    end (* rshdst *)
```

Figure 3.19: SML: Shortest Distance Function

```
{{{ PROC rshdst(l1x1,l1y1,l1x2,l1y2,l2x1,l2y1,l2x2,l2y2,dist)
PROC rshdst(VAL REAL32 l1x1,l1y1,l1x2,l1y2,l2x1,l2y1,l2x2,l2y2,
            REAL32 dist)

  REAL32 shl1l2,shl2l1,md1,md2,mmd:
  {{{ PROC shproj(l1x1,l1y1,l1x2,l1y2,l2x1,l2y1,l2x2,l2y2,shdst)
  PROC shproj(VAL REAL32 l1x1,l1y1,l1x2,l1y2,l2x1,l2y1,l2x2,l2y2,
              REAL32 shdst)

    REAL32 grad1,inter1,pp1x,pp1y,pp2x,pp2y,wi1,wi2,dp1:
    BOOL HV1:
    {{{ BOOL FUNCTION within(t)
    BOOL FUNCTION within(VAL REAL32 t)

      VALOF
        SKIP
          RESULT (t >= 0.0(REAL32)) AND (t <= 1.0(REAL32))

    :
    }}}
    SEQ
      romc(l1x1,l1y1,l1x2,l1y2,HV1,grad1,inter1)
      ProjPtToLine(l2x1,l2y1,HV1,grad1,inter1,pp1x,pp1y)
      ProjPtToLine(l2x2,l2y2,HV1,grad1,inter1,pp2x,pp2y)
      wi1 := rptwithinln(pp1x,pp1y,l1x1,l1y1,l1x2,l1y2)
      wi2 := rptwithinln(pp2x,pp2y,l1x1,l1y1,l1x2,l1y2)
      IF
        within(wi1)
          SEQ
            dp1 := rdist(pp1x,pp1y,l2x1,l2y1)
            IF
              within(wi2)
                shdst := rmin(dp1,rdist(pp2x,pp2y,l2x2,l2y2))
              TRUE
                shdst := dp1
        TRUE
          IF
            within(wi2)
              shdst := rdist(pp2x,pp2y,l2x2,l2y2)
            TRUE
              shdst := mmd

  :
  }}}
  SEQ
    md1 := rmin(rdist(l1x1,l1y1,l2x1,l2y1),rdist(l1x1,l1y1,l2x2,l2y2))
    md2 := rmin(rdist(l1x2,l1y2,l2x1,l2y1),rdist(l1x2,l1y2,l2x2,l2y2))
    mmd := rmin(md1,md2)
    shproj(l1x1,l1y1,l1x2,l1y2,l2x1,l2y1,l2x2,l2y2,shl1l2)
    shproj(l2x1,l2y1,l2x2,l2y2,l1x1,l1y1,l1x2,l1y2,shl2l1)
    dist := rmin(shl1l2,shl2l1)

:
}}}
```

Figure 3.20: Occam2: Shortest Distance Function

```
fun xysums (Sx,Sy,Sxy,Sxx,Syy) ((x,y):rpnt) =
    (Sx + x,Sy + y,Sxy + (x * y),Sxx + (x * x),Syy + (y * y))

val (Sx,Sy,Sxy,Sxx,Syy) =
    foldl xysums (0.0,0.0,0.0,0.0,0.0) coords
```

Figure 3.21: SML: Summations

```
{{{ PROC RCalcXYsums(X,Y,N,Sx,Sy,Sxx,Sxy,Syy)
PROC RCalcXYsums(VAL []REAL32 X,Y,VAL INT N,REAL32 Sx,Sy,Sxx,Sxy,Syy)

  REAL32 x,y:
  SEQ
    SEQ n = 0 FOR N
      SEQ
        x,y := X[n],Y[n]
        Sx,Sy := Sx + x,Sy + y
        Sxx,Sxy,Syy := Sxx + (x * x),Sxy + (x * y),Syy + (y * y)

:
}}}

Sx,Sy := 0.0(REAL32),0.0(REAL32)
Sxx,Sxy,Syy := 0.0(REAL32),0.0(REAL32),0.0(REAL32)
RCalcXYsums(xpts,ypts,pts,Sx,Sy,Sxx,Sxy,Syy)
```

Figure 3.22: Occam2: Summations

The conversion from recursion to iteration is again illustrated with a simple example. Figures 3.21 and 3.22 show how a summation over a set of 2D co-ordinates (part of a least squares routine) can be implemented by `foldl` with a suitable function in SML and using iteration in Occam2. Although this is a trivial example the conversion is quite general in that complex nested recursive functions are converted into similarly nested iteration. The only complication, as shown in the example, is that default values for recursion have to be replaced by explicit initialisation of variables. Mutually recursive functions are more problematical, however, but could be handled by additional control variables in the iteration.

The conversion of higher order functions into parallel constructs has been explained in more detail elsewhere [55]. Summarising this work, the `map` function is converted into a processor farm with one master processor and a linear pipeline of worker processors. This construct is simple to implement and has primitive load balancing but there is a practical limit to the length of the pipeline that can be used which is much smaller than the limitations of more dense arrangements of processors such as 2D grids or toroids.

The `map` function is translated into this construct by implementing the `map` function argument on the workers and decomposing the list over these workers. There are a number of design decisions that have to be taken at this point, however.

For low-level operations such as convolving a filter with an image, where communication dominates over computation, geometric decomposition generally works better than processor farming. For higher level operations, such as scanning a list of features and calculating transforms between pairs of features, computation dominates over communication and the load balancing inherent in processor farming becomes more effective relative to naive geometric decomposition.

Another issue is preloading of data. The list to be decomposed can be built into the work definition sent from the farmer to the workers or it can be distributed among the workers prior to beginning the computation. If communications times are a substantial fraction of the overall processing time, preloading of the list data may prove effective. This is partly because saturated communications can cause remote workers to be starved of work by workers physically nearer the farmer and also because optimisations can be made during initialisation, such as image compression, which is more effective when applied to a complete image.

Note also that preloading of data can be controlled at the source code level by program transformation, thus the following SML construct:

```
let
  f1 = f x
in
  map f1 l
end
```

corresponds to preloading, whereas the following equivalent section of code:

```
map (f x) l
```

corresponds to distribution of program data during computation. The idea here is that if the function's argument is applied in the skeleton instantiation `(f x)` then the implication is that the argument should be distributed with the list data whereas in the first version the function and data are bound together `f1 = f x` implying both should be present on the worker processor when `map` is evaluated.

The extensions to `map` such as `map2` and `mapll` are currently implemented using exactly the same parallel code since multiple argument lists can be `zip`ed up into a list of tuples and nested lists can be `flatten`ed into a single indexed list, see Appendix A for definitions. There are opportunities, however, for the additional structure in these functions to be exploited in the parallel code, for instance `mapll` could be implemented as a farm of farms on a 2D grid of processors. The central Hough transform implementation, mapping the `transform2` function over the list of Hough portion definitions is illustrated in Occam2 pseudo-code in Figure 3.23. This also illustrates partial application and preloading of data.

```
PAR

  -- Farmer processor
  SEQ
    ... Broadcast program parameters and images to workers
    ... Split Hough space up into rectangular portions
    PAR
      SEQ c = 0 FOR ColPortions
        SEQ r = 0 FOR RowPortions
          ... Send work request for Hough portion [c][r]
      WHILE PortionsProcessed < NumberHoughPortions
        SEQ
          ... Read message back from workers
          IF
            ... Detected line returned: add to global line list
            ... Portion processed: increment PortionsProcessed

  -- Worker processor
  SEQ
    ... Read message from farmer processor
    IF
      ... Program parameters and images: update local copy
      ... Work request: return lines then portion processed message
```

Figure 3.23: Occam2: Processor Farm Implementation

3.6.3 Validity of the Conversion Process

It has been shown by example that SML can be converted into Occam2 in such a way that the overall characteristics of the prototype are retained in the implementation. Note, however, that this process is not perfect and that, although the previous examples flatter the conversion process, the original SML and the resulting Occam2 do have significant differences, for instance modelling lists with arrays presents some problems due to the different time complexities of some of the basic operations.

An exact correspondence between the two is not necessary, however. Provided the major components perform the same calculations in the same order there is considerable latitude in the details of the computation. In fact, during hand conversion of code it is difficult to resist performing the obvious optimisations that become apparent, for instance folding multiply nested function calls into a single iteration loop. This is perfectly acceptable and, indeed, desirable since in the context of an automated parallelising compiler it is possible that an optimiser could be applied to the output code. There would be, however, an attendant loss of accuracy in the performance prediction.

3.7 Summary and Conclusions

Selected areas of work from combinations of the basic components (functional languages, computer vision, compilers and parallel processing) were mentioned, and the relationship between them was discussed. This was expanded for the case of functional programming for vision systems which has not previously been studied in great detail. A partial classification of vision algorithms was presented along with working SML examples, the idea being to present some of the technical difficulties in developing vision applications using functional languages and to show the benefit of functional languages in this context. The classification is far from complete but a substantial number of algorithms are represented. In particular, the problems of working with iconic images were illustrated and solutions in terms of list-based and array-based representations were shown.

The relationship between SML and Occam2 as source and target language constructs was explored by showing how the characteristics of the prototype could be retained in the implementation by giving an example of the Hough transform for straight lines. During the development of the applications for this thesis no major problems were encountered in directly converting the SML into reasonably efficient Occam2, although how this process could be automated is another question. In a parallelising compiler it would be inadvisable to have such widely different source and target semantics.

Chapter 4

Programming Methodology

4.1 Introduction

This chapter introduces a methodology of developing efficient parallel algorithms by prototyping in Standard ML, followed by implementing specific higher order functions as parallel skeletons in the target architecture. This process is intended to be guided by performance predictions based solely upon the characteristics of the prototype and a method of characterising the performance of the parallel skeletons is needed.

A very short review of performance modelling is presented in Section 4.2. Following this, a methodology is described in Section 4.3 and then two case studies are presented. The first is Gaussian filtering, Section 4.4, a simple, regular algorithm in the local window operation class of algorithms, Section 3.4.4. The second is edge tracking, Section 4.5, a more complex irregular algorithm, this has already been presented in Section 3.4.6.

4.2 Performance Models

To be able to link functional programs to parallel equivalents, some means of predicting the properties of the parallel implementation from the functional prototype is required. There are several approaches to this which depend on the various ways of characterising the performance of sequential programs.

Computational complexity analysis gives an indication of the behaviour of a program with respect to its inputs, in particular the asymptotic behaviour of space or time resource usage with respect to large input values. One problem with using complexity theory in predicting parallel behaviour is that it is difficult to automate the deduction of computational complexity for arbitrary code, which is undecidable. ACE [264] is an automatic complexity evaluator for

Backus' FP and can derive the time complexity of a functional program but is not guaranteed to work in all cases due to the undecidability problem. The method relies upon a library of recursive definitions and the probability of success depends on the extent of this library. Complexity analysis has been used to characterise the performance of algorithmic skeletons [10] but this approach is not common.

Microanalysis [265] involves characterising the performance of programs as a function of the performance of elementary operations such as addition, assignment and loop/recursion overheads. Again the problem is automating the generation of these functions, this involves solving difference equations which cannot always be solved analytically.

Both complexity analysis and microanalysis are static methods. These have problems with conditionals and other forms of conditional execution such as pattern matching where the performance of the program depends upon the input data. In this case profiling of program operation on representative data sets allows either counting or statistical analysis of alternative sections of code. There is still a problem in deciding upon what constitutes a suitable set of data and upon how extensible the characterisation is to other data sets.

Profiling can also be divided up into *sampling* and *counting* methods. Sampling is where the program is interrupted at regular intervals and a picture of where processing is concentrated can be built up, this has been used for Standard ML [266]. Counting is where passage through specific points in the program are recorded, examples of this are the approaches used in the PUFF compiler [57] and the SkelML compiler [40].

With all the above methods there is still the problem of relating the performance of the sequential code to the performance of the parallel version. This often requires the use of analytical methods to relate properties of the source program, such as data sizes, elemental function times and numbers of function calls to a specific architectural situation using extra parameterisation relating to the hardware such as communications costs and latencies. This approach has been chosen for skeletons work by Rangaswami [105] and To [42] who both develop performance models for functional languages targetted at distributed memory MIMD systems.

A common approach is to guess likely parameters for the process, such as the time to apply a function to a set of data (t_f), the time for the worker to acquire (t_k) and pass on (t_p) a given data element, the number of data elements (N) and the number of processors (P). Generally, the analysis proceeds in worker time space with purely empirical treatment of parameters such as startup overheads and the various network latencies. Bratvold [40] uses a linear pipeline of processors and assumes that there is a linear relationship between a processor's position in the pipeline and the number of data elements computed by that worker. Predictions are typically within 10-15% of the actual measured values.

Purely theoretical modelling has limitations when some of the parameters are difficult to predict, for instance the relationship between a processor's position in an arbitrary topology and the number of data elements computed is non-trivial, even for simple arrangements such as 2D arrays. In this thesis a much simpler method is presented which does not make any assumptions about the physical nature of the system being modelled but is not very general in that it is tied to one particular implementation and is also only likely to be well-behaved for parameters within a small range of values. The idea is to maintain *live* models of parallel constructs, executing on the target parallel architecture and parameterised according to the data provided by the prototype measurements. Thus, for instance, a processor farm is implemented with delays for processing times and artificially generated input and output data sets of a specific size and distribution are used to model the implementation of a `map` HOF.

4.3 Parallel Programs Through Performance Prediction

The core of the methodology revolves around the ability to predict the performance of the parallel implementation based only upon measurements made on the SML prototype and the known characteristics of the existing parallel constructs. The idea is that it is easier to manipulate the prototype than the parallel implementation allowing experimentation with the prototype to optimise the parallel performance of the algorithm. Once the parallel version exists, however, it may be possible to perform minor optimisations on the working parallel code.

The main components of the methodology are:

1. Sequential functional program using HOFs

   The starting point for the methodology is a working SML program that meets the algorithmic requirements of an application. It can be assumed that this prototype was constructed using a set of higher order functions that are known to have efficient parallel equivalents, although it need not be known, at this stage, which will yield useful parallelism. This process has been illustrated for several computer vision algorithms in Chapter 3.

2. Performance measurements on SML prototypes

   Here, the prototype is analysed for communications and computations costs. The exact measurements needed will be those upon which the parallel architecture can be parameterised and can vary from one architecture to another and between different skeletons. As outlined in Section 4.2 these measurements also depend on the nature of the performance models used. For the method proposed here, of maintaining live implementations modelling the parameterised HOF, a fair degree of flexibility in the number and nature of the parameters is achieved. Indeed the parameters can be deduced by relating the performance of the live model to the predictions achieved using the prototype's parameters.

At this point, all the candididate centres of parallelism should be analysed but if there are a large number (a program can contain many `map` instances), it may be possible to prune these using simple heuristics to save effort in making these measurements. For instance any `map`s over empty or singleton lists can be ignored.

3. Performance prediction of parallel implementation

   Once the performance measurements are available these can be plugged into performance models for the parallel constructs to estimate the parallel performance. Predicting the overall performance of a number of skeletal HOFs in the same program is difficult but could be estimated from the percentage of the overall calculation carried out by the HOFs. Any HOFs not giving good speedup can be run sequentially but the threshold for this is arbitrary, in the worst case the single HOF giving the best predicted speedup for the greatest amount of work could be used as a starting point.

4. Optimise predicted parallel performance

   At this stage a parallel implementation is hypothesized. It may then be possible to improve on the performance by re-evaluating the sequential code. For example, communications costs could be lessened by "merging" successive skeletons or data movement reduced by eliminating redundant operations. This process may be possible without having to redo the performance measurements, given a slight loss of accuracy.

5. Instantiate appropriate skeletons for HOFs

   The nominated HOFs which have been identified as giving useful parallelism need to be turned into parallel instances of those HOFs. This should be a fairly automatic process but could involve a small amount of code and data rearrangement to allow the parallel instance to be generated, for instance it is sometimes useful to lift lambda expressions into named functions for easier manipulation in the parallel version.

6. Parallel Occam2 generation

   The sequential code is directly translated from SML into parallel Occam2. Guidelines for this have been presented in Section 3.6.

7. Optimise the parallel implementation

   Once a working parallel program has been written it may be possible to optimise the performance, for instance by precalculating values on remote processors rather than transmitting them through skeletons. This process can be emulated in the sequential version but, using hand-coding, it may be more efficient in programmer time to perform experiments on a working parallel version than to subject the optimisations to the full methodology.

This methodology has been proposed based on previous research [55, 40]. The remainder of this chapter attempts to illustrate and evaluate the proposed methodology using two case studies, a simple straightforward example of Gaussian filtering and a more complex example of edge tracking. The entire process is carried out by hand and there is a great deal of scope for interpretation of the above guidelines but the general approach is shown to work in these examples.

4.4 Case Study: Gaussian Filtering of Intensity Images

4.4.1 Sequential Functional Program Using HOFs

A simple example of the use of a `map` higher order function is Gaussian filtering. This is a convolution operation of the type discussed in Section 3.4.4 where the filter is a simple Gaussian filter and the window operation is convolution. Figure 4.24 shows the generation of the basic filter operation, this would be the starting point for parallel implementation.

Unfortunately, the `rmap/cmap` operations at the core of the `local_window` operation give no control over the granularity of the parallelism, so an extra layer of processing is placed between the filtering operation and the top-level call. This is a decomposition operation whereby the lists are split up into equal-sized segments, the filter operation is applied to each segment and the result is constructed by the inverse of the decomposition. A problem with the convolution operation is that to apply the filter to a segment of a list requires data from outside the list. A related problem is that default data has to be supplied from outside the image altogether, this could be generated by extrapolation or more simply by defining a default value. Figure 4.25 shows one possible scheme with the required properties, the most essential of which is the property:

Lemma 4.1 *The following identity holds for* `calcgeomdecomp` *and* `flattengeomdecomp`*:*

```
(* Gaussian filter window generation *)
fun gaussConst s = 1.0 / (12.0 * pi * s * s)
fun gaussExpon s x y = exp(~((x * x) + (y * y)) / (2.0 * s * s))
fun genGauss s xn yn x y =
  let
    val c = gaussConst s
    val x = ((real x) + 0.5) - (xn / 2.0)
    val y = ((real y) + 0.5) - (yn / 2.0)
  in
    c * gaussExpon s x y
  end
fun gaussFilt s xn yn = LLists.fxylist 0 xn 0 yn (genGauss s (real xn) (real yn))

(* Define Gaussian filter *)
fun gaussoper s xn yn = convol (gaussFilt s xn yn)

fun gauss s xn yn img =
  LLists.shiftll 0.0 (xn div 2) (yn div 2) (local_window (gaussoper s xn yn) img)
```

The `gaussFilt` function generates a Gaussian filter with σ=`s` and window dimensions `xn` by `yn`. The `genGauss` function evaluates the filter at `(x,y)` relative to the centre of the filter window. The filter operation `gauss` is generated by applying the `local_window` function to the convolution operator instantiated with the Gaussian filter operation, this is based on code in Figure 3.7. There is a side-effect of `local_window` in that the image is shifted by half the filter window size which is negated by the `shiftll` function.

Figure 4.24: Generation of a Gaussian Filter

$$(\texttt{flattengeomdecomp}\; dc\; dr\; N) \circ (\texttt{calcgeomdecomp}\; dc\; dr\; N) \;=\; I \tag{4.30}$$

where `I` is the identity function.

Example:

```
- ((flattengeomdecomp 2 2 3) o (calcgeomdecomp 2 2 3))
  [[1,2,3],[2,3,4],[3,4,5]];
val it = [[1,2,3],[2,3,4],[3,4,5]] : int list list
```

Proof sketch: A simple proof of this would revolve around considering an arbitrary element `(x,y)` which is at `(x+(N/2),y+(N/2)) == (x',y')` in the padded image and ends up in subimage with index `((s1,e1),(s2,e2))` where

$$\texttt{s1} \leq \texttt{x}' < \texttt{s1} + \texttt{e1} \quad \text{and} \quad \texttt{s2} \leq \texttt{y}' < \texttt{s2} + \texttt{e2}$$

Since the order of this subimage within the image remains unaltered (properties of `mapll`), if $\texttt{x}' < \texttt{x}''$ in the starting image then it is also true in the output image. The final proof would be induction on the size of the subimages. □

Furthermore, if a function `ff` preserves the order of elements in a subimage then `ff` is promotable through Equation 4.30 by the above proof outline, giving rise to the required parallel decomposition in Figure 4.26. This is the basis for the parallel implementation.

```
(* Perform geometric decomposition on list list image *)
fun calcgeomdecomp xprtns yprtns N img =
  let
    val over = N div 2
    fun sh' 0 _ _ _ _ = nil
      | sh' cnt m delta over offset =
        ((m * delta) - over + offset,((m + 1) * delta) + over + offset)
          ::(sh' (cnt - 1) (m + 1) delta over offset)
    fun getseg img ((s1,e1),(s2,e2)) =
      (((s1,e1),(s2,e2)),LLists.segll (s1,e1 - s1) (s2,e2 - s2) img)
    val (xsz,ysz) = (length (nth(img,0)),length img)
    val (deltax,deltay) = (xsz div xprtns,ysz div yprtns)
    val gd = LLists.tabulatell (sh' (xsz div deltax) 0 deltax over over)
                               (sh' (ysz div deltay) 0 deltay over over)
  in
    LLists.mapll (getseg (LLists.padll 0 over img)) gd
  end

(* Flatten geometric decomposition *)
fun flattengeomdecomp _ _ N gd =
  let
    val over = N div 2
    fun inner ll =
      if (hd ll) = nil
        then []
        else (Lists.flatten (map hd ll))::(inner (map tl ll))
    fun outer gd =
      if (hd gd) = nil
        then []
        else (inner (map hd gd))@(outer (map tl gd))
  in
    outer (LLists.mapll (LLists.unpadll over) (LLists.mapll Gen.t2f2 gd))
  end
```

The `calcgeomdecomp` function operates by generating a list list of integer boundary definitions `((s1,e1),(s2,e2))`, including a boundary of size `over`, which are then used to extract the relevant subimage, this is less efficient than direct extraction but results in more transparent code. The input image is padded to give data from outside the image and the boundary definitions are offset to get the correct indices into the original image. `flattengeomdecomp` is the reverse of this process, the boundaries are removed followed by successive flattening in each dimension.

Figure 4.25: Geometric Decomposition for `list list` Data

```
fun geomdecomp (dc,dr,N) ffsubimage img =
  let
    val decomp = calcgeomdecomp dc dr N img
    val res = LLists.mapll ffsubimage decomp
  in
    flattengeomdecomp dc dr N res
  end

fun gdgauss (dc,dr,s,N) img =
  let
    fun ffgauss s N (a,subimg) = (a,gauss s N N subimg)
  in
    geomdecomp (dc,dr,N) (ffgauss s N) img
  end
```

A general purpose geometric decomposition skeleton is defined (`geomdecomp`) which applies the principles in Equation 4.30. The `img` parameter is broken up into geometric regions (subimages) by `calcgeomdecomp` and the parallel implementation is drawn from the `mapll` function using a parallel method not specified here, see Section 3.3.1. The processed regions are reconstituted into the output image by `flattengeomdecomp`. The function `gdgauss` creates a suitable instantiation function from the `gauss` function (Figure 4.24) and instantiates the `geomdecomp` skeleton with this function (a square filter window is assumed by setting `xn = yn = N`). The granularity of the decomposition is set by the `dc` and `dr` parameters which define how many subdivisions are required along rows and columns, the total number of subimages is `dc` × `dr`. The slightly open syntax is to make room from the profiler equivalent functions (Figure 4.27).

Figure 4.26: Gaussian Filtering Using Geometric Decomposition

The process of developing this example is perhaps more complex and involved than might be expected. In particular, the programmer requires an awareness of the parallel implementation. This is only in the most abstract sense, however, once the basic properties of the skeleton component (`geomdecomp`) have been characterised it can be reused for future development taking some of the effort out of this process. The technique of placing identity operations within sequential code to allow extraction of parallelism is well-known [42] and again, a library of such definitions could be hypothesized and used in the manner of a coordination language [164].

4.4.2 Performance Measurements on SML Prototypes

In the design of a new skeleton the first activity is to decide upon a suitable parameterisation and to arrange the measurement of those parameters that are dependent upon the sequential instantiation of the skeleton. This process is illustrated by developing as an example the implementation of a `map` HOF which applies a function (`ff`) to each member of a list implemented in parallel as a processor farm. The performance model is based on the size of the inputs and outputs of the application function `ff` and the average time required to execute these functions.

Due to the relatively coarse parameterisation of this construct a detailed profiling of the source

```
fun map_timed timecnt ff ffisize ffosize nil = ([],[],[])
  | map_timed timecnt ff ffisize ffosize (h::t) =
  let
    val (ffval,(ut,_,_,_)) = timen timecnt ff h
    val (fres,timelst,sizelst) = map_timed timecnt ff ffisize ffosize t
  in
    (ffval::fres,ut::timelst,(ffisize h,ffosize ffval)::sizelst)
  end

fun mapll_timed timecnt ff ffisize ffosize nil = ([],[],[])
  | mapll_timed timecnt ff ffisize ffosize (h::t) =
    let
      val (ffres',times',sizelst') = map_timed timecnt ff ffisize ffosize h
      val (fres,timelst,sizelst) = mapll_timed timecnt ff ffisize ffosize t
    in
      (ffres'::fres,times'::timelst,sizelst'::sizelst)
    end
```

For `map_timed` the function `ff` is applied `timecnt` times to each element `h` of the input list and the average user time `ut` is calculated, this process is carried out by the `timen` function. The `ffisize` function is a caller supplied function which calculates the input size of an element, the output size is calculated by `ffosize`. The result is a tuple comprising the actual result of the `map` application along with lists of times and sizes of the inputs and outputs which can be further processed to give average, minimum and maximum values. The `mapll_timed` function is simply a composition of `map_timed` functions. This is only needed when the output lists are required to retain their nested structure.

Figure 4.27: Timed `map` and `mapll` Functions

code is not required. This allows the use of the inbuilt profiler for New Jersey Standard ML to time function applications, although due to variations in computational overheads these need to be averaged over several identical runs. Thus the program is annotated by replacing the instances of the HOF that is to be parallelised with equivalent functions which collect the required information. The data sizes can be calculated by providing an auxiliary function which calculates data size from the sizes of atomic entities plus the sizes of constructed data such as lists. This method is not completely clean in a syntactic sense since the output value from the application of the HOF has to be extracted from the measurements. Figure 4.27 shows the timed equivalent for the `map` HOF.

Table 4.10 shows the average SML processing times and input and output sizes (which are the same in this instance) for the `gdgauss` function in Figure 4.26. These data are summarised in Figure 4.28. Various divisions of the test image are given from 2×2 up to 9×9 and for three sizes of convolution window size. Several values for the σ value of the Gaussian filter are also included. Most of the trends in this table are as expected; as the number of divisions increases, the transmission size per subimage decreases and the processing time decreases in proportion to this. The standard deviations are all very small indicating a highly regular computation and suggesting that the deviations should not be included in the independent variables in the performance model. The variation with window size is more significant but

Average times for `gdgauss`								
Decomp.	N	Gaussian $\sigma = 0.1$		Gaussian $\sigma = 1.0$		Gaussian $\sigma = 10.0$		Size (bytes)
		Time (s)	Time (σ)	Time (s)	Time (σ)	Time (s)	Time (σ)	
2×2	3	0.496	0.0204	0.502	0.0185	0.502	0.013	12560
3×3	3	0.209	0.00874	0.213	0.0105	0.214	0.00524	5792
4×4	3	0.123	0.00472	0.116	0.00436	0.118	0.0052	3380
5×5	3	0.0691	0.00387	0.0681	0.00276	0.0689	0.00388	2132
6×6	3	0.0496	0.00306	0.0489	0.00244	0.0501	0.00331	1616
7×7	3	0.0352	0.00317	0.0337	0.00231	0.0352	0.00315	1172
8×8	3	0.0265	0.00277	0.0253	0.00185	0.0262	0.00182	916
9×9	3	0.0256	0.00275	0.021	0.00171	0.0224	0.00246	800
2×2	5	0.958	0.0794	0.938	0.0325	0.956	0.0627	13472
3×3	5	0.386	0.0186	0.356	0.00827	0.37	0.0164	6416
4×4	5	0.233	0.00547	0.219	0.017	0.229	0.0107	3860
5×5	5	0.143	0.00452	0.143	0.00437	0.14	0.00458	2516
6×6	5	0.107	0.00488	0.106	0.00426	0.103	0.00408	1952
7×7	5	0.0695	0.00622	0.0725	0.00521	0.0726	0.00495	1460
8×8	5	0.052	0.00416	0.0543	0.00504	0.0512	0.00427	1172
9×9	5	0.0509	0.00309	0.0442	0.00324	0.0499	0.00385	1040
2×2	7	1.58	0.00685	1.63	0.0808	1.67	0.0365	14416
3×3	7	0.659	0.0345	0.7	0.0213	0.659	0.0576	7072
4×4	7	0.376	0.0295	0.382	0.028	0.381	0.0308	4372
5×5	7	0.249	0.00651	0.256	0.0126	0.25	0.0084	2932
6×6	7	0.189	0.00994	0.197	0.00801	0.182	0.0148	2320
7×7	7	0.13	0.00976	0.131	0.00715	0.135	0.0104	1780
8×8	7	0.111	0.00541	0.115	0.00687	0.118	0.00851	1460
9×9	7	0.0942	0.00748	0.0962	0.00834	0.0943	0.01	1312

Decomp is the number of divisions in each direction so 5×5 corresponds to 25 subimages. N is the size of the Gaussian filter element. The Time values are the average user times in seconds per subimage, measured over 10 identical runs, the standard deviations of the Time values (σ) are also given. The Size column is both the input (argument) and output (return value) size in bytes of the subimage processing function and corresponds directly to parallel communications overheads.

Table 4.10: SML Timing Summary for the Gaussian Filter Example

can be catered for in the performance model, the actual window size can be plugged into the prototype to generate the parameters that do affect the parallel performance, for instance the transmission sizes and processing times.

4.4.3 Evaluation of Performance Model

The next activity is to predict the performance of the parallel implementation from the measurements made on the prototype. To do this the performance of the basic parallel construct (in this case processor farming) has to be characterised over a useful range of its parameters. For a detailed investigation into the performance modelling of farms and pipelines see [186]. Details of simpler models used to predict processor requirements in the context of an automatic parallelising compiler are given in Bratvold [40]. A performance

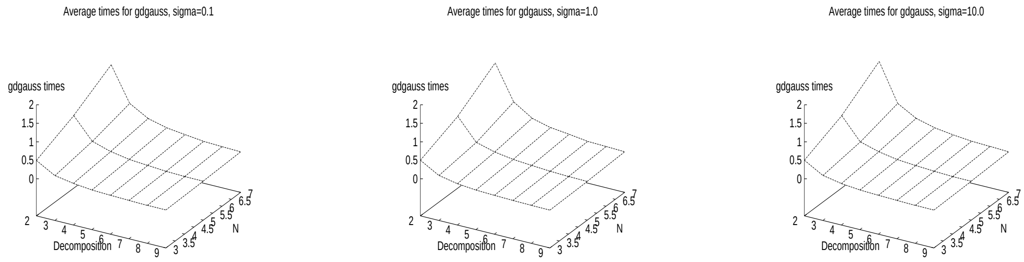


Figure 4.28: SML Timing Summary Graph for Gaussian Filter Example

model for processor farms can also be found in Darlington [160].

As already pointed out, a much simpler method is proposed here, the idea of maintaining *live* models of the parallel constructs, executing on the target parallel architecture and parameterised according to the data provided by the prototype measurements. Unfortunately, the prototype compiler (SML) does not have access to the parallel hardware (a Meiko transputer surface). To overcome this, a "dummy" processor farm is implemented and tested over a suitable range of values for the parameters. The performance is then characterised by numerical analysis techniques, which leads to a polynomial fit which can be used in developing the SML prototype.

A dummy `map` processor farm has been implemented with controllable parameters; number of processors (P), number of elements (N), average processing time per element ($t_f\mu$) and standard deviation of the t_f values, ($t_f\sigma$), output data size ($Osize$) and return value size ($Rsize$). The performance of this farm was measured for a range of values of these parameters around typical values from the prototype.

A general polynomial function was then fitted to this data using a least squares method and used to predict the performance of the implementation from the prototype parameters. Figure 4.29 shows the accuracy of fit for speedup using a first order fit and typical values measured from the prototype ($Osize = Rsize = 2000$ and $t_f\mu = 0.06$) over a useful range of processors ($P = 8\, to\, 26$) and subimages ($N = P\, to\, 6 * P$). Figure 4.30 shows the accuracy of fit for a second-order fit. Note that the errors are based upon the residuals and that the data-points are at the intersections of the lines in these figures. The *portions per processor* axis is the value N/P.

There are several points about this modelling:

- If N is not a multiple of P and there are relatively few portions compared to the number of processors then the odd portions drastically affect the performance (consider 3 portions processed by 2 processors). This has specifically been avoided by forcing N to be a multiple of P, although other factors can cause similar imbalances in processor

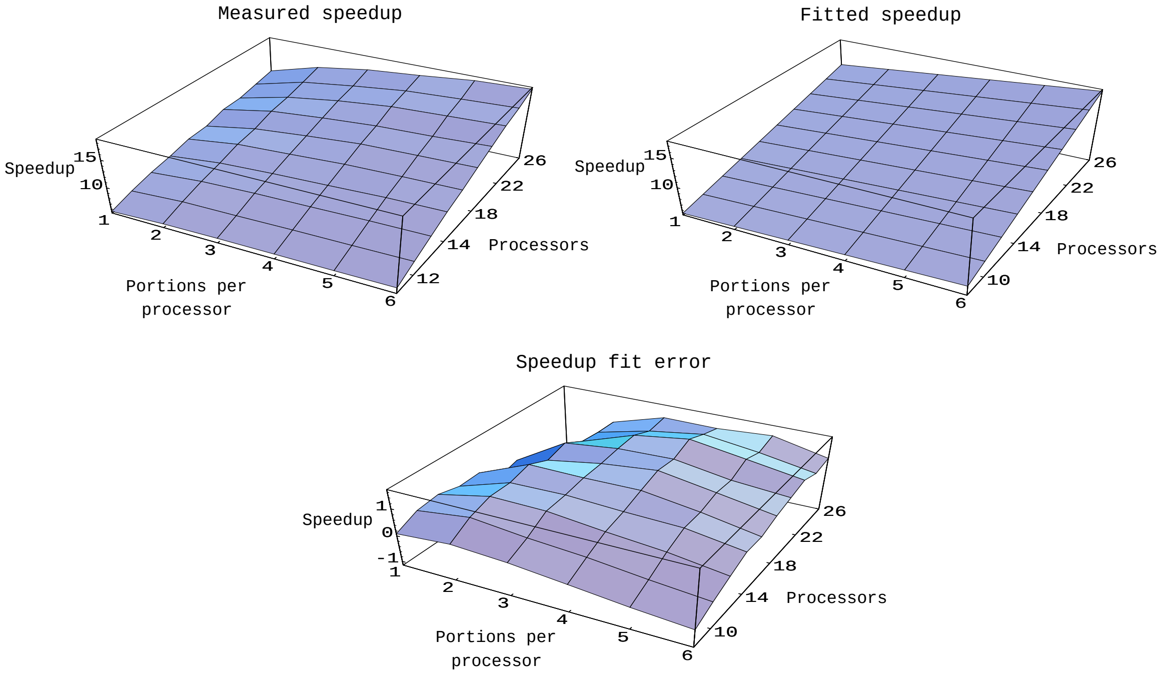


Figure 4.29: Accuracy of Fit for Numerical Performance Model, First-order Fit

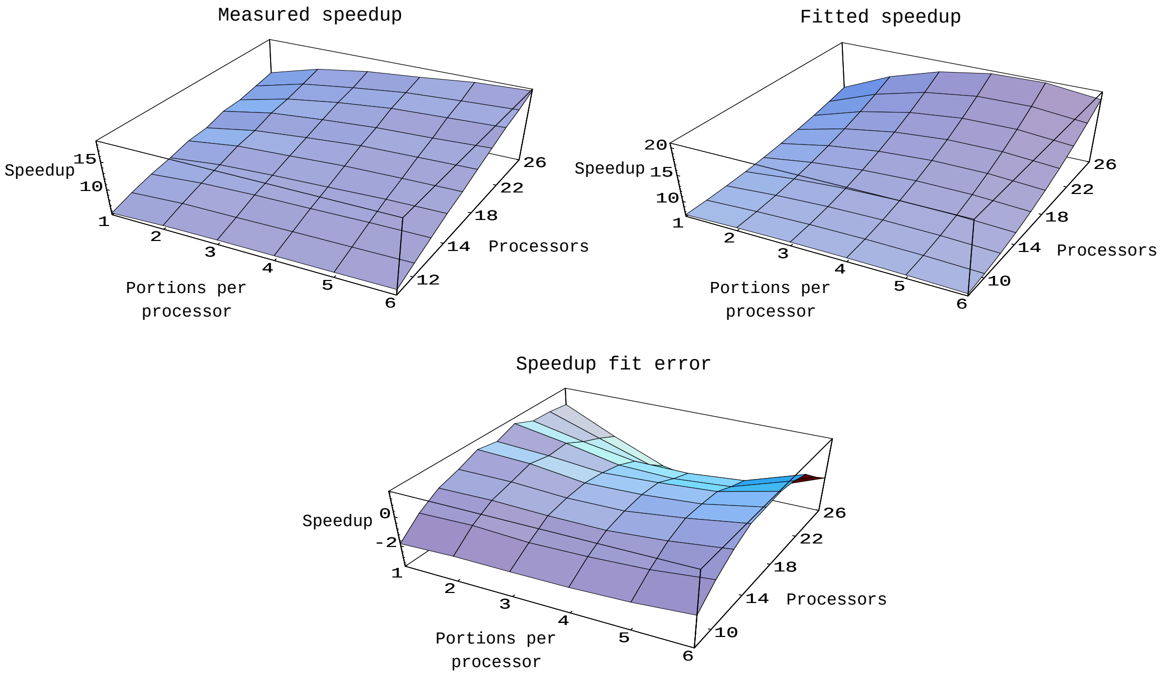


Figure 4.30: Accuracy of Fit for Numerical Performance Model, Second-order Fit

loadings, for instance if the variances in the t_f values were to become large a similar problem would arise.

- Although this looks like a reasonably good fit it should be noted that the data sample upon which the fit was based was chosen to avoid problems with coarse granularity (for instance small numbers of processors) and regions where other factors come into effect (for instance communications degradation for large numbers of portions).

- The values in Figures 4.29 and 4.30 are also very centralised, systematic deviation and outliers are more common near the boundaries of the subspace spanned by the data set.

The main utility of using function fitting is that it provides a simple means of predicting performance from within a compiler or manually using data generated by profiling. The two polynomial fits for the processor farming example can be expressed as a simple function, Figure 4.31.

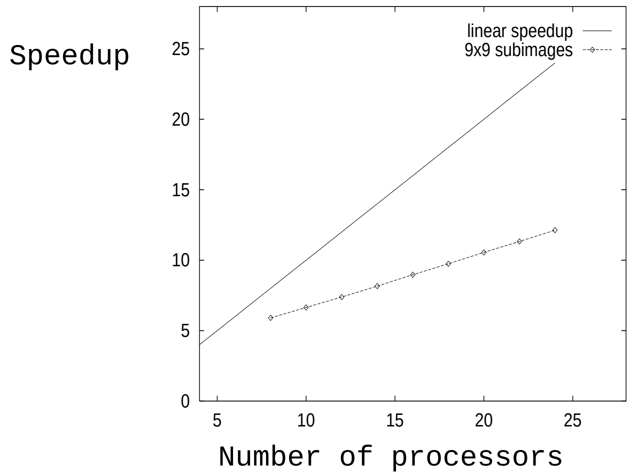
```
(* Fit from fitting routines *)
val test = (8.0,16.0,2000.0,0.06)

(* Linear fit, returns predicted speedup *)
fun lin (P,N,Osize,Tf) =
     1.14752197266
  +  0.40329182148    * P
  +  0.0323047377169  * N
  + ~0.00143308565021 * Osize
  + 61.0439300537     * Tf

(* Test prediction *)
- lin test;
val it = 5.6871968507724 : real
```

The linear fit function, parameterised according to number of processors P, number of subimages N, transmission size $Osize$ and time to apply the Gaussian filter to the subimage Tf. The measured value from the dummy farm is 5.92.

Figure 4.31: Fit Function for Linear Fit to Processor Farm Data

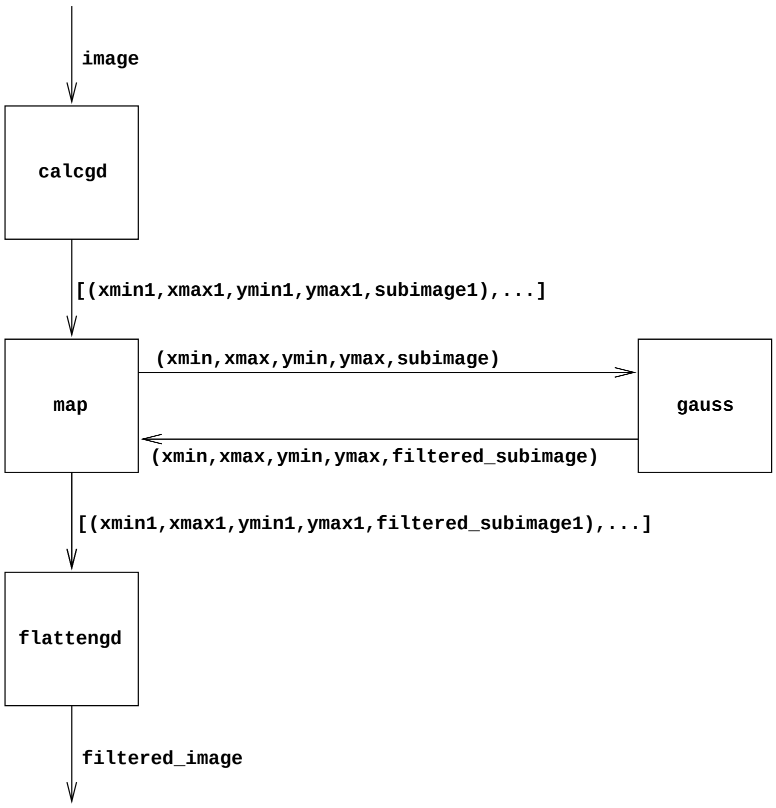
4.4.4 Instantiate Appropriate Skeletons for HOFs

The Occam2 implementation is a straight encoding of the SML code, implementing top-level sequential code on the farmer processor, the `calcgeomdecomp` and `flattengeomdecomp` operations in this case. The parallel component, the `map` over decomposed subimages is implemented as a processor farm. The structure of the Occam2 is illustrated in Figure 4.33. In this simple example there are not really any choices in terms of implementing skeletons in parallel, these choices were made implicitly at the prototype stage. In more complex examples



The prediction is for 9×9 divisions of the image, the Gaussian σ is 1.0 and N the window size is 5. The prediction is from the linear fit shown in Equation 4.31.

Figure 4.32: Performance Prediction for Gaussian Filter Example

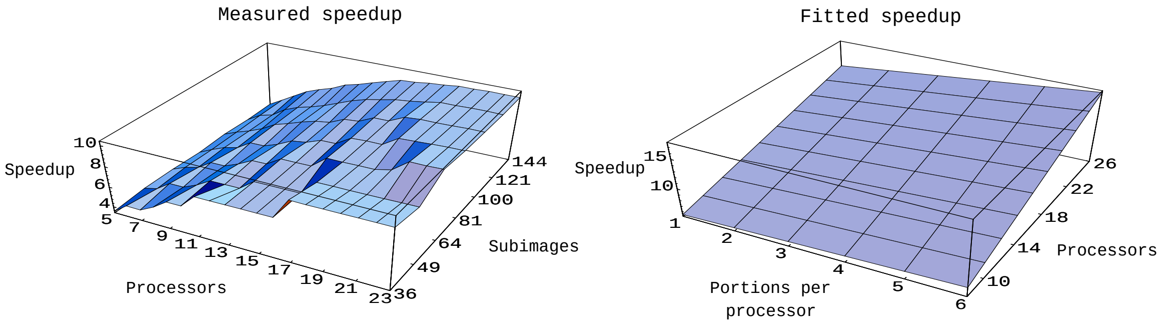


The input image is split up by `calcgeomdecomp` (`calgd`), the subimages are farmed out to the workers which implement the complete top level Gaussian filter (`gauss`) and the resulting filtered subimages are reconstructed using `flattengeomdecomp` (`flattengd`).

Figure 4.33: Structure of the Gaussian Filter Parallel Implementation

there will be considerable latitude for deciding which skeletons in the prototype are actually implemented in parallel and which are implemented sequentially, this choice would be guided by the prototype performance predictions.

4.4.5 Parallel Occam2



The speedup measured from the Occam2 implementation based on the system illustrated in Figure 4.33. The filter window size is 5 and the number of processors P ranges from 5 to 23. The number of subimages N ranges from 36 to 144 (not on a linear scale). These values all bracket the values measured from the SML prototype.

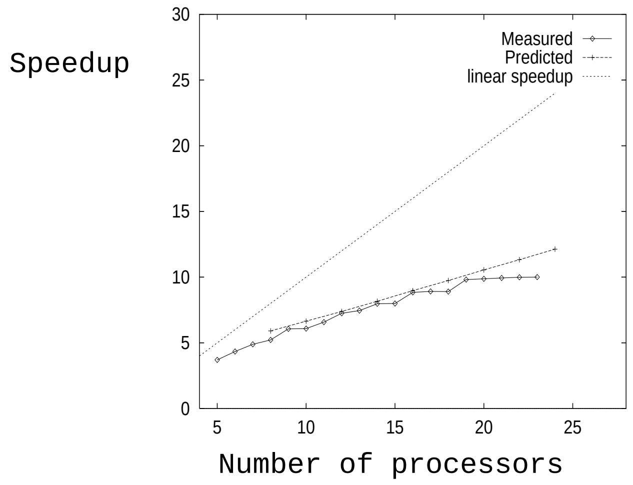
Figure 4.34: Performance of the Gaussian Filtering Example

The parallel algorithm implemented in Figure 4.33 gives the speedup characteristics in Figure 4.34. Note that the number of subimages does not affect the overall performance in this region of operation, the only effect is that when the number of subimages is small the residual computation causes a degeneration in performance (when the number of subimages is not a multiple of the number of processors).

Figure 4.35 gives a comparison of the measured speedup from the Occam2 implementation with the prediction from the SML prototype using the linear polynomial fit in Figure 4.29. There is reasonable agreement for the valid range of the model in terms of the general trend and absolute values. Another point illustrated by Figure 4.35 is that the effect of residual computation can be seen (where N is not a multiple of P). The prediction is effectively an upper bound on performance when an even distribution of subimages to workers is possible.

4.4.6 Possible Improvements to the Algorithm

This particular arrangement is suboptimal since image data is duplicated among the work packets sent out to the workers (the overlap between subimages required by the filtering routine). In general with operations on iconic images preloading of image data is often more



The measured speedup for the Occam2 implementation is compared with the prediction from the SML prototype for 81 subimages = 9×9 divisions.

Figure 4.35: Comparison of Predicted and Measured Performance for Gaussian Filter

efficient than sending image data during the computation. This requires a preload phase but this can use an optimised broadcast scheme rather than point to point transmissions.

This highlights a potential flaw in the methodology since the implementation was guided directly by the prototype. In some instances it may be possible to optimise data allocation independently of the prototype by performing semantics preserving transformations upon the parallel code but in this simple example where the implementation is a direct translation of the prototype it is possible for inefficiencies in the prototype to be carried through into the implementation.

4.5 Case Study: Edge Tracking of Edge Enhanced Images

4.5.1 Sequential Functional Program Using HOFs

The problems associated with developing a suitable prototype edge tracking program have already been discussed in Section 3.4.6. Neither of the imperative methods, using lists (Figure 3.11) or arrays (Figure 3.12) are really suitable for direct implementation from the prototype. This is because they require access to the data in a manner that is difficult to express in native SML, bearing in mind that it would be desirable to keep as close a correspondence between the prototype and implementation as possible.

The third edge tracking system (Figure 3.14) also has problems. Firstly it has a higher computational complexity than the imperative system so direct, fine-grained implementation of this algorithm is unlikely to lead to respectable parallel performance compared with a sequential implementation of the imperative method. Also, it is expressed as nested, non-associative `fold` functions which cannot be implemented in parallel.

The problem with the non-associativity of the `genlist` function can be overcome by spotting that the only difference between the left and right folded outputs is the order of the points in the lines and the order of the lines in the lists of lines:

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

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

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

This implies that an edge tracking system could be implemented using an associative fold, however, the property of the non-associative fold system that ensures all points are compared with all other points has to be maintained. It is difficult to see how this could be done while decomposing the associative fold for parallel execution. An obvious implementation of this, however is to use the comparison operator to `filter` a list of all point pairs and implement the merges as a sequential operation. This method is illustrated in Figure 4.36.

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

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

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

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

Figure 4.36: Filtering a List of Line Pairs with the Compatibility Operator

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

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

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

Here, an edge tracking algorithm is expressed as a general purpose geometric decomposition algorithmic skeleton. The `etportion` function is the code in Figure 3.12 adapted to work within a geometric boundary. A list of boundaries is generated by `calcgeomdecomp` by splitting the image into `dc` column segments and `dr` row segments. The `etcombine` function is based on the code in Figure 3.14 modified to use the code in Figure 4.36 to calculate the merge requirements followed by a `fold` operation to implement the merges.

Figure 4.37: A General Purpose Geometric Decomposition Skeleton

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

Although this solution has been arrived at with a fair degree of commonsense reasoning which would be difficult to automate, the prior experience in developing several different edge tracking systems has been used to provide a system which fits the requirements, Figure

4.37. This experience has also been packaged up in the form of the GD and DF [1] skeletons which could be reused in other vision applications. For instance, an identical method has been used to implement planar region growing from depth data, Austin [65].

4.5.2 Performance Measurements on SML Prototypes

At this point there is a working SML program that meets the algorithmic requirements. The prototype was developed using the algorithmic skeleton constructs described in Section 5.3 and the objective now, is to investigate the viability of the parallel implementation by making measurements on the behaviour of the working SML code. The edge tracking prototype was subjected to the methodology described in Section 4.2.

The complete SML prototype is based on the principle illustrated in Figure 4.37. This comprises a `map` over a geometric decomposition (GD), returning partial line segments followed by a `filter` over all pairs of line segments (which can be implemented using data farming (DF)) feeding into a non-associative `fold`. This identifies two potential centres of parallelism, the `map` over the geometric decomposition and the `filter` over all pairs of partial lines. These HOFs are replaced by calls to equivalent functions which accumulate data sizes and function execution times. Figure 4.38 shows the equivalent function for the `filter` skeleton, the equivalent function for the `map` construct is shown in Figure 4.27.

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

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

Figure 4.38: Timed `filter` Function

Table 4.11 shows the average processing times and input and output sizes for the first parallel construct, the `map ffportion` component in `geomdecomp` for various divisions of the test image. The fixed size of the inputs to the function is due to the fact that only the geometric

[1] Note that this application was developed before the four skeletons presented in Section 5.3 had been finalised so the discussion proceeds in terms of HOF names (`map` and `filter`) rather than the skeleton names. The concepts discussed are the same, however.

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

This table illustrates the relationship between the granularity of the decomposition and the processing time per element and the communications overheads. As the subimages get smaller, the processing time decreases in proportion and the output message size also decreases. The overall communications load increases, however, due to the increased overhead per message for fine granularity.

Table 4.11: SML Timing Summary

boundaries are supplied as inputs. The major parameter in the parallel implementation is the granularity of the decomposition, although in this case it cannot be decided in isolation because there is an important data dependency between the two centres of parallelism. The benchmarking and performance modelling is only presented for the first (tracking phase), the overall performance is assumed to be some compromise between the two phases set by the decomposition.

4.5.3 Evaluation of Performance Model

The next activity is to predict the performance of the parallel implementation from the measurements made on the prototype. Again the methods proposed in Section 4.4.3 are used. The basic construct is implemented with controllable parameters and characterised using numerical analysis techniques. The same dummy farm used in Section 4.4.3 can be used here directly but bracketing the approximate parameters measured from the prototype.

In this case, the controllable parameters are; number of processors (P), number of elements (N), average processing time per element ($t_f\mu$) and standard deviation of the t_f values, ($t_f\sigma$). A general cubic function gave the best fit to this data and was used to predict the performance of the implementation from the prototype parameters. Figure 4.39 shows the accuracy of fit for values measured from the dummy farm and fitted values of speedup for $t_f\mu$ and $t_f\sigma$ similar to those determined by the prototype.

To get a prediction from this model it was necessary to take into account the discrepancy in linear processing times between SML and Occam2. One approach to this is to match up constructs between the two languages and use only relative processing times in the prediction. This was adopted in Michaelson and Scaife [61]. Here a different approach is used, equivalent

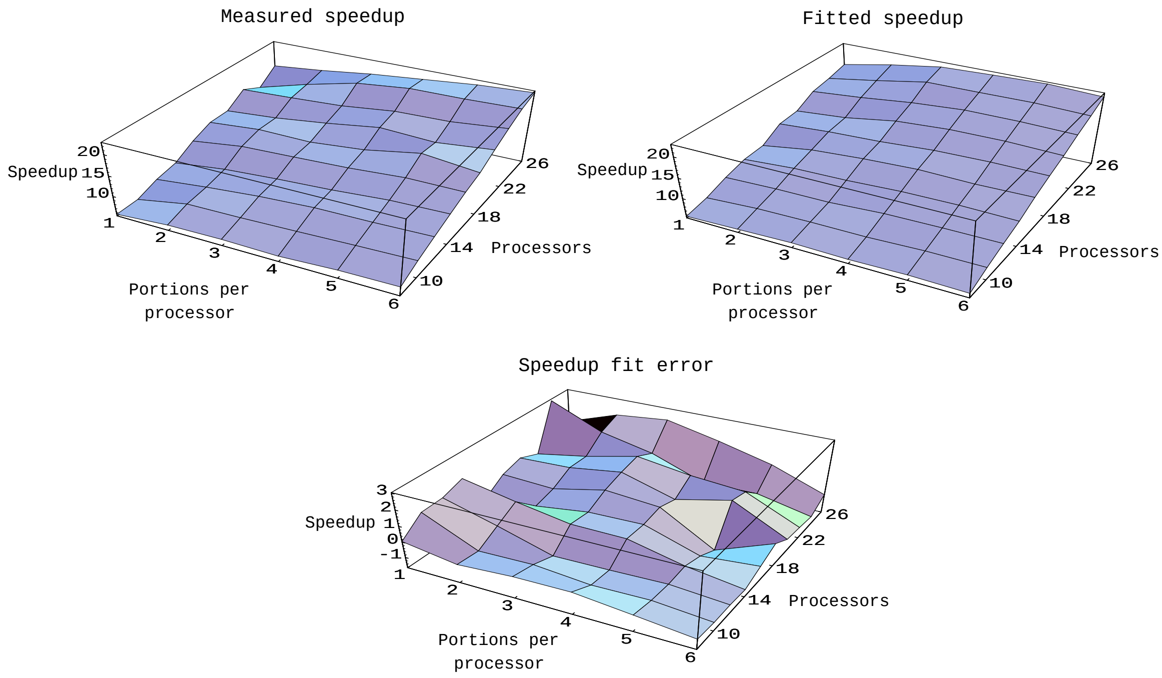


Figure 4.39: Accuracy of Fit for Numerical Performance Model

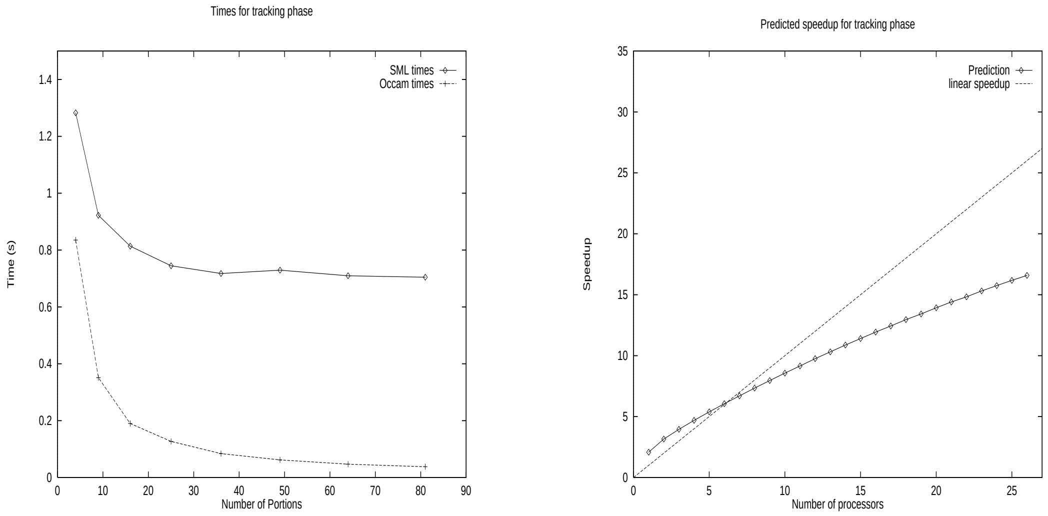
functions are written in SML and sequential Occam2 and timed on the same data sets. This gives a direct comparison between the two and allows a simple correction to be applied, Figure 4.40 a) shows average SML and Occam2 runtimes for the `etportion` function which indicates a fixed overhead in the processing times for the SML function which could be subtracted out.

Taking this correction into account, Figure 4.40 b) shows the accuracy of the fitted model for 25 image portions, $t_f\mu$ and $t_f\sigma$ were taken from the SML prototype but adjusted according to the measured discrepancy between the SML and Occam2 code. The superlinear speedup at low values is attributable to the fact that the model data did not include values for small numbers of processors where quantisation effects were more significant.

4.5.4 Instantiate Appropriate Skeletons for HOFs

The Occam2 implementation is a straight encoding of the SML, implementing top-level sequential code on the farmer processor, `map` and `filter` operators on the worker processors and implementing both `map` and `filter` as processor farms.

The structure of the final version is illustrated in Figure 4.41. This indicates a pipeline of four processes, two of which are sequential (`indx/comb` and `foldl`) and two of which are parallel (`map` and `filter`). These could be implemented as physically separate processors. Work on the prototype, however, showed that there is a great disparity between the amount of



a) SML and Occam times **b) Prediction from model**

Figure 4.40: Correction for Sequential Processing and Performance Prediction

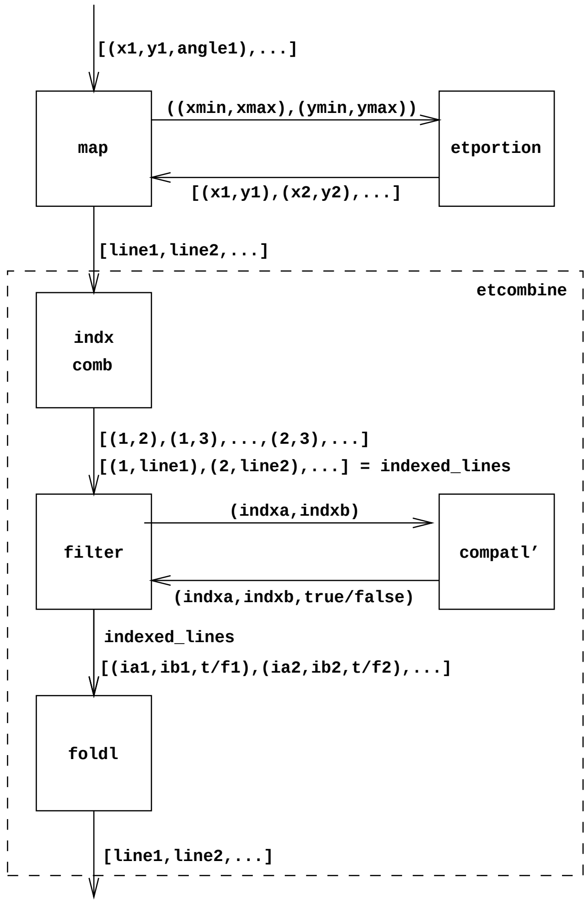


Figure 4.41: Structure of the Parallel Edge Tracking Implementation

computation between the individual elements in the pipeline. This would make the pipeline difficult to balance. For instance, the `indx/comb` node has almost no computation at all. The system is therefore implemented as a single farmer with dual purpose workers processing successively the `etportion` and `compatl'` functions.

4.5.5 Parallel Occam2

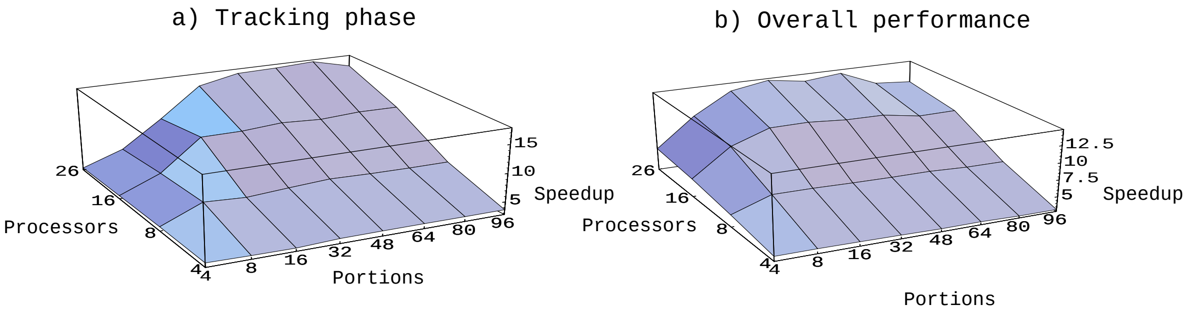


Figure 4.42: Performance of the Occam2 Edge Tracking System

The parallel algorithm is structured exactly as in Figure 4.41. Figure 4.42 a) shows the measured values of speedup for the tracking phase predicted in Figure 4.39.

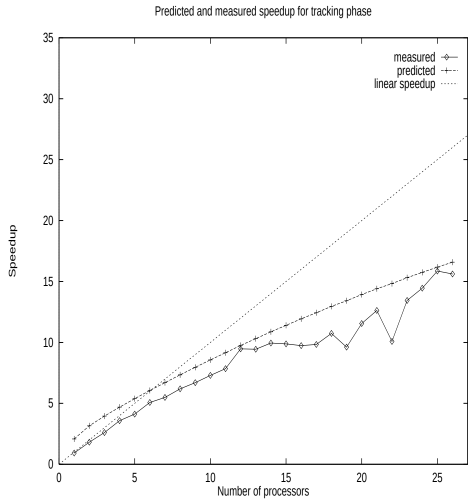


Figure 4.43: Comparison of Predicted and Measured Performance for Tracking Phase

Figure 4.43 gives a comparison of the measured execution times per image portion for the SML prototype and Occam2 implementation. There is reasonable agreement for the valid range of the model in terms of the general trend and absolute values although the effect of non-uniform distributions of data in the image can be seen.

Another point illustrated by Figure 4.43 is that the effect of residual computation can be seen (where N is not a multiple of P). The prediction is effectively an upper bound on performance when there is an even distribution of work among the t_f values, whereas real data will not have such regular distributions as were used in the modelling. Note how the performance of the edge tracker approaches that of the model when N is a multiple of P (about 12 and 25 processors).

Figure 4.42 b) shows the overall performance for the edge tracking system. This is dominated by the performance of the tracking phase for small numbers of portions but as the number of portions increases the amount of data being passed forward from the tracking phase increases and the $O(M^2)$ complexity of the merge phase starts to take effect. It would be difficult to predict exactly where the balance between the two phases lies. For the sparse image used in these experiments (Figure 8.102) and for the numbers of processors and portions considered this balance is only manifested as a very sublinear speedup at higher numbers of processors. More drastic effects are to be expected for denser images or larger numbers of portions. One conclusion from this might be that the predictions are only valid for simple, well-defined constructs such as the tracking phase implemented as a processor farm. If there are complex data dependencies within the algorithm, as in the case for the overall edge tracking system presented here, then the models become more complex and, presumably, less reliable. It may be that building larger systems from discrete modules will always require some "fine-tuning" of the resulting system. This was also found to be the case in Michaelson and Scaife [61].

4.5.6 Possible Improvements to the Algorithm

The Occam2 was implemented directly from the prototype guided by the presence of the recognised HOF skeletons. Once this was developed, however, it could be optimised, for instance by making better use of the geometric constraints within the program.

Instead of collecting segments of lines from processing the image portions the workers could be organised to directly communicate overlapping points between each other, so that the merge phase becomes unnecessary. This technique has successfully been used for region-growing in depth data [66] but requires much more sophisticated control and an indexing system for the patches that includes the processor number as part of the index and a global list of indices to be maintained by the farmer processor. This was conceptualised in the GDIW skeleton but it was not worth reimplementing the existing edge tracking phase. Another way of looking at this would be to consider such improvements as post-compiler optimisations, much in the same as way many imperative language compilers perform optimisation on their compiled output.

Subsequent work into integrating the edge tracking component into the overall system led to the development of the DF skeleton.

4.6 Summary and Conclusions

This chapter presented a methodology for developing parallel vision algorithms through SML prototype instrumentation followed by translation into parallel Occam2. Profiling methods were discussed and the advantages of dynamic profiling were used to justify the method of relating measured parameters from the prototype to the performance of dummy parallel constructs implemented on the target machine through numerical models. A methodology was outlined and implemented in two cases; a simple Gaussian filter and a more complex edge tracking example.

The case studies showed the difficulties in getting reliable predictions from SML prototypes over a wide range of operational characteristics of the parallel constructs. The technique of using numerical performance modelling gives very reliable results within a narrow range of parameters using very little *a priori* knowledge about the parallel construct. The main problem here is that a substantial database of numerical models would be required to give a general purpose performance model for each parallel construct. This is feasible, however, and could be built up over a period of time as applications were developed. There are always factors which are difficult to predict, however, for instance the problems of uneven load balancing for coarse grain parallelism, which this method will have problems with.

The remainder of this thesis concentrates upon applying the methods proposed in this chapter in the construction of substantial vision applications, rather than on improving and updating the methodology itself.

Chapter 5

System Design

5.1 Introduction

This chapter concerns the selection of algorithms and parallel constructs for a multisource segmentation system composed of discrete modules intended to be implemented in parallel with suitable "glue" to bind them together. The construction of a major application of this nature using algorithmic skeletons as a design tool has not previously been attempted and there are also very few complete parallel computer vision systems reported in the literature.

The design process has been divided up into several areas:

- The characteristics of the target architecture have to be analysed. The actual hardware itself is fixed, a network of T800 transputers, but this system is flexible in terms of wiring topology. A number of standard topologies are presented and their suitability for implementing abstract parallel constructs and for handling image data are assessed.

- A set of skeletons has to be devised. In practice these need not be fixed during the lifetime of the project but if any new skeletons are added during implementation they should be suitable for the existing hardware and topology. Here, the final set of skeletons that were implemented is discussed.

- Given the application task (that of processing multisource image data) suitable algorithms have to be selected from those presented in the literature. It would have been inadvisable to attempt fundamental vision research coincident with developing the entire system so well-studied vision algorithms suitable for implementation on the given architecture are used.

All of these are interrelated and they have no specific order of precedence, the design could be top-down or bottom-up. What is presented here is a summary of the major design decisions that occurred during the implementation of the system.

5.1.1 System Overview

The system to be constructed is for pose identification using both range image information and greyscale intensity images, concurrently. Model descriptions of specific objects are used and input consists of one intensity image, one depth image and one model description per execution of the system. Continuous on-line recognition is a different task.

The matching algorithm is based on pose clustering [267] which has been adapted to work on both intensity and depth images in a uniform cluster space [268]. This means that each image can be segmented separately with the information combined at the matching stage.

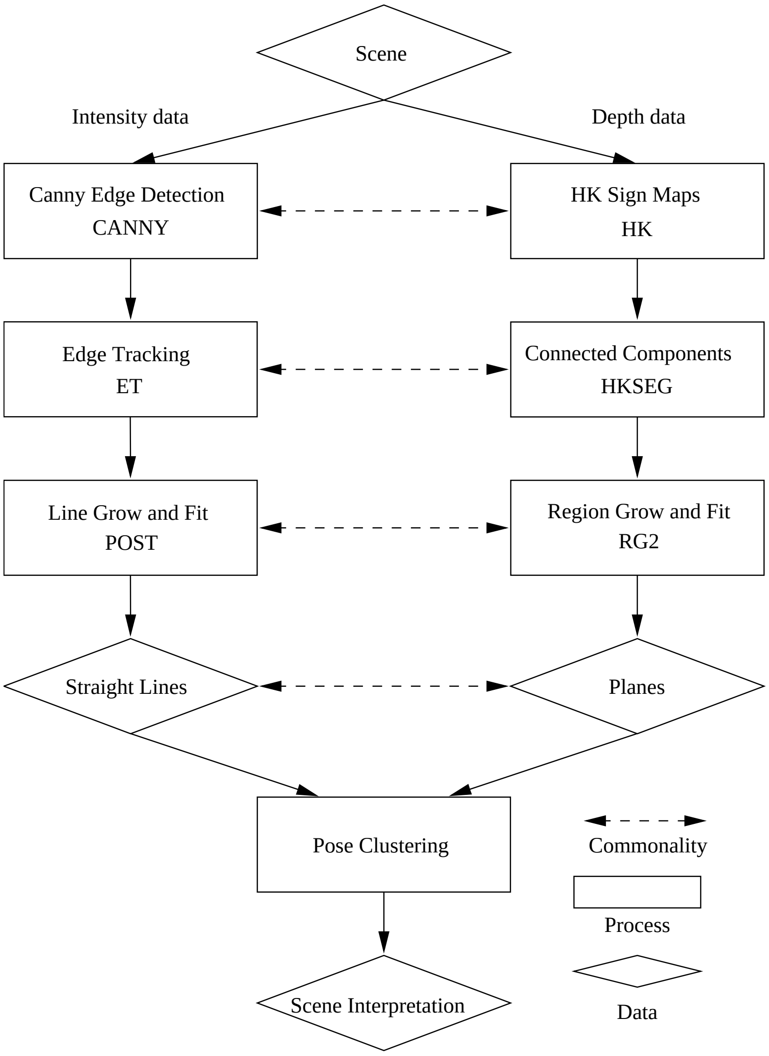


Figure 5.44: Modular Structure of the Parallel Object Recognition System

This section describes the depth and intensity segmentation components for the overall system depicted in Figure 5.44.

The mnemonics used in the block diagram are used throughout the thesis and are as follows:

- **HK**: Calculate mean and Gaussian curvature for depth image.
- **HKSEG**: Perform connected components on HK curvature values.
- **RG2**: Seed and grow within connected components patches.
- **CANNY**: Calculate Canny edge strength and angle in intensity image.
- **ET**: Edge track based on edge strength and angle.
- **POST**: Post-process tracked lines by scan and fit.

This diagram shows the overall structure of the proposed system and indicates the common structure between depth and intensity routes. For example, Canny edge detection for intensity data has structural and algorithmic similarities to the generation of HK sign maps for depth data. This similarity is desirable in order to minimise design and implementation effort. For instance, the calculation of Mean and Gaussian curvature given in Besl [269] proceeds by mapping a window over the image, computing the surface partial derivatives within the window which leads to the calculation of the surface curvatures. The Canny edge detector [248] also consists of a series of convolution operations to calculate edge directional components. These two are combined into a single operation called *local window operations* which have the same parallel structure but different instantiations.

There are other conceptual and structural links between the two routes. A region-growing approach is used for depth segmentation based around the method proposed in [247]. Part of the motivation for this is that it bears some similarity to edge tracking as a means of extracting lines from images [270]; both algorithms start from seed regions and grow these into image features using a similarity or difference measure. The relationship is not so close in this instance, however, since differences in the fundamental nature of the data (one looks for 1D features in 2D images, the other looks for 2D features in 3D images) mean that different parallel structures are required to give efficient parallel implementations in each case. Both of these are based on a geometric decomposition of the image. For the depth route a coarse segmentation is based around a connected components algorithm [271] run within geometric boundaries with inter-worker communications to connect patches which span boundaries. For the intensity route lines are tracked within the geometric boundaries followed by a second phase of processing where partial lines are compared and merged.

Finally, post-processing of detected features comprises fitting using a general multidimensional linear least squares method based on singular value decomposition [272]. This is applicable to any feature which is linear in its coefficients with suitable sets of basis functions, and is followed by a final growing of features based on the fit. The range patches are grown using a geometric decomposition with task spawning where geometric boundaries are encountered. The lines tracked by the edge tracking algorithm are selected or rejected based on the quality of the fit. The reason for this difference between the two routes is that the edge tracking similarity criterion allows tracking close to the ends of the lines but the nature of the HK sign data is such that the edges of the patches are generally missing from the segmentation, resulting in a loss of positional accuracy. The region-growing method allows the patches to be grown nearer to the edges.

5.2 System Architecture

The system will be required to host several different modules operating on different types of data. The expected overall size of the system will also be large in terms of code and data. This means that as much common code and data storage as possible is required. Thus, minimising resource usage was one of the main design criteria and turned out to be the major problem in the construction of the system. It also influenced many other aspects of system design, including the selection of both the communications model and the basic algorithms, see Section 5.5.

5.2.1 Message Passing

The use of a general message passing system would give the flexibility and control over data and event timing that is required by the system.

Tiny

Since the Meiko Occam2 implementation does not provide support for generic message passing systems, the Tiny routing harness [273] is employed which gives support for various types of message passing over any number of virtual channels, indexing processes on individual transputers. The communications overheads for Tiny are actually quite small for most sizes of messages since the routing tables are highly optimised and Tiny is itself implemented mostly in transputer assembly language. The major restriction is that there is a maximum message size which is the same for all processors on the network. Setting this too large causes Tiny's internal buffers to increase in size to the point where they take up a substantial portion of the available memory on each transputer. It also means that larger messages have to be split up and combined at the receiving end. Only one processor per logical channel is able to broadcast messages which would suggest the use of a single master processor for the system and the use of Tiny's indexing system to partition the remaining processors into different domains.

Tiny can handle both *blocking* and *non-blocking* message modes and can also define whether message order is preserved or not. One of Tiny's optimisations is to send messages by alternative routes if the system is busy and this can result in message order being scrambled. It was decided to use blocking messages, preserving message order. This is to give the system robustness at some slight expense of performance. Blocking is generally considered slightly more fault tolerant than non-blocking and is also easier to program due to the tighter control over messages.

Generic message passing

Similarly, a subsystem of generic message passing has been developed. This is simply a set of routines to generate messages by installing Occam2 data types in flat arrays, indexing with a unique message code and sending the message to the Tiny routines. Figure 5.45 illustrates the generic transmissions code that isolates the main processing from communications specific concerns. General purpose `PROC`s are provided in libraries, for example the `SetCounter` routine, which build the flat message from the unique code `IncCounter` and install the data using the array access routines `InInt16` which are provided for all Occam2 base types. Once built, the buffer is sent through the multiplexor `SendMux` using the specified channel `buf` and is despatched by the multiplexor process `MuxSend`. The receive process is effectively the reverse of the scheme shown here.

Another point, here, is that it is prudent to have one transmitting process and one receiving process for each transputer. Multiplexors can be provided for multiple transmission routines and communication between receiving and transmitting processes is through shared variables or message passing. This gives a generic view of processor communications illustrated in Figure 5.46. Here, all messages are received by a single task and distributed by Occam2 channels or by shared variables and all output messages are sent to a single multiplexor for transmission. On worker processors the only two tasks are for the depth and intensity processes. On the monitor processor, there are several tasks for depth and intensity processing, external communications with model matching and a primitive control system.

Rian

One item of proprietary software that is effectively pregiven is the Rian system, developed at Edinburgh University. This manages system resources such as file accesses and screen

```
PROTOCOL SEND
  CASE
    REQ
    FREE
    MESSAGE;INT;INT::[]BYTE
    DONE
:
[SendMuxSize]CHAN OF SEND toWorkers,fromMux:

PROC MuxSend([]CHAN OF SEND toWorkers,fromMux)
  INT finished:
  [3]INT proc,size:
  [3][TxRxSize]BYTE msg:
  SEQ
    finished := 0
    WHILE finished < (SIZE toWorkers)
      ALT w = 0 FOR (SIZE toWorkers)
        toWorkers[w] ? CASE
           REQ
            fromMux[w] ! FREE
          MESSAGE;proc[w];size[w]::msg[w]
            SEQ
              IF
                proc[w] = (-2)
                  finished := finished + 1
                TRUE
                  Send(toDebug,toTiny,proc[w],[msg[w] FROM 0 FOR size[w]])
              fromMux[w] ! DONE
:

PROC SendMux(VAL INT buf,proc,VAL []BYTE msg)
  SEQ
    toWorkers[buf] ! REQ
    fromMux[buf] ? CASE
      FREE
        toWorkers[buf] ! MESSAGE;proc;(SIZE msg)::msg
    fromMux[buf] ? CASE
      DONE
        SKIP
:

PROC InInt16([]BYTE Arr,VAL INT Indx,VAL INT I)
  SEQ
    Arr[Indx] := BYTE (I / 256)
    Arr[Indx + 1] := BYTE (I \ 256)
:

PROC SendCounter(VAL INT buf,proc,cntr,incdec)
  SEQ
    TxBuf[0] := IncCounter
    InInt16(TxBuf,1,cntr)
    InInt16(TxBuf,3,incdec)
    SendMux(buf,proc,[TxBuf FROM 0 FOR 5])
:

PAR
  MuxSend(toWorkers,fromMux)
  SEQ
    SendCounter(2,mastId,1,1)
```

Figure 5.45: Message Transmission Layer in Occam2

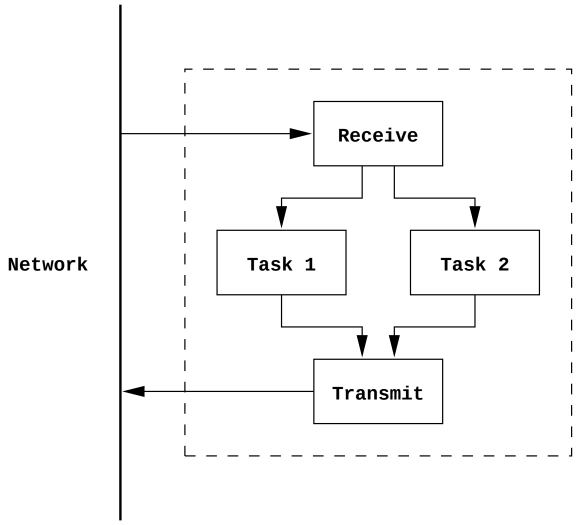


Figure 5.46: Generic Model for Processor Communications

handling (allowing screen logging to files) and also manages the supervisor bus which can be used for very slow debug messages (these can also be logged to file). This system is relatively unobtrusive, however, and can work under Tiny without any problems.

5.2.2 Processor Partitioning

The algorithms were selected and the system designed to try and maximise code reuse throughout the system, however, there are two distinct threads of data processing attributable to the intensity and depth routes, see Figure 5.44. In the absence of any data fusion these can occur completely independently. This would suggest a partitioning of the system into depth and intensity processors but there are various ways in which this could be done.

One possibility would be to have separate subnetworks of transputers for each route with a separate master processor for each. One of the objectives of the system, however, is to investigate reallocation of resources. The allocation of processors to the two routes is the most promising alternative but this is precluded by physically partitioning the system.

Another possibility is to have a single network of processors but to partition the network either by loading different code onto each type of processor when the system is booted or by loading the code and data for both routes onto all the processors and partitioning them by software flags. These three possibilities are illustrated in Figure 5.47.

A variation of the software partitioning scheme would be to overlay both routes on all the processors and have them run simultaneously on each processor. Unfortunately, having both routes running concurrently on each processor is not possible due to the memory requirements of the two routes. Part of the problem is that intermediate results need to be stored to allow sections of code to be run again. This would be required by an iterative improvement scheme

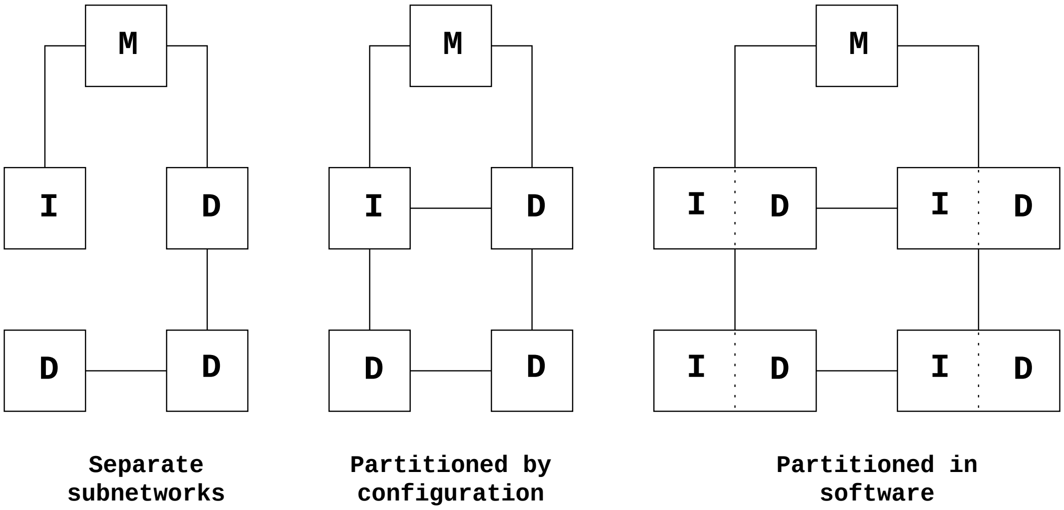


Figure 5.47: Partitioning Options for Depth and Intensity Processors

or control system which may be planned as a future development of the system. Another problem is that, for low-level routines, preloading of data requires substantial image storage on each worker.

The option of partitioning when the system is booted is viable but not as flexible as the others. It also has lower memory requirements than software partitioning. Thus the option of using a set of uniform worker processors with both sets of code on each worker but with shared temporary data gives the best flexibility for the available resources.

5.2.3 Processor Topology

The communications model and processor partitioning have been addressed but this still leaves several possibilities in terms of wiring topology. In keeping with the generic nature of the system design there is no reason why any physical arrangement of processors should be precluded by the design decisions up to this point. Much of the previous work in this area [240, 40, 61] has concentrated on linear pipelines of processors because of the ease of programming and the possibility of theoretical analysis. Some of the algorithms proposed here, however, have much greater data volumes and will therefore require much greater communications loads and so are likely to benefit from topologies with a much smaller diameter. Probably the two most popular topologies for vision systems are 2D arrays, with optional wraparound giving toroidal arrangements, and hypercubal arrangements. A similar wraparound is possible for the loose links on hypercubal arrangements. Figure 5.48 shows some of the arrangements possible. Five topologies are possible for elements with four external links. The linear pipeline is where the elements are connected to two others and only uses two of the four links. The ring is created by joining the ends of the pipeline. A 2D array is effectively sets of linear pipelines set side-by-side. A toroid is generated by “wrapping” the edges of a 2D array together. The "hypercube" arrangement shown is a limited form of hypercube where the processors are connected in maximally connected groups of four. Four of these groups are then maximally connected, and so on. The two equivalent representations illustrated here show the relationship between the hypercubal arrangement and the 2D arrangements. The figure on the left is not the usual illustration of a hypercube but was found to be useful in calculating link maps for the manually generated wiring files.

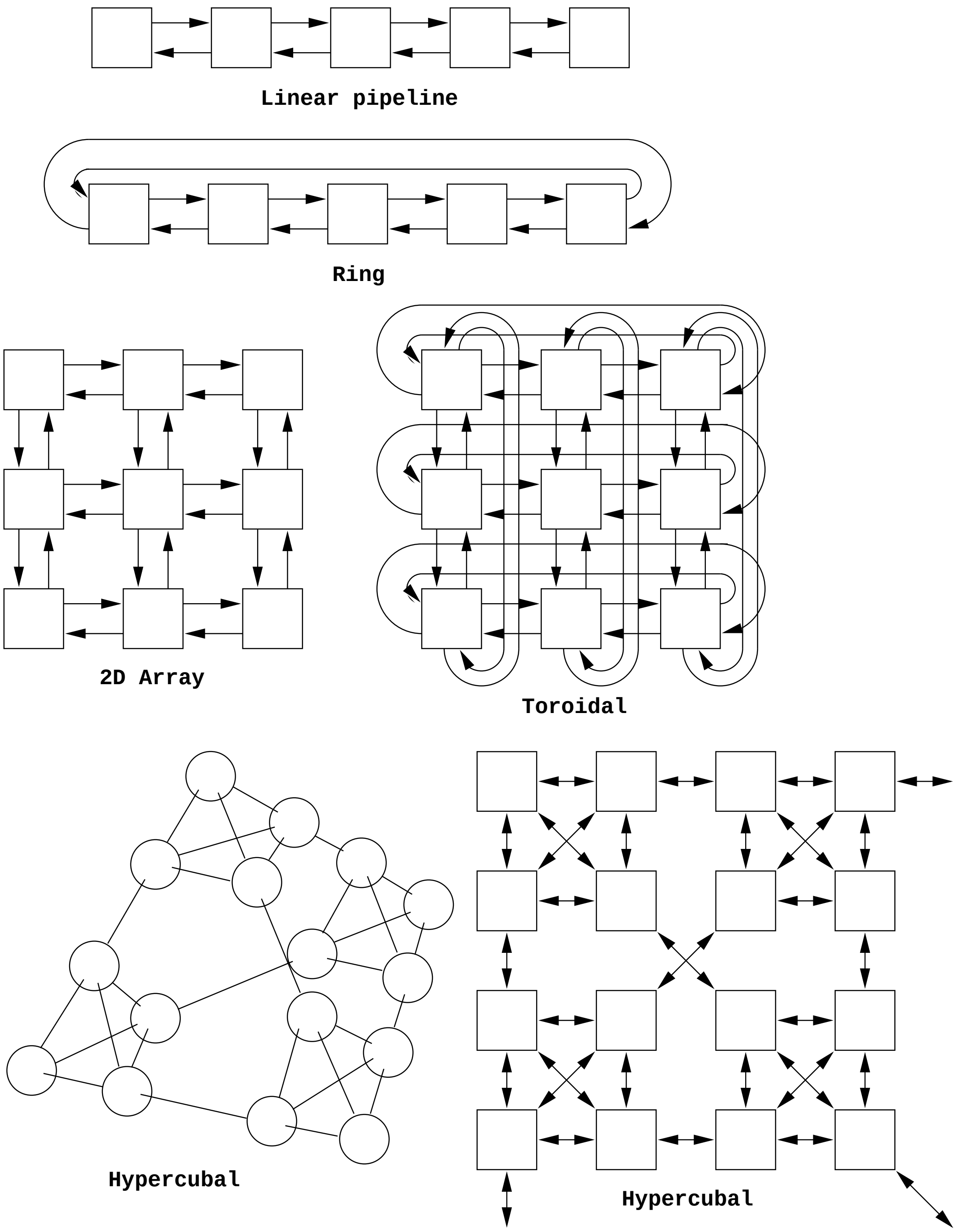


Figure 5.48: Topology Options for Uniform Processors

There are several ways of characterising a topology, depending on what aspect of performance is being considered. The maximum distance between any two processors is one method of quantifying the density of packing of processors. This is known as the topology *diameter*.

n	Linear	Ring	2D Array	2D Toroid	Hypercube
4	3	2	4	2	1
16	15	8	8	4	3
64	63	32	16	8	7
256	255	128	32	16	15

Table 5.12: Growth of Maximum Interprocessor Distance for Topology

The distance between two processors is the minimum number of links traversed to send a message from one processor to another. In the ideal case, all processors would be connected to all other processors and the maximum distance would be one. Due to the two-dimensional nature of silicon technology, however, there are a limited number of external connections from any one device. Even if this were a large number there would still be great complexity in the external wiring. Bus-based architectures are possible but the performance then becomes limited by the total network traffic rather than the distance between processors.

A k-dimensional hypercube of processors is characterised by N, the number of nodes, b, the base of the node indices and k, the number of digits in the node indices. Two nodes are connected if their indices differ in exactly one base b digit. If $N = b^k$ then the hypercube is said to be *complete*. Thus, Figure 5.48 shows a complete 16 node 2-dimensional hypercube of base 4, signified $(N, b, k) = (16, 4, 2)$.

For a linear pipeline of n processors, the maximum interprocessor distance is $n - 1$, for an even ring of n processors it is $n/2$, for a square 2D array of dimensions $\sqrt{n} \times \sqrt{n}$ it is $2\sqrt{n}$, for a toroid of dimensions $\sqrt{n} \times \sqrt{n}$ it is $\sqrt{n}$ and for a hypercube of the type shown with n processors it is $2^{\lfloor \log_4(n) \rfloor} - 1 = \sqrt{n} - 1$. The growth of these values with n is illustrated in Table 5.12.

One point from this table is that for small numbers of processors the differences between the maximum interprocessor distances is less significant than for large numbers of processors. Thus, for small numbers of processors, simpler arrangements are viable but as the number of processors increases more complex networks become essential. The one dimensional topologies run out of steam very quickly. More than about 10 to 20 processors result in communications problems. The two-dimensional arrangements work up to larger numbers of processors, certainly up to the 32 processors available. There is actually very little difference between the 2D toroidal and hypercubal arrangements.

The maximum interprocessor distance does not, however, give a very good reflection of the utility of the network for computational purposes. The designers of any program will attempt to minimise transmissions over large distances and arrange for processors which are local to each other to to run tasks which communicate with each other. A better measure of the density of a network is therefore the *mean interprocessor distance* which can be defined as:

$$\frac{1}{n(n-1)} \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} dist(i,j) \tag{5.31}$$

where $dist(i,j)$ is the distance between processors i and j, measured as the shortest number of links needed to traverse between them. This is more difficult to determine than the maximum interprocessor distance but analysis is possible. For a 2D array of processors of dimensions $m \times n$, $dist(i,j)$ is $|x_i - x_j| + |y_i - y_j|$ where x_i is the column number of processor i and y_i is the row number. This gives:

$$MIPD_{arr}(m,n) = \frac{1}{mn(mn-1)} \sum_{l=0}^{m-1} \sum_{k=0}^{n-1} \sum_{j=0}^{m-1} \sum_{i=0}^{n-1} (|l-j| + |k-i|) \tag{5.32}$$

which evaluates to:

$$MIPD_{arr}(m,n) = \frac{m+n}{3} \tag{5.33}$$

Setting $m = 1$ gives $(n+1)/3$ which is the expression for the linear topology. An equivalent analysis for the toroidal arrangement is possible but with a slight complication in that the expression for the sum of distances along a row or column depends on whether the row/column has an even or odd number of processors:

$$SUMD(n) = \begin{cases} \sum_{i=0}^{n-1} \left|(\frac{n-1}{2}) - i\right| = \frac{n^2-1}{4}, & n \text{ } odd \\ \\ \sum_{i=0}^{n-1} \left|(\frac{n-2}{2}) - i\right| = \frac{n^2}{4}, & n \text{ } even \end{cases} \tag{5.34}$$

This leads to the expression:

$$MIPD_{tor}(m,n) = \frac{1}{mn(mn-1)} \sum_{j=0}^{m-1} \sum_{i=0}^{n-1} (SUMD(n) + SUMD(m)) \tag{5.35}$$

which can be expressed as:

$$MIPD_{tor}(m,n) = \frac{mn(m+n) - (n \text{ } mod \text{ } 2)m - (m \text{ } mod \text{ } 2)n}{4(mn-1)} \tag{5.36}$$

This evaluates to $\frac{m+n}{4}$ for odd-sided arrays and $\frac{mn(m+n)}{4(mn-1)}$ for even sided arrays. Setting $m = 1$ gives $\frac{n+1}{4}$ for odd-numbered rings and $\frac{n^2}{4(n-1)}$ for even-numbered rings.

Tables 5.13 and 5.14 tabulate the $MIPD$ function for arrays and toroids of up to 6×6. These illustrate that the difference between these two topologies is not as different as the factor of 2 that the maximum interprocessor distance would suggest.

m,n	1	2	3	4	5	6
1	N/A	1.0	1.3333	1.6667	2.0	2.3333
2	1.0	1.3333	1.6667	2.0	2.3333	2.6667
3	1.3333	1.6667	2.0	2.3333	2.6667	3.0
4	1.6667	2.0	2.3333	2.6667	3.0	3.3333
5	2.0	2.3333	2.6667	3.0	3.3333	3.6667
6	2.3333	2.6667	3.0	3.3333	3.6667	4.0

Table 5.13: Mean Interprocessor Distances for Array Topology = $(m+n)/3$

m,n	1	2	3	4	5	6
1	N/A	1.0	1.0	1.3333	1.5	1.8
2	1.0	1.3333	1.4	1.7143	1.8889	2.1818
3	1.0	1.4	1.5	1.8182	2.0	2.2941
4	1.3333	1.7143	1.8182	2.1333	2.3158	2.6087
5	1.5	1.8889	2.0	2.3158	2.5	2.7931
6	1.8	2.1818	2.2941	2.6087	2.7931	3.0857

Table 5.14: Mean Interprocessor Distances for Toroidal Topology

A similar analysis for the hypercubal arrangement was not attempted but an SML program was written to calculate the mean interprocessor distance by searching a graph structure for the distances. This was not computationally feasible for hypercubes of more than two levels but a comparison is possible for 4 and 16 processors. This is shown in Table 5.15.

This shows that the hypercubal arrangement does have a slightly denser configuration than the equivalent array or toroidal arrangements. It is doubtful whether this gives any significant edge in terms of general programming for small numbers of processors (certainly up to the 32 processor maximum of the system used in this thesis). In practice, the hypercubal arrangement was used for most experiments despite the greater complexity of wiring than a 2D toroid.

To verify these calculations, the emulated farm program discussed in Section 4.4.3 was run on three of the topologies, linear pipeline, 2D array and hypercubal under various conditions of communications and processing loads, these are presented in Figure 5.49 which shows the results of these tests and confirms the predictions. The `Os` value is the size of the output work packets and `Tf` is the (uniform) time to process a single work packet, 1000 packets are processed in total so the results are more or less steady state processing. The communications load is low at 16 bytes and high at 3800 bytes and the processing time is low

Level	Procs	Lin	Ring	Arr	Tor	Hc	Hct
1	4	1.6667	1.3333	1.3333	1.3333	1.0	1.0
2	16	5.6667	4.2667	2.6667	2.1333	2.2	2.1

Key: Lin = linear, Ring = ring, Arr = 2D array, Tor = 2D toroid, Hc = hypercubal and Hct = hypercubal but with loose links tied as for ring and toroidal.

Table 5.15: Mean Interprocessor Distances

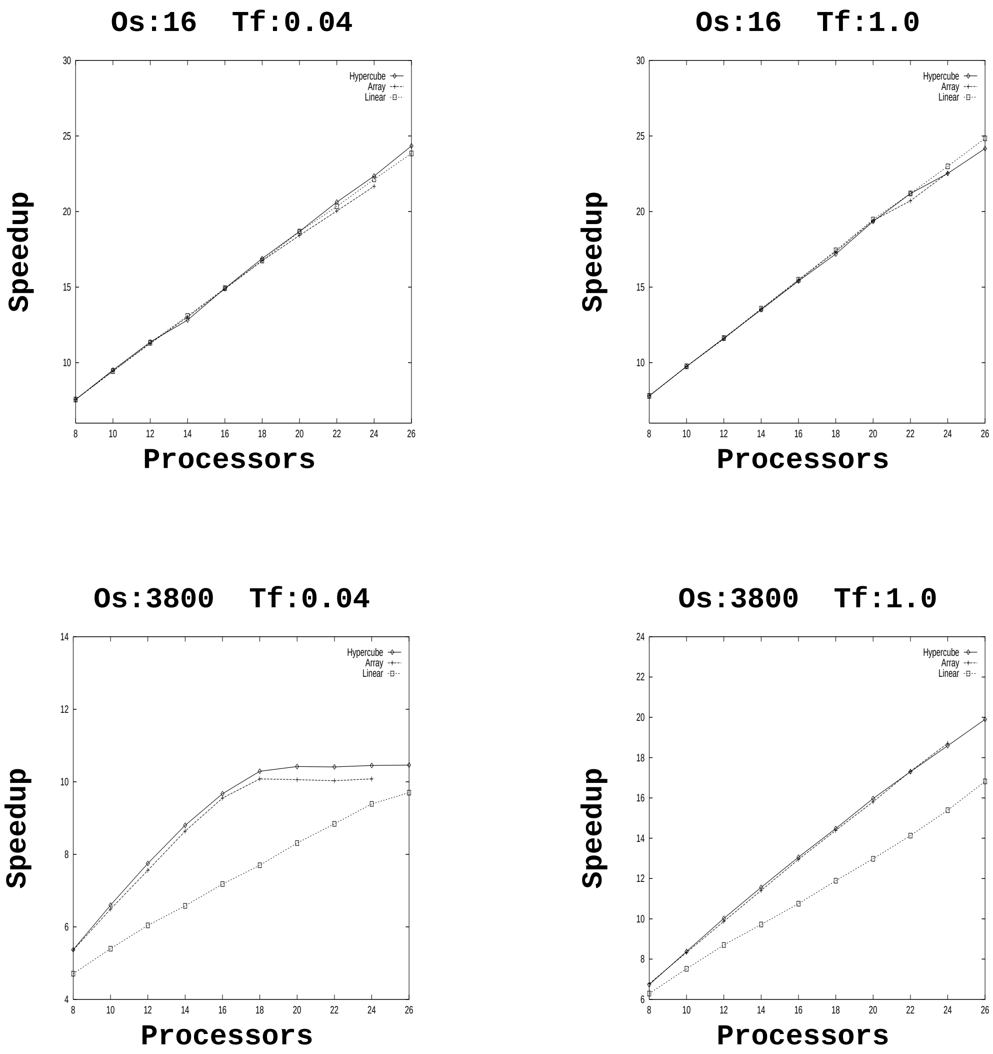


Figure 5.49: Topology Performance for Emulated Farm

at 0.04 seconds and high at 0.1 seconds, relative to the processor/network characteristics. When the communications loads are light there is very little difference between the three arrangements. Under heavy loads the linear arrangement performs much worse than the other two and there is very little difference between array and hypercube arrangements with maybe a slight advantage for hypercubal topologies when the processing load is light and the communications load is heavy (this probably accentuates the communications overheads rather than any topological differences).

5.2.4 Data Placement

Preloading of data has been found to be essential for some of the algorithms used [61], particularly low-level processing. Taking this into account along with other experience from the development of various image processing algorithms led to the development of a global data placement strategy. The maximum size of the various buffers depends on whether the system is expected to output the segmentation for further processing or not. The following figures are for a version with no such output routines which leaves more memory for line, patch and image storage.

The major items of data that require permanent storage on the system are as follows.

Depth and intensity images. The maximum image size that can be handled is about 256×256 pixels. This is not sufficient for many applications but there is available data from laser range-finders which can be reduced in size down to the required 256×256. Intensity images can be stored in `BYTE` format and so each intensity image requires 64k of storage. Depth images require `REAL32` storage and so require 256k. Due to the preloading recommendation both of these images need to be present on all the workers and the master processor.

Work image. In addition, the system requires the ability to store iconic data for intermediate results, one for depth and one for intensity although the strategy of partitioning processors by depth/intensity allows these to be merged into one. This work image is another 256k. The master processor also needs an extra `REAL32` iconic image for rendering depth and intensity data for output to files.

Workspace. Both the master and worker processors have a buffer which can be reused by different section of code which do not run concurrently. These are accessed by Occam2 `RETYPE` statements and the buffer is currently set at about 300k.

Segmented lines and associated data. The relationship between the number of lines, the maximum size of a line and the total memory allocation for lines is not simple due to

Memory	Worker	Master
Intensity image	56216	56216
Depth image	207488	207488
Work image	207500	409132
Calibration data	693	693
Patches	794084	794884
Lines	198412	198412
Communications	8208	8208
Constants	372	476
Params	196	196
Global variables	1548325	1201261
Function variables	555987	625883
Workspace	614400	307200
Total	4191881	3810049

These values were calculated with Tiny workspace 50k, work buffer 300k, image maximum size 191×264, 11 patches of 9500 points and 90 lines of 250 points.

Table 5.16: Total Memory Requirements for Segmentation System

the additional data associated with each line, for instance fit coefficients, line endpoints, processor allocations and others. In addition, the worker processors define substantial amounts of storage for temporary lines, although most of these are stored in the shared workspace. A typical number of lines required for the storage of decomposed lines segments in images of up to 256×256 is around 90 lines each of about 250 points, `REAL32` x and y giving 175k for the major line buffer.

Segmented patches and associated data. Again there is a substantial amount of auxiliary and temporary data associated with patch data. Due to the two-dimensional nature of the data, however, much more space is required for patches than lines. The maximum number of patches is currently 11, each of 9500 points which is only just suitable for the test images used. Note that patches are stored as `INT` x and y arrays giving a total of 816k for the main patch data.

Tiny internal buffers. The amount of space required by the routing software depends on the number of channels and the maximum message size which are global system wide constants. Currently, this buffer is set at about 50k.

An SML program was written to run through the variable allocations in the completed system and calculate the total amount of each of the various memory usages in the system. This data is summarised in Table 5.16.

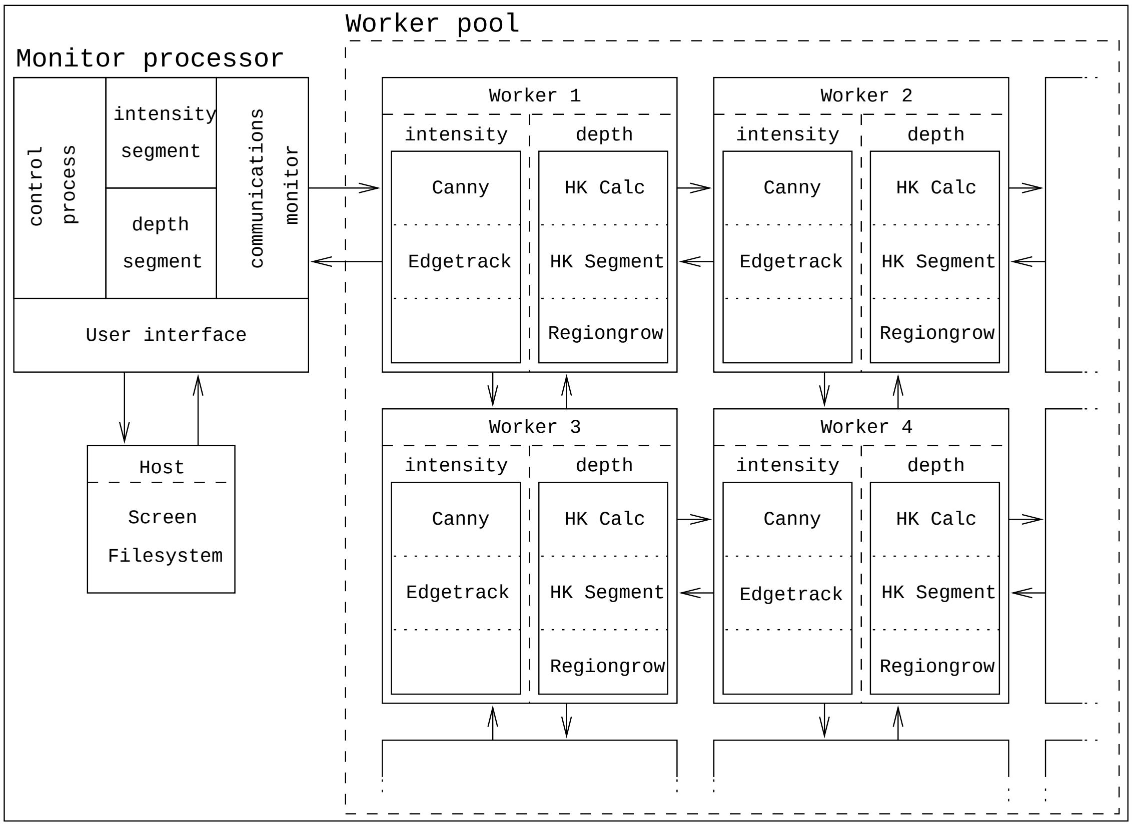


Figure 5.50: Organisation of the Parallel Hardware

5.2.5 Parallel System Organisation

The target platform is a network of T800 transputers with 4 MBytes ram per transputer. A system is required which is flexible in terms of topology (both data and code) and which is reasonably easy to program. As indicated previously, care was taken in choosing algorithms to maximise the likelihood of common structure between the various components. The hardware was then designed to implement several simple parallel constructs in a uniform manner. Figure 5.50 is a schematic of the resulting parallel system configuration.

Since control and monitoring of the progress of the calculations is desirable and since load balancing may also be required the system was implemented as a single monitor processor and a set of uniform worker processors. The monitor controls all higher level activities such as file access and user interface and all worker activity is dependent upon commands from the monitor.

5.3 Skeleton Design

On this platform a small number of parallel constructs have been implemented which are instantiated with the code required to process the vision algorithms. Note that the number of these which would be required was not known in advance but new ones were created or existing ones generalised and parameterised as system development proceeded. Past experience has shown that complete systems can be built using only a very small number of these constructs [61] but for programmer convenience and flexibility in determining parallelism several such constructs were developed in the course of this work. In fact the system was built using only four such constructs:

Name	Description
GD	Geometric decomposition
DF	Generalised data farming
TF	Task farming with task spawning
GDIW	Geometric decomposition with interworker communications

Figure 5.51: Four Skeletons Proposed for Parallel Object Recognition

5.3.1 Geometric Decomposition (GD)

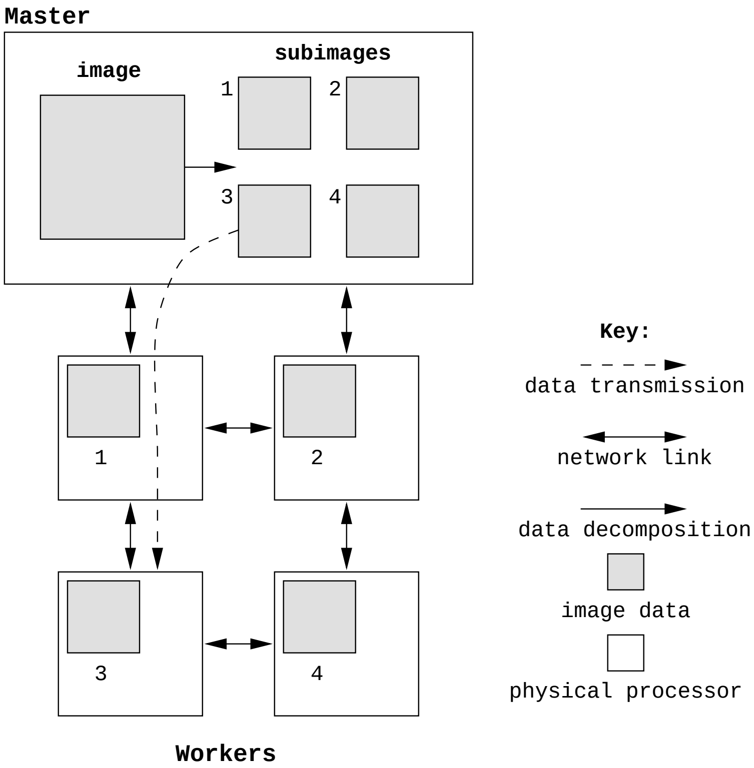
The first of these is a geometric decomposition over data where the data is divided up into a number of equal-sized sections and each section is processed by a different worker. This skeleton is applicable where the data in each element can be processed independently of the other elements and the amount of work associated with each element is the same. The simple expedient of dividing the data up equally among the workers leads to an optimal solution in terms of communications costs. Note that the data could be iconic in which case the image is divided up into a number of subimages.

The Standard ML code for this skeleton has already been shown in Figure 4.26 and the operation of this skeleton has been illustrated in Section 4.4. Figure 5.52 graphically illustrates the parallel operation and Figure 5.53 shows Occam2 pseudocode for this process. A number of points have been noted during the use of this skeleton in constructing vision algorithms:

- Although there is no overlap between the subimages in the illustration here, it was found necessary for window operations which required data from outside the subimage. This presents no problems since processed data is only returned from within the subimages. In addition, the preloading condition means that this extra data is already present on each worker so there is no communications overhead for the overlap data.
- Since there is no communication between the subimages as they are processed the processor topology is relatively unimportant, especially when the task messages consist of co-ordinates into preloaded data.
- The decomposition could be of several different types, for instance row-based, column-based or rectangular subimages. Indeed, the subimages need not cover the entire image,

for example a single feature in the image could be treated using a geometric decomposition. In this instance, however, some means of describing the subregions is required, for example a processor allocation iconic map.

- The processes of distributing the input data and gathering the output data need not be included in the skeleton's definition. These operations can be viewed as skeletons in their own right (communications skeletons) and can sometimes cancel each other out for optimisation purposes, for instance when data is gathered and then redistributed immediately.



The image is split up into the same number of subimages as there are processors and each processor runs the instantiation function on each subimage.

Figure 5.52: Operation of the Geometric Decomposition (GD) Skeleton

5.3.2 Generalised Data Farming (DF)

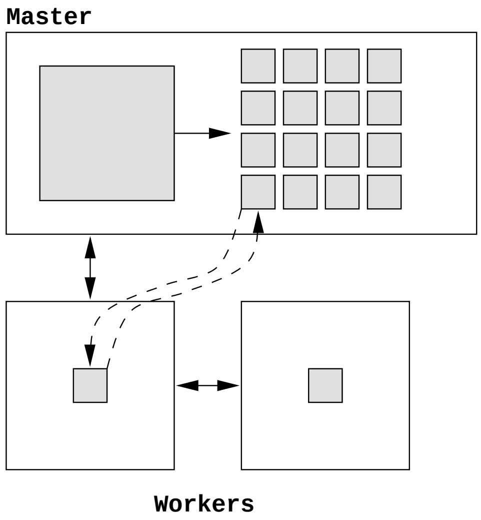
The second construct is a general processor farming harness. Several routines require decomposition over irregular data, for instance an arbitrary list of patches of various sizes. This decomposition can be handled by worker farming in which work allocations are doled out by the master giving primitive, inherent load balancing. This method is only efficient, however, if there are more data elements than processors and one data element does not dominate all the others.

The SML description of this construct is the same as for the geometric decomposition, the

Master	Worker
<broadcast parameters>	<receive parameters>
<broadcast static global data>	<receive static global data>
P := <number of processors>	
<decompose data into P	
equal-sized elements>	
SEQ p = 0 FOR P	<receive data element>
<transmit element p to processor p>	<process data element>
	<transmit worker free msg>
WHILE <workers not free>	
<receive worker free msg>	
<transmit data upload msg>	<receive data upload msg>
<receive local data into global data>	<transmit local data>

Figure 5.53: Occam2 Pseudocode for Static Geometric Decomposition

difference is in the parallel implementation. The process of modelling this construct will also be slightly different since a much finer decomposition is required which will be in a different region of the construct's performance space. Figure 5.54 illustrates the basic operation and Figure 5.55 shows Occam2 pseudocode for this construct. Note that again the data could be iconic in nature leading to farming of subimages.



The data is split up into many more subelements than there are worker processors. The farmer processor transmits work packets on demand until none are left and the workers are no longer processing any data. A worker receives a work packet, processes it and returns any output data and a task completion message. See Figure 5.52 for key.

Figure 5.54: Operation of the Data Farm (DF) Skeleton

5.3.3 Task Farming with Task Spawning (TF)

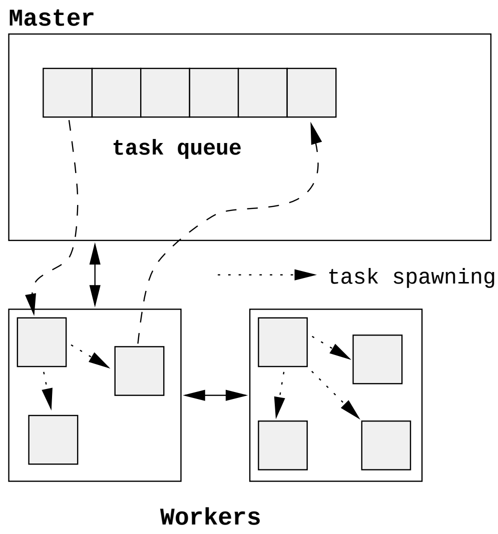
This is a generalisation of the previous skeleton in that the same worker farming methodology is used. In this case, however, each task sent out to the workers can generate more tasks which are then returned to the master and added to a task queue.

In this case the data has to be returned upon completion of each task since the output may be

Master	Worker
<broadcast parameters>	<receive parameters>
<broadcast static global data>	<receive static global data>
<decompose data into data queue>	
PAR	WHILE TRUE
ALT	<receive data msg>
<receive local data into global data>	<process data>
<receive worker free msg>	<transmit local data>
SEQ	<transmit worker free msg>
WHILE <data queue not empty>	
SEQ	
<find free worker>	
<get data from queue>	
<transmit data to worker>	
WHILE <workers not free>	
<wait>	

Figure 5.55: Occam2 Pseudocode for Data Farm

required in processing the spawned tasks. Note also that, in some instances, no decomposition is required prior to running this construct since a single initial task can spawn many others. The main disadvantage of this approach is that there can be significant startup and shutdown times if the granularity of the tasks is too coarse and that communication costs become more significant if the granularity is too fine. The efficient use of this skeleton therefore depends on choosing the correct granularity. The parallel implementation is illustrated in Figure 5.56 and the Occam2 code required is shown in Figure 5.57.



In this skeleton, the data is decomposed into a queue of task definitions. The data is then farmed out and the task queue maintained as new task definitions are generated by the worker processors. The workers process the task definitions and return output data, new tasks and a termination message.

Figure 5.56: Operation of the Task Farm (TF) Skeleton

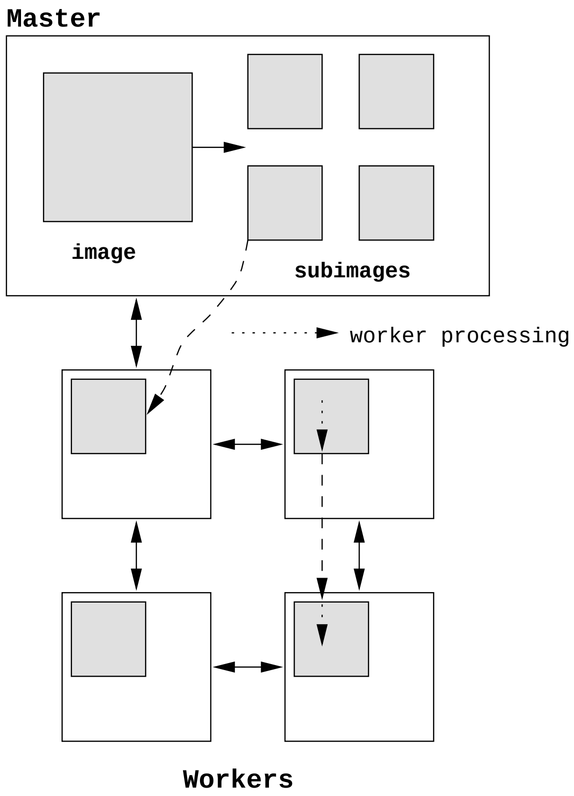
Master	Worker
`<broadcast parameters>`	`<receive parameters>`
`<broadcast static global data>`	`<receive static global data>`
`<decompose data>`	
`<initialise task queue from decomposition>`	
`PAR`	`WHILE TRUE`
`  ALT`	`  <receive task msg>`
`    <receive new task>`	`    <process task>`
`    <receive local data into global data>`	`    <transmit new tasks>`
`    <receive worker free msg>`	`    <transmit local data>`
`  WHILE <task queue not empty>`	`    <transmit worker free msg>`
`    AND <workers not free>`	
`    IF`	
`      <task queue not empty>`	
`        SEQ`	
`          <find free worker>`	
`          <get task from queue>`	
`          <transmit task to worker>`	

Figure 5.57: Occam2 Pseudocode for Task Farm

5.3.4 Geometric Decomposition with Interworker Communications (GDIW)

The final construct is slightly more complicated and applies to geometric decomposition of images. In this mode, the master divides the image up equally among the workers and then triggers the workers to process their subimages independently. In this case, however, data can cross inter-processor boundaries and the workers communicate among themselves to allow processing to cross these boundaries.

This requires more complex control than the other skeletons since each worker needs to be aware of which worker is processing which data so it can send boundary data to the correct worker. Termination is also more complex since all the workers must remain active until all the subimages have completed processing. Another problem is that if global consistency is to be maintained then the master must relate workers' local data to global data. This construct is also much more difficult to load balance, especially in the presence of sparse data. The main advantage of this method is that it is more optimal in terms of communications costs and memory usage than the farming methods. An interesting comparison can be made with the task farm skeleton since an algorithm expressed using interworker communication can also be expressed using the task farm by treating boundary information as a new task. In fact the source language construct for this skeleton is essentially the task farm skeleton but when a task is generated it can be placed in a list associated with a particular element of the decomposition. Figure 5.58 illustrates the operation of this skeleton and Figure 5.59 shows the Occam2 pseudocode.



This is a variant of the geometric decomposition skeleton (GD) with the added complication in that the worker processors can communicate directly. This requires a more complex scheme on the workers for the management of incoming messages and, in particular, clean termination of the algorithm.

Figure 5.58: Operation of the Interworker Communications (GDIW) Skeleton

Master	Worker
`<broadcast parameters>`	`<receive parameters>`
`<broadcast static global data>`	`<receive static global data>`
`P := <number of processors>`	`<receive decomposition>`
`<decompose image into P`	`PAR`
` equal-sized subimages>`	`  ALT -- read messages`
`<broadcast decomposition>`	`    <receive subimage>`
`SEQ p = 0 FOR P`	`      <send to main task>`
`  <transmit subimage p to processor p>`	`    <receive boundary>`
`PAR`	`      <send to main task>`
`  ALT`	`    <receive terminate msg>`
`    <receive local data into global data>`	`  SEQ -- main task`
`    <receive worker free msg>`	`    <read subimage>`
`  WHILE <workers not free>`	`      <process subimage>`
`    <wait>`	`      <transmit boundary to`
`  <broadcast terminate msg>`	`       relevant worker>`
	`      <transmit local data>`
	`      <transmit worker free msg>`
	`    WHILE <not terminated>`
	`      <read boundary>`
	`        <process boundary>`

Figure 5.59: Occam2 Pseudocode for Interworker Communications

5.4 Skeleton Usage

5.4.1 Skeleton Interface

The skeletons defined need to be presented to the programmer. For the methodology presented in this thesis the programmer requires both:

- *A definition of the source higher-order functions.* These can be characterised by their type-signatures and a description of the semantics in the source language.
- *The behaviour of the parallel equivalent.* The nature of the parallelism can be described principally by the decomposition and the associated parameters such as granularity.

These should be in a form which allow both the construction of functional prototypes and visualisation and control of the parallelism. The usage of a particular skeleton is therefore the combination of these two. The programmer's job is to write the application using as many of the HOFs as possible (sometimes more than one will be applicable at a given point in the program) and then use the profiling methods described in Chapter 4 to decide which will give good overall parallelism. The programmer is not shielded from the constraints of expressing the application using the provided skeletons. To a certain extent, however, some of the burden of deciding which skeleton instances will yield good parallelism is taken up by the profiling mechanism.

GD

The Geometric Decomposition behaves as a functional `map` and can be used as a direct replacement for it, `map` has type:

```
('a -> 'b) -> 'a list -> 'b list
```

The use of the GD skeleton, Figure 5.60, is restricted to the situation where `'a` is instantiated to a geometric area definition (`gd`) and `'b` is instantiated to a tuple consisting of a `gd` area definition plus the processed subimage from that area. The idea, here, is that the number of processors for the skeleton are fixed (separately for depth and intensity data) and that the whole image has been distributed to each worker processor.

To use this skeleton the programmer must generate the geometric decomposition using the `GeomDecomp` function, this has the following syntax:

```
GeomDecomp start_processor number_of_processors image_rows image_columns
```

```
type proc = int
type gd = proc * (int * int * int * int)
type 'a image = 'a list list
datatype itype = Depth | Intensity

val iGD : gd list ref
val dGD : gd list ref
val GeomProc : itype * int * int -> proc option

val GeomDecomp : int * int * int * int -> gd list
val gdCombine : (gd * 'a image) list -> 'a image

val GD : (gd -> gd * 'a image) -> gd list -> 'a image
val GD2 : (gd -> gd * ('a * 'b) image) -> gd list -> 'a image * 'b image
```

Figure 5.60: Type-signature for the GD Skeleton

This gives a list of area definitions in the form `(xmin,ymin,xmax,ymax)` which can be given as argument to the GD skeleton, notice that the number of processors is an argument to the decomposition. The GD skeleton itself has the following syntax:

```
GD ff gd_list
```

The GD instance function `ff` is passed one of the area definitions from the area definitions list and should return the processed image plus its area definition (which may have shrunk in the event of overlap). The processed image is recovered by GD by applying the `gdCombine` function to the list of values returned by the GD instance function.

Since the numbers of depth and intensity processors are fixed in the system in this thesis, the decompositions can be pre-computed (`iGD` and `dGD` for intensity and depth, respectively). The processor number (`proc`) is included in the decomposition definition for debug purposes, the `GeomProc` function gives the processor number for a given image co-ordinate for a given image type (intensity or depth). The `GD2` skeleton is an example of handling multiple output images from one input image, essentially this is the same skeleton but with different output data.

This scheme gives the programmer direct control over the granularity in the functional prototype at the expense of some additional coding effort. Strictly speaking, the entire skeleton could be wrapped up in a single function with the same type-signature as `map`, handling all decomposition issues internally but one purpose of the protoytpe is to allow experimentation with different parallel versions without the complexity of implementation on a parallel machine.

This skeleton is particularly suitable for regular image processing applications over local areas, for example the local window operations in Section 3.4.4. It also suitable for pixel-wise operations such as those in Section 3.4.1.

DF

The Data Farm skeleton has a very general interface, identical to the `map` function, Figure 5.61, for which it is a direct replacement.

```
val DF : ('a -> 'b) -> 'a list -> 'b list
```

Figure 5.61: Type-signature for the DF Skeleton

Since there can potentially be a great number of `map` instances in a functional program the programmer can use some initial judgement in deciding which to implement as DF instances and which to leave as sequential `maps`. For example, `maps` over empty or singleton lists will give no parallelism at all. In addition, the parallel implementation presented here is a very coarse-grain implementation so fruitful `maps` are likely to occur at the top-level of the code rather than inside deeply nested function calls. Once the trivial `maps` have been pruned and the remaining `map` instances converted into DF instances, the full profiling methodology can be used to decide which to implement in parallel.

There are many situations to which this skeleton can be applied so it is difficult to present quantitative guidelines for the use of the DF skeleton in vision algorithms. Any list data for which processing the list elements outweighs communications costs is applicable. It is also possible to convert other data into list form, for example an image into a list of subimages, provided the conversion does not outweigh the parallel performance gain. The farming parallel implementation means that this skeleton is preferred to the GD skeleton for irregular data since a limited degree of load-balancing is built into the skeleton. This was seen in the edge tracking example in Section 4.5.

TF

The Task Farm skeleton is more complex in that explicit handling of intermediate results is called for. This is to allow the skeleton to expose parallelism by manipulating these partial results. This requires two instance functions one to process tasks and one to combine the partial results. The sequential implementation of a task processing function takes the task and returns the processed task plus any newly created tasks. The combine function is expected to manage tasks in such a way as to prevent duplicate task processing and also to accumulate the final result of the computation. Figure 5.62 shows the parameters to the TF skeleton prototype.

```
type no_procs = int
type task = 'a
type worker_result = 'b
type final_result = 'c list

fun TF (master:(task list -> task list -> final_result
                -> (worker_result * task list) list
                -> final_result * task list))
       (worker:task -> (worker_result * task list))
       (workers:no_procs)
       (tasks:task list)
       : final_result

fun master (done:task list)
           (todo:task list)
           (final:final_result)
           (worker_results:(worker_result * task list) list)
           : final_result * task list
```

Figure 5.62: Type-signature for the TF Skeleton

Here the TF function is passed the combine function (here called `master` because the current implementation is a process farm with the combine function implemented on the farmer processor), the sequential task processing function (`worker`), the number of processors (`workers`) and the initial task list. Note that the final result is a list, this allows the use of the empty list as an initial value for final result accumulation.

The parameters to the `master` function are also shown in Figure 5.62. `done` and `todo` are the lists of pending and completed tasks, respectively. These are provided by the skeleton to allow the `master` function to filter duplicate tasks. `final` is the current *partial* result for the overall computation. `worker_results` is the list of values returned by `worker` function applications, including both the worker's partial result and the list of new tasks generated by that application. The `master` function is expected to filter the lists of tasks, returning a new list of pending tasks and accumulating the worker's partial results into a new partial overall result. Termination occurs when there are no pending tasks.

Although the HOF prototype for this skeleton sequentialises the computation into discrete phases the parallel implementation farms out tasks asynchronously and thus has the same load-balancing properties as the DF skeleton.

There are many situations in image processing which fit this model. Growing and tracking algorithms [274, 275, 276, 277] are particularly suitable. Care has to be taken during the implementation, however, since geometric features such as lines can sequentialise the computation. In this case it is possible to use multiple seed-points with a TF skeleton to give a set of partial results which can be merged. The current partial `final_result` value then has to be used in conjunction with pending and completed tasks to prevent duplication of work.

GDIW

This skeleton, like the TF skeleton, attempts to exploit parallelism by manipulating intermediate results. In this case the imtermediate results are work definitions which span geometric boundaries. One optimisation of parallelism, for this skeleton, is to send these work packets directly between cooperating processes thus eliminating redundant communications through a common master processor.

```
type gd = proc * (int * int * int * int)
type phase = int option
type proc = int
type 'a msg = 'a
type 'a result = 'a

fun gdToProc ((x,y):int * int) (gdl:gd list) : gd option

fun GDIW (master:((proc * 'a msg -> unit)
                  -> (proc -> 'a msg option)
                  -> phase -> proc -> gd list -> 'b result option))
         (worker:((proc * 'a msg -> unit)
                  -> (proc -> 'a msg option)
                  -> phase -> proc -> gd -> unit))
         (gd:gd list)
         : 'b result =

fun master (send:proc * msg -> unit)
           (recv:proc -> msg option)
           (phase:int option)
           (mastid:proc)
           (gd:gd list)
           : 'b result option

fun worker (send:proc * msg -> unit)
           (recv:proc -> msg option)
           (phase:int option)
           (mastid:proc)
           (gdl:gd list)
           (gd:gd)
           : unit
```

Figure 5.63: Type-signature for the GDIW Skeleton

Setting up this skeleton requires more programming effort than the other three skeletons since the messages (work packets) are handled explicitly. The skeleton function itself, Figure 5.63, is passed a `master` function which controls the overall computation and a `worker` function which runs the sequential component, generating and handling interworker messages. The geometric decomposition is the same as for the GD skeleton.

The `master` function is passed explicit functions for sending and receiving messages. The messages themselves are handled by the skeleton which is a direct modelling of the parallel equivalent. The `phase` flag is only used by the HOF prototype and is not present in the parallel version. This is primarily to allow the `master` function to detect the first call and perform any initialisation steps. This function also signals termination by returning a `(SOME result)` value to the `GDIW` function (`NONE` implies that the computation continues).

The `worker` function is also passed the message send and receive functions. The `master` "process number" is present to allow addressing of messages to the master processor, the process number for the worker is deduced from the geometric decomposition (`gd`) also passed in. The entire decomposition is given (`gdl`) to allow the use of the `gdToProc` function to relate specific pixels in the image to the processor number handling that pixel.

The `GDIW` function itself operates by successively calling the `master` and `worker` functions until the `master` function terminates the computation.

This skeleton could have many uses in vision applications. Although difficult to setup it has the advantage of optimised communications relative to the other three skeletons presented here. It is present mostly to provide an alternative when none of the other skeletons yields good parallelism due to communications overloading. Although described as a geometric decomposition the type-signature indicates that it has more general uses. The `gd` type could be replaced by a more general work packet definition leading to a static decomposition over a variety of types of data but still retaining the ability to communicate directly between individual, distributed components.

5.4.2 Skeleton Equivalence

It became apparent during system development that there was sometimes more than one way of implementing a particular algorithm using the basic constructs. This situation is complicated by the fact that some of the skeletons are also equivalent to each other in the sense that they can be implemented in terms of each other. Different parallel implementations result from these choices and the problem is to select the combination with the best parallel performance.

The DF skeleton is a generalisation of the GD skeleton, this can be seen in the fact that their types can be unified with the substitution:

```
{'a/gd, 'b/(gd * 'c image)}
```

This takes into account the fact that `gdCombine` has been applied to the output data from the GD skeleton. In fact, the GD skeleton is a DF skeleton in which a decomposition into subimages is suggested by the programmer.

In a similar manner, the TF skeleton is a generalisation of the GDIW skeleton. The TF skeleton can implement the GDIW skeleton by defining a work packet to be a `gd` definition plus a message. Thus, when a message is generated by the GDIW skeleton this is equivalent, in the TF skeleton, to generating a new task with the `gd` definition for the *recipient* processor plus the message itself.

There is also a relationship between the DF and TF skeletons, but they are not directly equivalent. The TF skeleton can be viewed as a DF skeleton in which the complete list of tasks is known in advance. The two are virtually identical in their parallel implementations, both being data farms. The only difference is that the list of pending data is static in the DF skeleton but changes dynamically for the TF skeleton. Another way of viewing this is to say that the TF skeleton's data contains dependencies not present in the DF skeleton's data.

Finally, there is a close correspondence between the parallel implementations of the GD and GDIW skeletons. In this case the only difference is the fact that local messages are passed between the worker processors. The HOFs presented to the programmer are markedly different, however.

Apart from these equivalences it is also possible to achieve the effect of one skeleton with a combination of two skeletons. Consider, for example, the GDIW skeleton which could be implemented as a straight GD phase but collecting the messages followed by subsequent farming out of the collected messages.

In general, decisions about which combination of skeletons to implement lies in the experience and judgement on behalf of the programmer. As an example, consider the connected components algorithm on depth data and edge tracking in intensity data. These are very similar algorithms but handle different types of data. For the depth data a GDIW skeleton was used but the intensity data was handled by the combination of a GD skeleton followed by a separate merge phase implemented using the DF skeleton. The reason for this is the volume of depth data versus intensity data. Parameters such as volume of data can be assessed during the early stages of the algorithm's development where data sizes/communications times could be measured but the final parallel construct had not been decided.

5.5 Algorithm Selection

Here, an attempt is made to justify the particular methods chosen. Note that ease of parallel implementation is one of the major design criteria, as well as more usual factors such as quality of segmentation, sequential processing speed and ease of sequential implementation.

5.5.1 Local Window Operations

For intensity images the Canny edge detector [248] (see Appendix C.2) is probably the best all round edge detection scheme that fits in with the overall system philosophy (i.e. modularising the system as far as possible). Indeed the Canny edge detection methodology is itself expressed as a number of discrete phases; *smoothing*, *gradient detection*, *non-maximal suppression* and *hysteresis thresholding*. A number of edge detection schemes have been studied for parallelisation purposes [240] and, in general, the best implementation is a geometric decomposition.

For depth images there are numerous methods of approaching segmentation of range data, most of them surface-based rather than edge-based [278, 279]. A number of methods use clustering [280, 281, 282] and Hough transforms [258] and some are driven by models [283, 284]. Other methods include random sampling [285] and combined surface and volumetric segmentation [286]. More recently, techniques such as neural networks [287], detecting invariants [288] and probabilistic methods [289] have been used. The most attractive methods from the point of view of this system, however, are those based on surface curvature such as mean (H) and Gaussian (K) curvature based segmentation [43, 269, 290].

Edge-based methods usually have some form of interpolation to close contours prior to segmentation which are not very elegant. Clustering and Hough transform methods tend to have high memory requirements and the target hardware is rather limited in this resource. There is too close a coupling between the model matching and segmentation phases in model-driven methods for separate implementation in a modular manner. By this it is meant that performing segmentation requires data from the model-matching phase which implies a data feedback mechanism which woule be difficult to implement given the modular method proposed in this thesis. Most of the other methods are difficult to evaluate both for segmentation quality and the potential for parallel execution, and so were not considered. This leaves the curvature-based segmentations.

In fact, it is not difficult to envisage symmetric processing of depth and intensity images using most of these methods, for instance, curvature-based segmentation of lines in intensity images is possible and the Hough transform method applies equally well to 2D and 3D data. The combination of preprocessing the image into a homogeneity property such as edge strength or mean and Gaussian curvature, followed by a tracking algorithm offers a good compromise in terms of modularity, resource requirements and the potential for reconfigurability. The depth images are therefore processed using Besl and Jain's HK-based segmentation algorithm [269].

5.5.2 Aggregation According to Homogeneity Measures

Edge tracking[1] as a means of extracting lines from images has been known for a long time [270] but there is little in the literature concerning edge tracking in parallel, this is probably not computationally intensive enough to attract designers of parallel vision systems. The problems of implementing an edge tracking algorithm in parallel have already been investigated [62] and found to give acceptable, if sublinear, performance.

For depth data a connected components algorithm can be used to give a coarse segmentation. Connected components algorithms have been developed for a number of parallel machines [291, 292, 293]. These are often two pass algorithms and are computationally quite expensive. For instance, Sunwoo *et al.* [291] divide the image into subimages and run a labelling algorithm within each subimage followed by a merging of regions across subimages. In some algorithms, however, the merge phase uses geometric information to reduce communications overheads. For instance, Shiloach [294] presents an algorithm that only transmits border pixels while combining subimages. In fact many parallel connected components algorithms operate in a similar manner adapted for various architectures.

A combination of subimage processing with interprocessor communications to reduce communications overheads matches the coarse-grain MIMD hardware and also aligns quite well with the edge tracking method described above. The major difference is that in the edge tracking algorithm the merge phase is executed without interworker communications due to the smaller volumes of data.

5.5.3 Final Segmentation and Feature Determination

For the contours generated by the edge tracking process the best method would be to collect seed regions from within the contour and grow these according to the fitted parameters in much the same manner as the region-growing routines for depth processing. In practice, however, the entire contour is fitted and selected or rejected on the basis of goodness of fit.

Region-growing in parallel has also been studied [295, 276], the methodologies are very similar to those for some connected component labelling algorithms, the idea being to partition an image into disjoint regions. For depth data, the full seed region determination and subsequent growing algorithm was implemented. This was used because of the lack of robustness in the initial curvature-based segmentation. Given a good seed region, region-growing based on residuals tends to remove outliers. Outliers are common in depth images, especially those

[1] Notice that what is called *edge tracking* here, is tracing sequences of pixels within an image under a homogeneity criterion and is not to be confused with *edge tracking* [275] or *contour tracking* [274] which involves following a given contour as it moves around in a sequence of images.

derived by laser triangulation (which is how the data used in this thesis was generated), and particularly near edges and discontinuities.

Overall, there is a high degree of symmetry between the edge tracking method and the region-growing method. There are, however, also implementational discrepancies necessitated by the differences in volume, dimensionality and topology of the data being operated on.

5.6 Summary and Conclusions

In this chapter, the design of a parallel multisource segmentation system was presented. The available parallel hardware was described and skeletons suitable for implementing vision algorithms on this hardware were devised. Finally, vision algorithms were selected on the basis of suitability for both the application and the skeletons available. To summarise the achievements of this chapter:

- The most difficult design decisions concerned the architecture. To implement a variety of different algorithms in parallel on a uniform system requires a fair degree of flexibility and generality. Unfortunately, most parallel processing systems are segregated into a variety of different classes with specific architectural characteristics which make them suitable for one of several parallel processing models. The system here was based on a generic message passing architecture which is considered to be quite flexible.

- Four skeletons were identified during the course of building the system. These were presented, retrospectively, as a set of skeletons suitable for parallel vision implementations, and were geometric decomposition (`GD`), data farming (`DF`), task farming with task spawning (`TF`) and geometric decomposition with interworker communications (`GDIW`). Some of these are equivalent to others and any one algorithm could be implemented using a variety of different combinations of these. No general rules for their application were formed but it was found that geometric decomposition was more useful for low-level algorithms whereas farming was more useful for intermediate data structures such as lists of lines or planes. The `TF` and `GDIW` skeletons present difficulties with SML representation as they include essentially imperative concepts. They also cause problems with performance modelling since distributed communications are harder than simple master/slave arrangements.

- The selection of vision algorithms was made on the basis of commonality of structure and on potential for parallel implementation. This has led to a system which is not state-of-the-art computer vision but which should be suitable for implementation in a parallel system, given the restrictions of hand coding. The employment of more advanced computer vision techniques should not present too many problems. The investigation into the partial classification of vision algorithms in Section 3.4 did not present any cases which could not be handled using the proposed methodology, provided suitable skeletons are available.

Chapter 6

SML Prototype Implementation

6.1 Introduction

This chapter describes the construction of the SML prototype for the proposed parallel object recognition system illustrated in Figure 5.44. The selection of individual algorithms has been described in Section 5.5. In theory, the prototype should be constructed without regard to target architecture considerations (Section 5.2). In practice, the algorithms were originally selected with the target system in mind.

Here, the implementation of each vision module in SML is presented so that the skeletons can be analysed for computation and communications times. The SML representation of the four skeletons described in Section 5.3 is discussed and their instrumentations according to the methods in Chapter 3 are also shown.

6.2 Skeleton Prototypes

It may seem strange to present the SML prototypes for the skeletons after the parallel implementations have been defined, however, the process of designing a new skeleton can be essentially the reverse of that for an application. The skeleton is developed for a specific application on a particular parallel architecture and a suitable sequential prototype has to be developed to represent that skeleton in the prototype application. This is not always the case but for specialised skeletons (such as those used for vision processing in this thesis) this reverse implementation is more likely than for simpler, more general skeletons such as `map` implemented as a processor farm. This is because simple HOFs such as `map` are commonly used within sequential functional languages and are likely to be implemented in parallel as part of an application. Complex patterns of data communication, however, are only considered when implementing an algorithm which requires them, for example the `GDIW` skeleton in this thesis.

In this section, the sequential implementations of the four vision skeletons are presented. In general these are variations on mapping functions over lists but with imperative complications in some cases. The instrumentation of each skeleton is also described, which is generally based upon combining the measurements from the internal `map` components.

6.2.1 Geometric Decomposition (GD)

This skeleton is essentially a functional `map` at its core. The intention here, however, is to provide support for image data. Mapping functions over independent regions of an image is so common in vision processing that it has been wrapped up into a single skeleton.

Figure 6.64 shows the SML prototype for this skeleton. `GeomDecomp` splits the supplied image into subimages according to the number of available processors, the supplied function (`ff`) is then applied over the list of subimages and `gdCombine` reconstructs the original image from the processed subimages. All of these operations are handled by the SML prototype, including complications such as overlap mentioned in Section 5.3.1.

Obtaining a performance prediction for this skeleton is essentially the same as for the Gaussian filter example in Section 4.2. The `map` function at the core of the prototype can be treated exactly as in that example, the only complication being the decomposition and reconstruction of the image. These can be ignored, however, since in the current system the geometric decomposition is precomputed and the recombination is handled by placing returning data directly into the correct portion of the final image. Figure 6.64 shows the timed version of this skeleton which uses the `map_timed` function shown in Figure 4.27. The `ffisize` and `ffosize` functions determine the size of the data and are supplied by the caller to the skeleton. The timing information is dumped into a log file by the `printtimes` function, indexed by the `name` parameter.

6.2.2 Data Farm (DF)

This is the simplest case where the `DF` skeleton can be represented by a simple list `map` function (Figure 6.65). This skeleton can operate on any list of data such as lists of patches or points or even lists of images or subimages. Note that this is a purely functional operation and presents no problems to the SML prototype. As for the GD skeleton, performance prediction is obtained from the `map_timed` function illustrated in Figure 4.27 in conjunction with the

```
(* The Geometric Decomposition skeleton *)
fun GD ff image = gdCombine (map ff (GeomDecomp image))

(* Timed version of GD *)
fun GD_timed name ff ffisize ffosize image =
  let
    val gd = GeomDecomp image
    val (imgs,times,sizes) = TimedSkels.map_timed ff ffisize ffosize gd
    val _ = printtimes (name,times,sizes)
  in
    gdCombine imgs
  end
```

Figure 6.64: The Representation of the Geometric Decomposition (GD) Skeleton

methods of Section 4.2.

```
(* The Data Farm skeleton *)
fun DF ff data = map ff data

(* Timed version of DF *)
fun DF_timed name ff ffisize ffosize data =
  let
    val (result,times,sizes) = TimedSkels.map_timed ff ffisize ffosize data
    val _ = printtimes (name,times,sizes)
  in
    result
  end
```

Figure 6.65: The Representation of the Data Farm (DF) Skeleton

6.2.3 Task Farm (TF)

There is a problem with this skeleton in terms of representing the generated tasks in the source language. These tasks are generated arbitrarily and are essentially imperative in nature which gives problems with a functional prototype. The solution adopted is presented in Figure 6.66. The problem of handling generated tasks is achieved by wrapping an iterative layer around the farming operation (`map`). The list of processed tasks (`done`) is maintained and passed to the `master` function to allow pruning of duplicate tasks. The final results (`final`) are also maintained and passed to the master function for accumulation. The number of processors (`workers`) is used to partition the computation into blocks. This is not necessary for the prototype but should give a slightly more accurate simulation. The order in which the tasks are processed compared to the parallel implementation is altered but the same numbers and types of tasks should be generated allowing summation of communications and computational costs for performance modelling.

Similarly, there is a problem with obtaining a performance prediction under imperative task

```
(* TF:
 *  'a = task, 'b = worker result, 'c = final result
 *  TF (master:'a list -> 'a list -> 'c list -> 'b list -> 'c list * 'a list)
 *     (worker:'a -> 'b)
 *     (workers:int)
 *     (tasks:'a list)
 *)
fun TF master worker workers tasks =
  let
    fun iter [] done final = final
      | iter tasks done final =
      let
        val (htasks,ttasks) =
          if List.length tasks < workers
            then (tasks,[])
            else (List.take (tasks,workers),List.drop (tasks,workers))
        val worker_results = map worker htasks
        val done = htasks@done
        val (nfinal,ntasks) = master done ttasks final worker_results
      in
        iter (ttasks@ntasks) done nfinal
      end
  in
    iter tasks [] []
  end
```

Figure 6.66: The Representation of the Task Farm (TF) Skeleton

generation. This, however, is handled quite simply by treating the unfolded iteration as a single `map` implementation. This is implemented in Figure 6.67, where each iteration is timed individually and the timing results accumulated across iterations. Two sets of sizing functions are supplied, one for the master (`misize`,`mosize`) and one for the worker (`wisize`,`wosize`). Timing results are composed by the `times_oo` function supplied by `TimedSkels` and retained in an imperative list of results (`times`). To allow master and worker times to be combined in the same list the results are unioned by the `datatype ('m,'w) res`. The `apply_timed` function is like `map_timed` but returns `unit`. This method will only work if the tasks are generated in a manner consistent between the prototype and implementation.

6.2.4 Geometric Decomposition with Interworker Communications (GDIW)

This skeleton can really only be implemented using imperative features in the prototype. The problem is modelling the pattern of communications and computation in the final implementation. As mentioned previously, Section 5.3.4, this could be implemented using the task farm (TF) skeleton but with a loss in communications performance. In one vision algorithm (HKSEG), however, it was felt that the performance gain compensated for the additional complexity of the GDIW implementation.

```
(* Timed version of TF *)
fun TF_timed name master worker misize mosize wisize wosize workers tasks =
  let
    open TimedSkels
    datatype ('m,'w) res = M of 'm | W of 'w
    val times = ref ([],[],[])
    fun tupdate times' = times := times_oo (times',!times)
    fun iter [] done final = final
      | iter tasks done final =
      let
        val (htasks,ttasks) =
          if List.length tasks < workers
            then (tasks,[])
            else (List.take (tasks,workers),List.drop (tasks,workers))
        val (worker_results,wtimes,wsizes) =
          map_timed (!rpts) worker wisize wosize htasks
        val _ = tupdate (List.map W worker_results,wtimes,wsizes)
        val done = htasks@done
        val masterff = master done ttasks final
        val (mres,mtimes,msizes) =
          apply_timed masterff misize mosize worker_results
        val _ = tupdate (List.map M mres,mtimes,msizes)
        val (nfinal,ntasks) = hd mres
      in
        iter (ttasks@ntasks) done nfinal
      end
    val result = iter tasks [] []
    val ref (_,times,sizes) = times
    val _ = printtimes (name,times,sizes)
  in
    result
  end
```

Figure 6.67: The Timed Version of the TF Skeleton

The prototype for this skeleton is illustrated in Figure 6.68. This skeleton is implemented in a purely imperative manner. The `gd` parameter is the geometric decomposition provided by the `GeomDecomp` function in Figure 6.64. Each task (processor) is provided with the processor number of the master processor, the whole geometric decomposition definition and functions for sending and receiving messages. The idea is to attempt a simulation of the communications and computation pattern such that it is possible to make reasonable conclusions about the performance of the final implementation.

The basic geometric decomposition is exactly the same as for the GD skeleton. The main difference in the decomposition is that each worker now has to be passed the *entire* decomposition to allow it to relate specific pixels to the processor responsible for those pixels.

A message table is created to store the messages being passed between processors, this is of the size of the decomposition plus one for the master processor and is an array to keep the performance of the prototype from degrading on large numbers of processors. The master processor is called once with a phase flag set to zero, this is to allow the master to initialise internal data upon the first call and to possibly terminate immediately upon bad data. Termination is flagged by the master processor returning the final result from the skeleton although each application of the GDIW skeleton may itself have a complex set of termination conditions built into the master and worker functions.

This simple modelling is not a true reflection of the parallel implementation and could potentially introduce errors into the predictions. Depending on the application, the pattern of computation could be radically altered by changing the message order which in turn can change the pattern of generated messages. What is required here, however, is a rough estimate of the total amount of communications generated by the network as a whole and the total amount of work carried out by an "average" worker. This prototype should allow such estimates provided the application is not overly sensitive to changes in either the computation or communications.

Figure 6.69 shows a method of obtaining timing information from this skeleton. The same method of accumulating times across iterations as for the `TF_timed` skeleton (Figure 6.67) is used. The only additional complication is that the messages passed between processors have to be sized and accumulated separately. The idea is to *linearise* the computation as before but adding the interprocessor messages to the linearised view of the computation. Strictly speaking, the time to process these messages and the size of any return messages should also be measured. This is difficult since the return messages are not sent back to source and may not generate any "reply" messages at all. Given the assumption that interworker messages do not require any time to process and do not generate any further messages then

```
(* GDIW:
 * master/worker (send:proc * msg -> unit)
 *               (recv:proc -> msg option)
 *               (phase:int option)
 *               (mastid:proc)
 *               (gd:proc * (int*int*int*int) * 'a)
 *)
fun GDIW master worker gd =
  let
    val workers = length gd
    val procs = workers+1
    val mastid = workers
    val msgtab = Array.fromList (List.tabulate (procs,fn proc => []))
    fun send ((proc:proc),msg) =
      let
        val proc = if proc >= procs then proc mod procs else proc
      in
        Array.update (msgtab,proc,(Array.sub (msgtab,proc))@[msg])
      end
    fun recv (proc:proc) =
      case Array.sub (msgtab,proc) of
        [] => NONE
      | (h::t) => (Array.update (msgtab,proc,t); SOME h)
    fun run (phase,gd) =
      (List.map (worker send recv (SOME phase) mastid) gd;
       case master send recv (SOME phase) mastid gd of
         SOME result => result
       | NONE => run (phase+1,gd))
  in
    case master send recv (SOME 0) mastid gd of
      SOME result => result
    | NONE => run (1,gd)
  end
```

Figure 6.68: The Geometric Decomposition with Interworker Communications (GDIW) Skeleton

the linearisation can consist of treating these messages as zero-processing messages and can be summarised in the same way as the other `map` messages.

6.3 Vision Algorithm Prototypes

This section describes the SML prototype for the five phases implemented in parallel in Chapter 7. Some of these were written at an early stage in the systems development and use slightly different notation from later code. The intensity route has been well studied in the past and intensity route phases were used as examples in Sections 4.4 and 4.5. The depth route represents the major work in this section, having been implemented from scratch. The depth route is also, for the most part, imperative in nature. A system of referenced variables was developed to mimic the behaviour of the parallel system control flags and some of the major data components are arrays for prototype efficiency. In theory, these could be implemented functionally but would have greatly slowed down the sequential performance of the prototype. Thus, a very close correspondence between prototype and implementation is achieved at the expense of functional purity in the prototype. The standard prototype skeletons were used as sources of parallelism in nearly all cases, however.

6.3.1 Test Image and Parameters

For each phase a speedup prediction is presented for one particular image. This is the image shown in Figure 8.102 and is typical of the images used with the system in this thesis. Table 6.17 gives the global parameters for each phase that were used in the prototype. Each prediction was obtained using the methods in Sections 4.4 and 4.5, the details of the predictions are omitted but some justification as to their validity is given.

6.3.2 Canny Edge Detector (CANNY)

The particular interpretation of the Canny edge detection scheme used here involves a series of convolution operations followed by edge thinning through non-maximal suppression. The calculation details are summarised in Appendix C.2. The first convolution is a simple 1D Gaussian filter smoothing operation but is carried out in the x and y dimensions simultaneously. The smoothed images are then edge enhanced by convolving with a 1D derivative of Gaussian filter in the opposite direction to the smoothing operation for both x and y smoothed images. This gives edge strength components in each direction which are resolved to give an edge magnitude and direction. The final stage is another local window operation whereby the pixel at the centre of a nine-pixel neighbourhood is compared with adjacent

```
(* Timed version of GDIW *)
fun GDIW_timed name master misize mosize worker wisize wosize msgsize gd =
  let
    open TimedSkels
    datatype ('m,'w) res = M of 'm | W of 'w
    val times = ref ([],[],[])
    fun tupdate times' = times := times_oo (times',!times)
    val msgs = ref ([]:(string * int) list)
    val workers = length gd
    val procs = workers+1
    val mastid = workers
    val msgtab = Array.fromList (List.tabulate (procs,fn proc => []))
    fun send ((proc:proc),msg) =
      let
        val proc = if proc >= procs then proc mod procs else proc
        val _ = msgs := (msgsize msg) :: (!msgs)
      in
        Array.update (msgtab,proc,(Array.sub (msgtab,proc))@[msg])
      end
    fun recv (proc:proc) =
      case Array.sub (msgtab,proc) of
        [] => NONE
      | (h::t) => (Array.update (msgtab,proc,t); SOME h)
    fun run (phase,gd) =
      let
        val workerff = worker send recv (SOME phase) mastid
        val (wres,wtimes,wsizes) = map_timed workerff wisize wosize gd
        val _ = tupdate (List.map W wres,wtimes,wsizes)
        val masterff = master send recv (SOME phase) mastid
        val (mres,mtimes,msizes) = apply_timed masterff misize mosize gd
        val _ = tupdate (List.map M mres,mtimes,msizes)
      in
        case mres of
          [SOME result] => result
        | _ => run (phase+1,gd)
      end
    val masterff = master send recv (SOME 0) mastid
    val (mres,mtimes,msizes) = apply_timed masterff misize mosize gd
    val _ = times := (List.map M mres,mtimes,msizes)
    val result =
      case mres of
        [SOME result] => result
      | _ => run (1,gd)
    val ref (_,times,sizes) = times
    val _ = printtimes (name,times,sizes)
  in
    result
  end
```

Figure 6.69: The Timed Version of the GDIW Skeleton

Parameter	Value
`CANNY`	Canny edge detector
N	7
Low	10
High	255
Sigma	1.0
Scale	20.0
`ET`	Edge tracker
EdgeThresh	200
AngleThresh	10
MinLinePts	20
ETportions	25
`HK`	Mean/Gaussian curvature calculation
HKN	7
Hthresh	0.0045
Kthresh	0.00045
Qthresh	1.0
MDSthresh	2.0
`HKSEG`	Coarse HK segmentation
HKSEGN	7
maxDepthDiscrepancy	10.0
minDepth	140.0
maxDepth	10.0
minPatchSize	500
`RG2`	Region-growing
NumSamples	10
SampleSize	100
SampleFactor	4
GrowThreshP	0.5
GrowThreshC	4.0
FitThL	0.1
FitThH	5.0

Table 6.17: Parameters for Prototype Test Image

pixels normal to the edge directional component at the centre pixel and is suppressed if it has an edge magnitude lower than either of its neighbours.

This algorithm has previously been well studied and implemented in SML and Occam2 by Koutsakis [240] and Michaelson and Scaife [61], and subsequently used as a test case for an automated parallelising compiler [296]. Koutsakis [240] used data farming of various decompositions of the image, for instance farming pixels or rows of pixels. This gave good performance up to about 7 processors but farming of these fine-grain data elements proved to be communications bound leading to poor performance for larger numbers of processors. Building on this experience, a geometric decomposition skeleton (`GD`) in conjunction with preloading of image data is used.

The sequential Canny implementation is thus exactly as implemented by Koutsakis [240], modified to fit in with the system here and parallelism consists of applying this sequential Canny implementation to a list of subimages generated by the `GD` skeleton. The only complication with the `GD` skeleton prototype is that the Canny algorithm generates two images from a single input image. The modified `GD2` skeleton was written to perform the first part of the `GD` skeleton but returning a list of pairs of subimages which are then reconstituted into two whole images. Figure 6.70 shows the `canny` function which runs the Canny algorithm on a whole image. The `canny_gd` function takes a geometric decomposition definition (supplied by `GeomDecomp`) and applies the `canny` function to the subimage, the `Img.subimg` function extracts a subimage from a decomposition definition and a whole image. The `runCA` function applies `GD2` to `canny_gd` and saves the resulting images.

The speedup prediction for this phase is shown in Figure 6.71. This shows excellent speedup and is to be expected considering the minimal communications costs (a few bytes per subimage) compared to the computation times (of the order of half a second per subimage).

6.3.3 Edge Tracking (ET)

The edge tracking phase has already been described in detail in Section 4.5. This was an early attempt at an implementation under the described methodology and as such the prototype used the `map` and `filter` HOFs as the sources of parallelism. These could be expressed as `GD` and `DF` skeletons, which is effectively how they were implemented in the parallel version.

6.3.4 Mean and Gaussian Curvature Calculation (HK)

The calculation details for HK are given in Besl [269] and are summarised in Appendix C.1.2. These are simpler than the Canny operation in that the curvatures can be calculated using

```
(* Apply canny to a whole image (subimage) *)
fun canny lower sgm scf img =
  let
    val so = apply (non_max_sup lower scf) (3,3) (smooth_and_find_edges sgm img)
    val (strength,orientation) = unzipll so
    fun r2ilim lo hi r = imax (imin (floor r) hi) lo
    val istr = map (map (r2ilim 0 255)) strength
    val iori = map (map (r2ilim 0 255)) orientation
  in
    (istr,iori)
  end

(* Generate a subimage from decomposition and apply Canny *)
val over = 7
fun canny_gd lower sgm scf (gd as (_,(xmin,xmax,ymin,ymax))) img =
  let
    val _ = print ("canny_gd: "^Sl.st4s Sl.sz (xmin,xmax,ymin,ymax)^"\n")
    val subimg = Img.subimg (xmin,xmax,ymin,ymax) 0 over img
  in
    (gd,canny lower sgm scf subimg)
  end

(* Run the Canny edge detector *)
fun cannyff gd = canny_gd (!CannyLow) (!CannySigma) (!CannyScale) gd (!iFrame)
fun runCA () =
  if !CAflag
    then (print "[Running Canny edge detector...]\n";
          CAimgs := GD2 cannyff (!iGD);
          iFrameSave (!intensityfile) [#1(!CAimgs),#2(!CAimgs)] (!CAout))
    else ()
```

Figure 6.70: The Prototype of the Canny Edge Detector (CANNY)

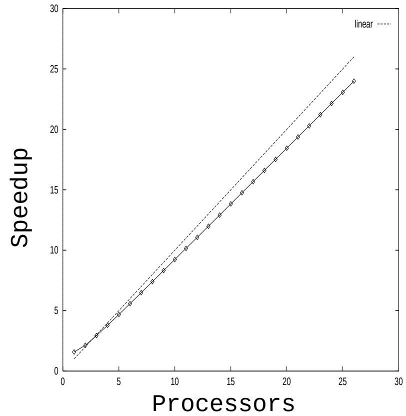


Figure 6.71: Speedup Prediction from CANNY Prototype

a single pass over the data. The first and second-order surface gradients in the x and y directions are calculated by convolving with special sets of normalised polynomials which yield values which are related to the various gradients. These gradients are then combined to give the mean and Gaussian curvatures which are then thresholded to give a sign value (negative, zero or positive) for each curvature. These two sign values are then combined into a single sign code for each pixel. Since all these operations are separable the HK calculation can be carried out independently at each pixel.

This phase is thus essentially a local window operation in the same mould as the Canny edge detector. In fact, it is slightly simpler from the point of view of the `GD` skeleton since the result image can be coded as a single image, despite there being four distinct values returned by the calculation.

The HK calculation is shown in Figure 6.72. The details of the calculation are in Appendix C.1.2. The function `hk` calculates the `H` and `K` values for a rectangular area of the image and the function `hkScan` calculates a single pixel and is scanned over the required area. The results (including `Q` and `MDS`) are encoded in a single integer (`sc`) and saved in a global array (`wFrame`) for later processing. The results of the HK calculation are output in image format by the `output_hk` function. By analogy with the Canny phase the `GD` skeleton applied to subimages is the most likely to give good parallelism. Note that this is an almost exact emulation of the Occam2 code.

The speedup prediction is shown in Figure 6.73 and shows near linear speedup. The tail off at low processor numbers is due to small sample errors. This is almost identical to the Canny example but with slightly lower computation times.

6.3.5 Coarse HK Curvature Segmentation (HKSEG)

The sign map of mean and Gaussian curvature values is processed using a connected components algorithm. This algorithm works on the HK encoded image generated by the previous phase and attempts a coarse segmentation of the image based solely upon the curvature classification.

The raw mean and Gaussian sign values are smoothed using a simple median filter to exclude the large numbers of small areas resulting from noisy data, although there is also a minimum patch size parameter. The central growing algorithm is very simple, each point in the list of current points is checked and neighbouring points which have the same HK sign value are added to the list of current points. Growing then continues until the list of current points is empty. A mask plane indicates which points have been checked. Additional constraints

```
fun hk (gd as (proc,(gdXmin,gdXmax,gdYmin,gdYmax))) =
  let

    (* Announce processing *)
    val _ = print ("hk: "^Sl.st4s Sl.sz (gdXmin,gdXmax,gdYmin,gdYmax)^"\n")

    (* Scan function over given geometric decomposition *)
    fun scanGd ff =
      List.tabulate (gdYmax - gdYmin,
                     fn y => List.tabulate (gdXmax - gdXmin,
                                            fn x => ff (gdXmin + x,gdYmin + y)))
    (* Initialise *)
    val (N,M,b) = HKInit (!HKN)

    (* Calculate H and K at given pixel *)
    fun hkScan (x,y) =
      let
        val (fu,fv,fuu,fuv,fvv,H,K,Q,MDS) = HK (N,M,b) (x,y)
        val sc = HKsigncode(H,K,Q,MDS)
      in
        WorkVal (!wFrame) (0wx7fffffc0,Word.fromInt sc,x,y);
        sc
      end

  in
    (gd, scanGd hkScan)
  end

(* Run the HK calculation *)
fun runHK () =
  if !HKflag
    then (print "[Running HK calculation...]\n";
          HKimg := GD hk (!dGD);
          output_hk (!HKimg) (!HKout))
    else ()
```

Figure 6.72: The Prototype of the Mean and Gaussian Curvature Calculation (HK)

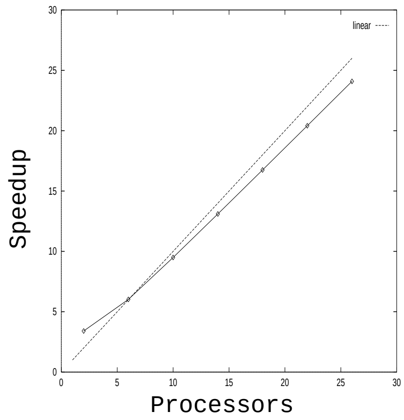


Figure 6.73: Speedup Prediction from HK Prototype

are also placed on the compatibility check, there is a minimum and maximum depth value to exclude baseplane data (background data) and there is also a maximum depth discrepancy which improves the segmentation slightly (i.e. pixels separated by more than this distance are considered to be on different patches).

In this case a `GD` skeleton with interworker communications (`GDIW`) is implemented. This phase was perhaps the most difficult to implement in parallel and it was specifically for this algorithm that the `GDIW` skeleton was developed. The HKSEG algorithm works on the HK encoded image generated by the previous phase and attempts a coarse segmentation of the image based solely upon the curvature classification [297].

Boundary processing is more complicated than for an instance of a `GD` skeleton since a processor reaching a boundary point has to check firstly, which processor is on the other side of the boundary and secondly that a message has not already been received from the processor on the other side of the boundary indicating that it has already reached that point. Timing conflicts between neighbouring processors are minimised by a system of prioritisation of points during growing, all pending boundary points are flushed prior to resuming interior processing. This also minimises idle time once local processing has terminated (all processors have to remain active until all points have been checked in the image).

As indicated in Section 6.2.4 the prototype really needs to be imperative in nature if it has any chance of tracking the pattern of communication and computation. There are several implications of this:

1. Since the worker processors are all "active" simultaneously they need separate variable spaces for their local computation. This problem is addressed in Figure 6.74. The values are stored as an array (`locals`) and are initialised by the master processor on phase 0. When a task runs it calls `setTask` with its processor number. The only values needed are `wFrame` a local work image, `lpatch` a list of `(x,y)` points for growing and `patches` the total number of distinct patches on that processor (not the global number of patches). Initialising the local data upon entry to each processor creates an overhead in the prototype not present in the implementation but is implemented imperatively for speed. Note that this is handled outside the skeleton code. Strictly speaking, this should be handled by the skeleton itself but would have entailed much more complex code with greater overheads.

2. The workers need to be able to address each other directly and, in the case of the GDIW skeleton this means relating processors to geometric areas of the image. This is handled by the GDIW skeleton itself since all workers have access to the full decomposition and

```
val locals = ref (Array.array0 : (word Array2.array ref
                                 * (int * int) list ref
                                 * int ref) array)

fun origWFrame () = DepthData.wFrame
val wFrame = ref (ref (Array2.array (1,1,0w0)))
val lpatch = ref (ref ([] : (int * int) list))
val patches = ref (ref 0)

fun initTask procs =
  locals := Array.fromList
              (List.tabulate
                (procs, fn _ => (ref (array2Copy (!(origWFrame()))),
                                ref [],
                                ref 0)))

fun setTask taskId =
  (wFrame := #1 (Array.sub (!locals,taskId));
   lpatch := #2 (Array.sub (!locals,taskId));
   patches := #3 (Array.sub (!locals,taskId)))
```

Figure 6.74: Handling of Local Variables in the HKSEG Prototype

can work out from that which processors are handling which data (this is the `GeomProc` function in Figure 6.75).

3. All data and tasks have to be indexed to maintain consistency of global data. This indexing is important since it is possible for two processors to grow different sections of the same patch simultaneously. The technique used here is to index each patch fragment with the processor number of the processor which grew it, and maintain a global list of local patch correspondences on the master processor.

   If a worker grows up to a boundary, a message is sent to the processor on the other side of the boundary to continue growing. When a worker finishes growing, the local patch number and processor number are returned to the master with the list of grown points. When a worker receives a boundary point from a neighbouring processor and that point is already allocated to a local patch it sends a message to the master indicating the clash. The master processor is then able to spot when two patches correspond and maintains a conversion table which is applied to relabel the final patch data. The entries in this table have to be transitive.

   This results in a complex pattern of messages which is highly data dependent and could adversely affect the accuracy of the performance modelling.

4. A variety of message types are required. A complex datatype of message types is used during the operation of the prototype which exactly matches the messages used in the parallel implementation. The exact semantics of the message passing system cannot be duplicated (for instance, blocking versus non-blocking, message order preservation) but

the total number and size of generated messages should be a reasonable approximation.

5. Since the GDIW skeleton runs continually the post-processing operations have to be carried out by the master prior to termination (data needs to be uploaded from the workers). This results in a complex phase at the end of the master operation. A system of mode flags is used to determine which post-processing operation is current. Again there is a slight overhead introduced in the interests of maintaining prototype accuracy.

The complete operation of this skeleton is outlined in Figures 6.75, 6.76 and 6.77. These three code segments illustrate the major points in the worker processor that generate messages. Figure 6.75 shows the `chkxy` function which adds a selected pixel to the growing list of patch points and to a global work image which codes the processor and patch number. If the pixel is on the subimage boundary then a message (`HKSegBound`) is sent to the processor on the other side of the boundary to continue growing of the patch over the boundary. The function `GeomProc` gives the processor id of a given pixel in the current decomposition.

```
fun chkxy (send,proc,patchno,x,y) =
  (WorkVal (!(!wFrame))
          (0wx7fe0e0ff,(Word.<< (Word.fromInt proc,0w16))
           orb (Word.<< (Word.fromInt patchno,0w08)),x,y);
   if ((x < !gdXmin) orelse (x >= !gdXmax))
     orelse ((y < !gdYmin) orelse (y >= !gdYmax))
     then
       ((* send physical point to neighbouring processor *)
        case GeomProc(Depth,x,y) of
          SOME gdproc =>
            (if proc <> gdproc
               then send (gdproc,HKSegBound (x,y,proc,patchno))
               else ())
        | NONE => ())
     else ())
```

Figure 6.75: Adding a Point to a Growing List in the HKSEG Prototype

Figure 6.76 shows the function `scan` which is an outer layer for the main region-growing loop. This is required to check for a valid seed point (not already checked and not background data). The function `scanPtmp` is the main recursive growing loop, `Ptmp` is a list of points still to be checked by the growing routine. This routine also keeps track of whether growing hit a subimage boundary or not. Once the patch has grown, generating the `HKSegBound` messages, a decision has to be made as to whether to keep the patch or not. If the patch is larger than a threshold size or is in contact with a boundary then it is retained and registered with the master processor by sending an `HKSegNonBound` message to the master giving the patch number (`patchno`) and the processor id (`proc`), otherwise it is discarded.

Figure 6.77 shows the `seedpoint` function which encodes outer level logic for the growing

```
fun scan (send,mastid,taskId,cx,cy,proc,patchno,mask,minsize) =
  let
    val hkcode = Vote (!HKSEGN,cx,cy)
  in
    if (not (IsWork (!(!wFrame)) (mask,cx,cy,true))) andalso (hkcode <> 4)
      then
        let
          (* found seed point *)
          val (wx,wy,cz) = depthPoint(cx,cy)
        in
          if (cz > !maxDepth) andalso (cz < !minDepth)
            then
              let
                (* install (cx,cy) in Ptmp patch *)
                val Ptmp = [(cx,cy,cz,hkcode)]
                val _ = WorkBit (!(!wFrame)) (mask,1,cx,cy)
                val (boundhit,patch) =
                  scanPtmp (mask,cx,cy,cz,hkcode,Ptmp,false,[])
                val pts = List.length patch
              in
                (* decide whether to keep patch *)
                if not minsize orelse (pts >= !minPatchSize) orelse boundhit
                  then
                    ((* scan patch and place in work image *)
                     List.app (fn (x,y) => chkxy (send,proc,patchno,x,y)) patch;
                     (* send patch ID *)
                     send (mastid,HKSegNonBound (proc,patchno));
                     !patches := !(!patches) + 1)
                  else ()
              end
            else ()
        end
      else ()
  end
```

Figure 6.76: The Growing Routine in the HKSEG Prototype

routine. This is the entry point to the growing routines and is needed to prevent duplication of work. If the seed point has not been visited before it is grown using the `scan` function which will register the new patch with the master. Otherwise this seed point has already been grown and must correspond to another patch. This results in a correspondence message (`HKPatchCorrespond`) being sent to the master to allow the patch index table to be updated.

```
fun seedpoint (send,mastid,taskId,x,y,proc,patchno,mask,minsize) =
  let
    val seedwrk = Array2.sub (!(!wFrame),y,x)
  in
    if seedwrk andb 0wx001f1f00 = 0wx001f1f00
      (* grow from seed *)
      then scan (send,mastid,taskId,x,y,proc,patchno,mask,minsize)
      (* send patch correspondence message *)
      else
        if not minsize
          then
            let
              val thisproc = wtoi ((seedwrk andb 0wx001f0000) shrb 0w16)
              val thispatch = wtoi ((seedwrk andb 0wx00001f00) shrb 0w08)
            in
              if (thisproc <> proc) orelse (thispatch <> patchno)
                then send(mastid,
                          HKPatchCorrespond (thisproc,thispatch,proc,patchno))
                else ()
            end
          else ()
  end
```

Figure 6.77: The Outer Level of the Growing Routine in the HKSEG Prototype

The master processor code is very simple apart from the post-processing phases. Once the parameters are downloaded (including the decomposition) and the computation is triggered the only processing on the master involves reading the patch registration and correspondence messages and updating the index table.

The post-processing operations are:

Patch renumbering. The elements in the index table are mapped onto final patch numbers which are then transmitted to the workers where the grown fragments are renumbered.

Uploading. The patch fragments held locally on the workers are read back onto the master and combined (according to their indices) into contiguous patches.

Downloading. The final patch definitions are dumped back onto the workers for future processing.

The speedup prediction is shown in Figure 6.78. Here the prediction shows greatly sub-optimal speedup but again this is to be expected, the computation times are very low (ranging from 0.007 seconds to 0.1 seconds) compared to the size of the messages (up to 800 bytes for small numbers of processors). Note that the condition for the validity of this prediction mentioned in Section 6.2.4 can be justified for this phase. The patch correspondence messages take very little time to process and so they can be treated as zero-processing messages.

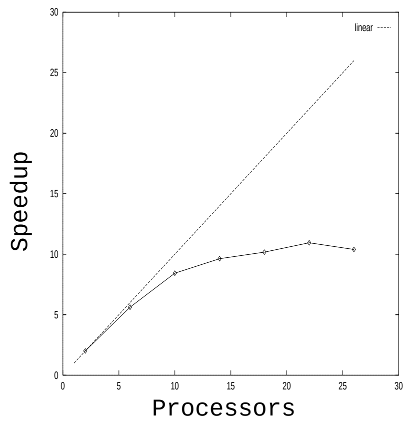


Figure 6.78: Speedup Prediction from HKSEG Prototype

6.3.6 Region-growing from the Coarse Segmentation (RG2)

Once a coarse segmentation based on the signs of mean and Gaussian curvature has been achieved, the next step is to sample the patch and perform region-growing by extending the sample regions. The process of sample selection is crucial to the success of this phase since growing is based on residuals which are prone to being thrown out by outliers. Given a good initial sample and the right threshold for the data, however, region-growing cleans up the data by rejecting outliers.

This phase has many similarities with the HKSEG routine. The main difference is that for RG2 there is a lot more work in the similarity measure. For the HKSEG phase similarity is based on a simple metric read directly from the pre-processed image which has almost no work associated with it at all. For RG2 the similarity measure is based on residuals from patch fits which involves much more computation. It is for this reason that the RG2 phase was implemented using the `TF` skeleton rather than the `GDIW` skeleton.

The RG2 phase is itself split into two phases. In the first, patches are sampled for suitable sets of seed points, the second phase grows the patches from the seeds.

Patch sampling is based on a random seed plus a limited growing within the coarse patch up to the sample size. A preset number of samples are taken from each patch and the sample with the lowest χ^2 value is retained, the patch is then grown if the χ^2 value is less than a threshold value otherwise it is rejected.

This phase is illustrated in Figure 6.79 and is handled by creating a list of seed definitions, i.e. a fixed number of samples from each patch which are then handled with the data farm skeleton `DF` using a simple sequential filter on the output to select the best seeds. The patch data is reset and a list of virtual sample definitions created. There are `NumSamples` to be taken from each patch of which (at most) `SampleFactor` samples are to be taken on a single processor. `RG2GETSAMPLES` is the worker function and grows small samples from a random set of points selected from the patch. The best sample is returned. The `RG2bestsample` function updates the sampled patch with the sample data if the fit is better than the current fit. Note that the `for` function is an imperative "for" loop implemented in SML.

```
fun GetSamples () =
  let
    (* initialise patch data *)
    val _ = print "[Sample patches]\n"
    val _ = InitPatchData ()

    (* Generate list of sample definitions *)
    val samples =
      for (0,!maxpatch,fn p =>
           if Array.sub (!patchhkcode,p) = 0 andalso Array.sub (!Ppcnt,p) > 0
             then List.tabulate (!SampleFactor,
                                 fn _ => (p,
                                          !NumSamples div !SampleFactor,
                                          Array.sub (!patchfnum,p)))
             else [])

    (* Run skeleton, filter best samples *)
    val samples = DF RGGETSAMPLES (flatten samples)
    val _ = List.map RG2bestsample samples
  in
    samples
  end
```

Figure 6.79: The Master Patch Sampling Routine for the RG2 Prototype

The growing process is similar to the connected components algorithm but with some minor complications. The initial seed is used to initialise the mask plane and growing proceeds by the same process of neighbour selection as for the connected components algorithm. In this case, however, neighbouring depth points are chosen if their residuals from the current fit are less than a global threshold. If the residual is greater than the threshold then it is retained in a separate list of rejected points. The reason for this is that since the initial fit cannot be guaranteed to be exact relative to the rest of the patch a policy of refitting after the patch size increases by a preset number of points is implemented. When a refit is performed, the list of rejected points is then re-examined and if any are within tolerance of the new fit they are added to the growing patch.

The `FarmDivisions` function is the top level call to the growing routines on the master processor and is depicted in Figure 6.80. The main problem is in generating the task lists.

A list of tasks are generated from the sampled patches, initially there will be very few tasks, possibly only one. The function `pointstodivisions` splits points lists up into geometric areas (in fact the same geometric decomposition as before but much finer). A task is then defined as a decomposition definition (`xyd`), the patch number, the fit coefficients and the list of seed points. Thus, `todo` is the initial list of tasks. The task farm skeleton is run on the initial task list resulting in a list of grown patches. These are processed by `procPatch` which updates the patch fit and installs the grown data in the global patch data. Finally, the resulting points lists are refitted and added to the imperative patch data, the `ReconstructPatches` function renders the patch data in an intensity image for visual output. Note that `ffor` is an imperative "for" loop allowing accumulation of data.

```
fun FarmDivisions () =
  let
    val _ = print "[Farm divisions]\n"
    val todo =
      ffor (0,!maxpatch,
        (fn (p,todo) =>
         let
           val patchpts = Array.sub (!patchpts,p)
           val patchfnum = Array.sub (!patchfnum,p)
           val chisqpp = (Array.sub(!patchchisq,p)) / Real.fromInt(patchpts)
         in
           if (Array.sub(!patchhkcode,p) = 0) andalso (patchpts > 0)
             andalso (patchfnum = 0) andalso (chisqpp <= !FitThL)
             then
               let
                 val xypts = Array.foldr (op ::) [] (!(Array.sub (!xypatch,p)))
                 val divs = pointstodivisions xypts
                 fun maketask (xyd,xydivpts) =
                   (xyd,p,!(Array.sub(!coeffs,p)),xydivpts)
               in
                 (map maketask divs)@todo
               end
             else todo
         end),[])

    val patches = TF stask sresult master worker (!dprocs) todo

    val _ = List.map procPatch patches
  in
    ReconstructPatches patches
  end
```

Figure 6.80: The Master Region-growing Routine for the RG2 Prototype

The operation of the skeleton itself is relatively simple. Figure 6.81 illustrates the `master` process. This function is only required to update the global patch data and remove duplicates. This is handled by the information in the task definition and the list of pending and completed tasks passed by the TF skeleton. These can be in either the tasks generated by other workers in the current iteration (`tasks`), the current list of pending tasks (`todo_tasks`) or in the list of completed tasks (`done_tasks`). The post-processing in this case is also simple, the patch fragments are combined into whole patches and the patches are rendered into an intensity image.

```
fun updatePatch done_tasks todo_tasks
                (((patch,xygrow,a,ma,chisq),wtasks,xydiv),(finals,tasks)) =
  let
    val _ = updateFit patch a (length xygrow)
    fun notIn wtask tasks =
      not (isSome (List.find (fn d => taskEq (d,wtask)) tasks))
    fun notDone wtask = notIn wtask (done_tasks @ tasks @ todo_tasks)
    val ytasks = List.filter notDone wtasks
  in
    (addpatch finals,tasks@ytasks)
  end

fun master done_tasks todo_tasks finals worker_list =
  List.foldr (updatePatch done_tasks todo_tasks) (finals,[]) worker_list
```

Figure 6.81: The Master Function for the RG2 Prototype

The `worker` process is presented in Figure 6.82. This grows the data from the patch sample within the boundary of the subimage, maintaining a list of points which lie on the boundary. The work image is initialised with null values (`GdInitMask`) and the patch data installed (`SetWorkPatch`). The main growing loop (`GrowRegion2`) returns the new fit data plus the grown points (`divs`) which are turned into new tasks. The boundary data is then split up into new tasks and returned with the grown points lists. The only complication is that the fit has to be periodically updated to match the grown points. Local refitting is not a problem within a process but global consistency needs to be maintained. This is handled by noting that the fit functions are linear in their coefficients and the fits can be updated by a simple linear combination. This is why the fit coefficients are included in the task definitions since the coefficients returned by processing a task are used to update the global fit and, retrospectively, the coefficients in any pending tasks.

```
fun worker (task as (xydiv,patch,coeffs,xybest)) =
  let
    val (xmin,xmax,ymin,ymax) = divtogdxy xydiv
    val _ = (gdXmin := xmin; gdXmax := xmax; gdYmin := ymin; gdYmax := ymax)
    val _ = GdInitMask (!wFrame)
                       (0wx7fffffff,0wx60001f00,0wx1fffffff,0wx00000000,
                        !gdXmin,!gdYmin,!gdXmax,!gdYmax)
    val _ = SetWorkPatch (patch,xybest)
    val (res,divs) = GrowRegion2 (patch,xybest,true)
    fun maketask (xyd,xydivpts) = (xyd,patch,!a,xydivpts)
    val tasks = map maketask divs
  in
    (res,tasks,xydiv)
  end
```

Figure 6.82: The Worker Function for the RG2 Prototype

The speedup prediction is shown in Figure 6.83. Again, the speedup profile is very good. No prediction for the `GetSamples` phase was generated, the overall region growing algorithm is dominated by the `FarmDivisions` component.

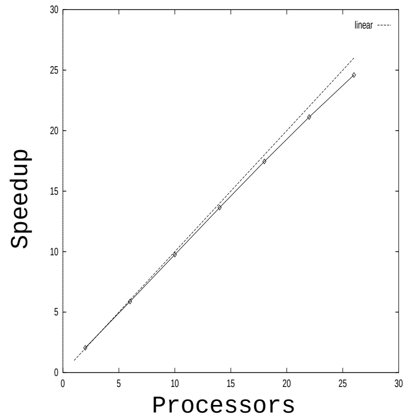


Figure 6.83: Speedup Prediction from RG2 Prototype (FarmDivisions)

6.4 Summary and Conclusions

In this chapter, the construction of a complete and fully functional multisource segmentation system in SML was shown. A close correspondence between the prototype and target systems was obtained but the placement of skeletons within the code was developed and maintained by hand which limited the degree of experimentation that could be carried out on the prototype. To summarise the achievements of this chapter:

- The four vision skeletons were successfully represented in SML in such a way that the characteristics of the parallel implementation could be modelled. The DF and GD skeletons were implementations of functional `maps`. The TF and GDIW skeletons both have imperative features and this reduces the accuracy of the modelling is in these cases. It may be possible to refine the analytical techniques to work from pure functional skeletons but this is a complex problem outside the scope of this thesis.

- Each of the five selected vision algorithms were implemented in SML using the prototypes of the pregiven skeletons. One module (ET) used different notation but could have been expressed using the standard set. The CANNY, ET and HK phases were all implemented with the functional GD and DF skeletons and presented no problems for prototyping. The HKSEG phase used the GDIW skeleton and required a great deal of extra coding to get the prototype to match up with the intended parallel implementation. The RG2 phase used the TF skeleton and required very little additional code but is possibly less efficient than an implementation based on the GDIW skeleton. In both cases, a close correspondence with the final implementation was achieved.

- For each vision algorithm a speedup profile was generated by taking the instrumentation data from the timed versions of the skeletons and plugging this into polynomials generated by running the dummy farm construct on the transputer system. In some cases near linear speedup is predicted (CANNY, HK, RG2), in others highly suboptimal speedup is predicted (ET, HKSEG). These speedup profiles can only be considered a rough approximation to the actual behaviour of the parallel system.

- It should be noted that the prototype lacks much of the support code present in the parallel implementation, the prototype only contains the core algorithms targetted for parallel implementation. Much of the data monitoring and post-processing functions are lacking but would only be needed if the prototype were subsequently to be used in linking together a complete system with other prototypes, for instance the matching phase which the output from the system presented here feeds into.

- The prototype phases were linked together with a minimum of linkage code. The control system in the parallel version is supported by a system of reference variables. This is at odds with the final implementation which required much more effort to get the phases to mesh with one another. This is partly because the prototypes are all sequential and some "shortcuts" could be taken in the prototype but not in the parallel version and partly because the prototype does not feed into a final matching phase as the parallel system does. The purpose of the prototype was to study the parallel phases, not the overall system, however.

Chapter 7

Parallel System Implementation

7.1 Introduction

This chapter describes the construction of the complete multisource segmentation system in parallel Occam2 illustrated in Figure 5.44 and based closely on the prototypes presented in Chapter 6. The parallel system is composed of five discrete modules implemented in parallel with suitable sequential "glue" to bind them together so the individual modules and the linking code are described separately.

The process of developing a parallel implementation from an SML prototype is relatively automatic once the basic mode of parallelism (skeleton) is decided. The bulk of the effort goes into instantiating the skeleton with code which has been transformed from the prototype SML into native Occam2, this was discussed in Section 3.6 and is a relatively mechanical process. The skeletons described in Sections 5.3 and 6.2 have been implemented in Occam2 and these are the basis for parallelism in the Occam2 implementation.

7.2 Parallel Algorithm Implementation

This section discusses the construction of the selected parallel phases. The basic system architecture is described in Section 5.2 and the functions of the individual modules have already been described in Section 5.5. Here the implementation of each module is discussed based on the prototypes presented in Section 6.3 and the instantiation of each skeleton is described. Typical parallel performance for each module under favourable conditions is given and compared with the predictions obtained in Section 6.3 but the overall behaviour of the system is left to Chapter 8. The performance results given here are for one specific image, see Figure 8.102.

7.2.1 Parallel System Environment

The system to be implemented is highly complex in terms of the control and synchronisation of processes and the placement and transmission of data. Since developing and debugging parallel systems is a difficult process, some basic defensive programming principles were adopted during system construction:

1. All messages are tagged with labels which relate them to their intended destination. The tags themselves are unnecessary and are a system overhead but stray messages can be more easily identified.

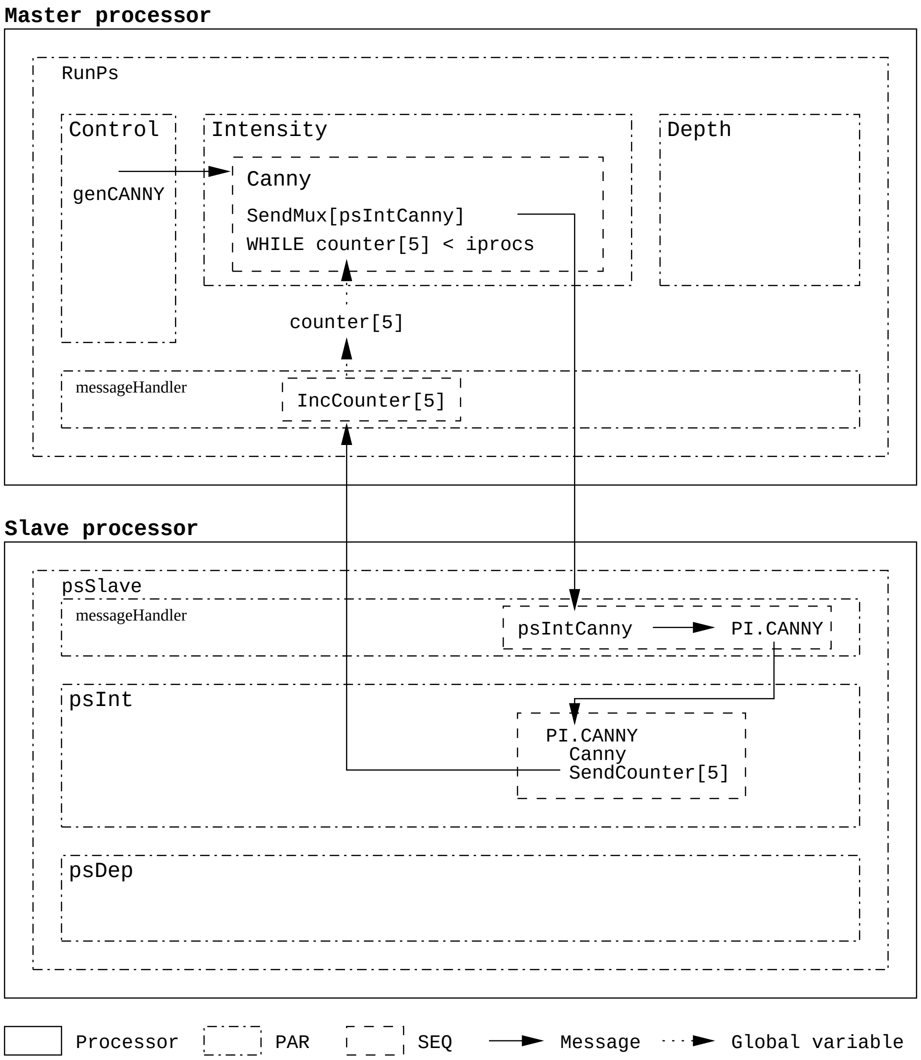
2. Similarly, there is only one message reception point on all processors, again reducing the possibility of messages being received by the wrong process. This may cause higher latencies in message send/receive times but is again worthwhile for robustness.

3. As much code as possible is made generic and kept in libraries. This is not easy to arrange under Occam2 but minimises the amount of debug effort.

4. Since a worker cannot be concurrently a depth processor and an intensity processor, the code for these two are physically separate processes although they share a common memory area. This also applies to the master processor but in this case the two can run concurrently.

Figure 7.84 illustrates the operation of some of these principles for the Canny edge detector. The `Control` process on the master handles initialisation and basic input/output and triggers the various modules. This sends a `genCANNY` message through a dedicated channel to the `Intensity` process which runs the master code for the parallel Canny edge detector. This process broadcasts any phase parameters and then sends a `psIntCanny` message through the output multiplexor to the workers and then waits for the termination messages. The `psIntCanny` message is received by the `messageHandler` process on the worker processor. This then passes the message on as a `PI.CANNY` message to the `psInt` process which runs the sequential Canny algorithm on its segment of the image. Once processing of the subimage is complete the `SendCounter` library function generates an `IncCounter` message which is passed through the output multiplexor to the `messageHandler` process on the master. This simply increments a global variable (`counter`) and the `Canny` process terminates when this counter reaches the number of intensity processors.

Note that the image data had already been broadcast to the workers and that the geometric decomposition definition was also precalculated and transmitted to the workers. Handling of this type of data upload/download is carried out by the multiplexors themselves using



The relationship between the various software components is illustrated for the Canny edge detector. Note that there are other layers of code around the objects shown here, for instance the Rian and Tiny harnesses, but these are omitted for clarity.

Figure 7.84: Illustration of Software Components

library functions. For example, segments of images are sent from master to worker and back again at several points in the program so there is a central library consisting of the various components required to make this work on any processor. Termination messages are also handled by generic code implemented in libraries. For this purpose a system of counters has been implemented such that any processor can increment a counter on any other processor, invisibly.

One point about the speedup measurements presented in this thesis is that the normal speedup calculation:

$$speedup = \frac{time\ for\ 1\ processor}{time\ for\ P\ processors} \tag{7.37}$$

is implemented online by estimating $time\ for\ 1\ processor$ by summing the total amount of work carried out by the workers during the actual calculation. This gives a slight overestimate of speedup because the communications times for a single parallel processor are not included. This has been verified to be quite close to the actual speedup for light communications loads. The main advantage of this is that it helps system development by giving an instant speedup estimate each time the system runs which can be used to "tune in" the performance without having to assume a value for sequential runtime. Another advantage is that, due to memory limitations, it was not possible to run on one processor for some phases but this method allows a close approximation to the time for a single processor run.

7.2.2 Canny Edge Detector (CANNY)

This is a straightforward implementation of the modified Geometric Decomposition skeleton `GD2` (Figure 6.70), Figure 7.85 shows the highest level Occam2 in the final implementation. Preloading is implemented for efficiency reasons and is handled by a single broadcast when the parallel system initialises. This also eases the communications burden since processors require data from outside their processing region as the window operations on boundary pixels need data from outside the local subimage. Note that preloading is also likely to be more efficient than sending only local data to each worker, an optimised broadcast scheme can be used to send the image data to all workers whereas sending local data requires individual messages for each processor. Note that all data upload and download operations along with timing measurements and terminations messages are handled by generic code built into the multiplexors for the respective processors. The Canny calculation is simply triggered by sending a start message (`psIntCANNY`) to the worker, this is read by the worker's input multiplexor and forwarded to the intensity process as a `PI.CANNY` message. Note also that all worker functions operate in this manner, each half of the workers' main tasks consists of a single `CASE` statement indexed by these constants (`PI.XXX` for intensity, `PD.XXX` for depth). Termination is flagged by the `SendCounter` routine which implements a system of numbered counters on all processors.

```
-- Master code
{{{  generate CANNY edge map
        cmsg.genCANNY
          SEQ
            ...  transmit CANNY parameters
            {{{  trigger CANNY calculation
            counter[5] := 0
            TxBuf[0] := BYTE psIntCANNY
            SEQ p = iprocstart FOR iprocs
              SendMux(2,p,[TxBuf FROM 0 FOR 1])
            }}}
            {{{  wait for CANNY termination messages
            WHILE counter[5] < iprocs
              Delay(100)
            }}}
            ...  redistribute CANNY patch data to all workers
}}}

-- Worker code
WHILE psIntRun
  SEQ
    fromPs ? CASE
      PI.CANNY
        SEQ
          CalcCannyFilters()
          Canny((2 * imax(imin(n - 1,CannyN / 2),1)) + 1)
          SendCounter(2,mastid,5,1)
```

Figure 7.85: The Occam2 Implementation of the CANNY Phase

Each worker is sent a rectangular area definition to define its area of processing. Since the Canny algorithm is applied sequentially to the local data, the skeleton instantiation function simply takes a rectangular portion of the image and runs the complete edge detection scheme on that subimage. The translation of the sequential SML into sequential Occam2 for this algorithm proved to be relatively simple. Table 7.18 shows the other parameters required to instantiate the skeleton. The resulting edge detected image is retained distributed but can be uploaded from the workers as a separate operation, if required.

Figure 7.86 shows a typical speedup profile for 2D meshes of processors. The speedup profile is almost linear up to the 28 processors available and agrees well with the prediction from the prototype. Considering the simple decomposition and the fact that image download and upload are not included in the calculation one would expect linear speedup from this phase.

Global parameters	N	Size of window
	Low	Intensity lower threshold
	High	Intensity upper threshold
	Sigma	Gaussian filter sigma
	Scale	Output scale
Skeleton	GD	Geometric decomposition
Global fixed data	gD	Geometric decomposition
	iFrame	Raw intensity data
Instantiation	Canny	Canny algorithm within subimage
Input	gDnum	Subimage number
Output	mFrame	Edge magnitude subimage
	aFrame	Edge angular subimage

Table 7.18: Canny Phase Parameterisation

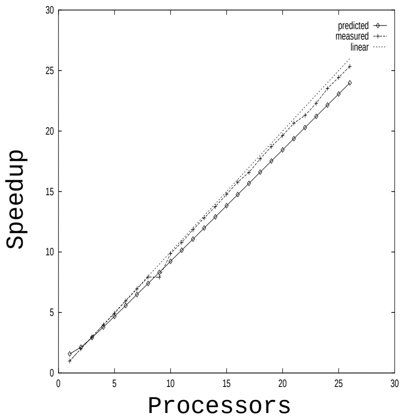


Figure 7.86: Canny Phase Parallel Performance

7.2.3 Edge Tracking (ET)

The edge tracking algorithm was discussed in depth in Section 4.5. The top-level Occam2 fold is shown in Figure 7.87. The first phase is a growing phase within a geometric decomposition, this is implemented directly as a farm using the predefined geometric decomposition. The `MergeLines` function implements the merging phase as a data farm where the farmed data is generated *in situ* from all pairs of line numbers. The post processing phase involves running Scher's algorithm on each line and is implemented in parallel (one processor per line). For comparison with the other modules in this section Figure 7.88 illustrates the basic scheme, Table 7.19 shows the parameterisation for the two phases and Figure 7.89 shows the performance achieved for 25 subimages.

```
-- Master code
{{{  edge tracking
        cmsg.trackET
          SEQ
            ...  transmit ET parameters
            {{{  run ET farm
            SEQ
            ...  geometric decomposition
            ...  initialise
              {{{  run farm
              SEQ por = 0 FOR ETportions
                SEQ
                  ...  find free processor
                  ...  send track command
              WHILE freeprocs(iprocstart,iprocs) < iprocs
                SKIP
              }}}
            }}}
            {{{  merge ET lines
            ...  PROC MergeLines([]INT sl,INT slcnt)
            SEQ
              TxLineData()
              SEQ l = 0 FOR maxline
                sl[l] := l
              slcnt := maxline
              MergeLines(sl,slcnt)
              ...  zap short lines
            }}}
        ...  perform ET post processing
        ...  write detected lines to result file
}}}
%        ...  check for ellipses
```

Figure 7.87: The Occam2 Implementation of the ET Phase

7.2.4 Post Processing of Line Data (POST)

The lines output by the edge tracking routine are fitted and rejected if the fit quality is worse than a threshold. Lines which pass this phase are scanned and segmented further using

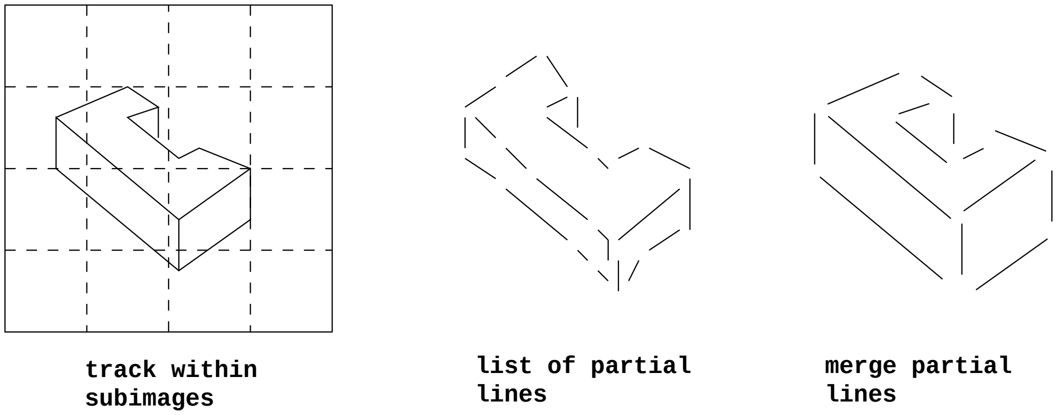


Figure 7.88: Illustration of the Edge Tracking Process

Global parameters	EdgeThresh AngleThresh MinLinePts	Edge magnitude threshold Angular threshold Minimum number of points per line
Phase 1: Edge tracking		
Skeleton	DF	Data farming
Global fixed data	mFrame aFrame	Edge magnitude image Edge angular image
Instantiation Input Output	EdgeTrack siDef linexypts	Edge tracking within subimage Subimage definition (x/y min/max) List of partial lines
Phase 2: Partial line merge		
Skeleton	DF	Data farming
Global fixed data	linexypts	List of partial lines
Instantiation Input Output	MergeLines l1,l2 merge	Line merge predicate Pair of line numbers Merge lines boolean

Table 7.19: Edge Trackng Parameterisation

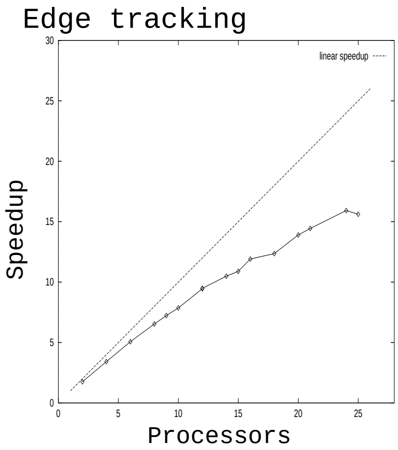


Figure 7.89: Speedup for Edge Tracking Implementation on a 2D Mesh

Global parameters	HKN HKthresh	Size of window HK sign threshold
Skeleton	GD	Geometric decomposition
Global fixed data	gD dFrame	Geometric decomposition Raw depth data
Instantiation Input Output	HKcalc gDnum hkFrame	Calculate HK signs within subimage Subimage number HK curvature sign subimage

Table 7.20: HK Calculation Parameterisation

Scher's algorithm [298]. Line acceptance/rejection is a sequential overhead to the system but the implementation of Scher's algorithm is parallelised inasmuch as each line is tracked on a separate processor, this operation is tacked onto the end of the edge tracking phase (Figure 7.87).

7.2.5 Mean and Gaussian Curvature Calculation (HK)

This module was implemented using the same method as the Canny edge detector. Preloading of the depth data was followed by a geometric decomposition (GD) with global updating of the resulting values. Again, the sequential Occam2 is a direct translation of the SML prototype (Figure 6.72), and is almost identical to the Occam2 for the Canny edge detector (Figure 7.85).

The implementation is straightforward, Table 7.20 shows the parameterisation of this skeleton and Figure 7.90 shows the speedup achieved for a typical image with suitable parameters for that image. This agrees well with the prediction from the prototype.

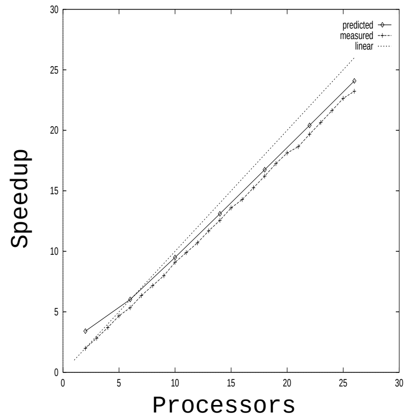


Figure 7.90: Speedup for HK Calculation Implementation

7.2.6 Coarse HK Curvature Segmentation (HKSEG)

The parallel implementation of this phase is exactly as modelled by the prototype in Section 6.3.5. Figure 7.91 presents the top-level control of this phase in the Occam2 implementation. Running the `GDIW` skeleton requires very little additional control apart from the more complicated termination conditions, each worker remains active, processing any messages from neighbours until a termination flag is received from the master. The interworker messages are handled by the input multiplexor on each worker, this process communicates with the code here through global variables. Patch renumbering is done in parallel, the renumbering table is simply transmitted to each worker prior to upload of the result image. There is a final scrub of small patches after renumbering (this is necessary because small patches can span geometric boundaries) which is an additional sequential overhead.

```
-- Master code
{{{  coarse HKSEG segmentation
        cmsg.HKseg
          SEQ
          ...  transmit HKSEG parameters
          {{{  trigger HKSEG calculation
            counter[4] := 0
            TxBuf[0] := psDepHKSEG
            SEQ p = dprocstart FOR dprocs
              SendMux(1,p,[TxBuf FROM 0 FOR 1])
          }}}
          {{{  wait for HKSEG termination messages
            WHILE counter[4] < dprocs
              Delay(100)
            SEQ p = dprocstart FOR dprocs
              SendCounter(1,p,4,1)
          }}}
          ...  renumber patch labels to be consecutive
          ...  send HKSEG patch labels
          ...  send HKSEG work data request packets
          ...  redistribute HKSEG patch data to all workers
          ...  scrub HKSEG small patches
}}}

-- Worker code
    {{{  PD.HKSEG
              PD.HKSEG
                SEQ
                  ...  reset work bits
                  ...  scan boundary points first
                  ...  scan interior
                  {{{  wait for neighbours to finish
                  SendCounter(1,mastid,4,1)
                  WHILE counter[4] = 0
                    flushbound()
                  }}}
    }}}
```

Figure 7.91: The Occam2 Implementation of the HKSEG Phase

Table 7.21 gives some details of the skeleton instantiation, in this case two functions have

Global parameters	HKSEGN	Size of median filter window
	MaxDepthDisc	Maximum depth discrepancy
	minDepth	Minimum depth for real data
	maxDepth	Maximum depth for real data
	minPatchSize	Minimum patch size
Skeleton	GDIW	GD with interworker communications
Global fixed data	gD	Geometric decomposition
	hkFrame	HK curvature sign subimage
Instantiation	HKseg	HK segmentation within subimage
Boundary	flushbound	Process boundary messages
Input	gDnum	Subimage number
Output	proc,patch	Processor/patch identity
	proc,patch,x,y	Grow over boundary message

Table 7.21: HK Segmentation Parameterisation

to be provided, one to perform the region-growing (`HKseg`), the other to handle incoming messages from neighbouring workers (`flushbound`). Both of these are implemented in terms of the `seedpoint` function in Figure 6.77.

Figure 7.92 shows a schematic of this process. Upon completion of the algorithm the data is distributed among the workers, there then follows a tidy up operation where the master renumbers the patches based on the local labels and sends the labels to the workers to update their local data with the new labels. This is followed by an upload of patch data giving correctly numbered patches on the master.

Figure 7.93 shows the parallel performance achieved using this method which is less than that achieved for the other skeletons. The schematic highlights the reason for this. There is a potential inefficiency associated with geometric decomposition in this context. Not all data points represent patches which leads to an imbalance of data between processing elements in a geometric decomposition. This could easily be cured, however, by the provision of a mask plane which defines processor/data point allocations. Before the main algorithm commences, the actual data in the image could be divided equally between workers, defined by this mask plane, which would be transmitted prior to running the main algorithm. Since the algorithm is based on processing of iconic data, only minor changes would be required to fit in the allocation plane. This was not implemented, however, due to memory limitations.

The prediction from the prototype has roughly the same profile but deviates in a number of areas. This is probably due to inaccuracies in the modelling mentioned in Section 6.2.4.

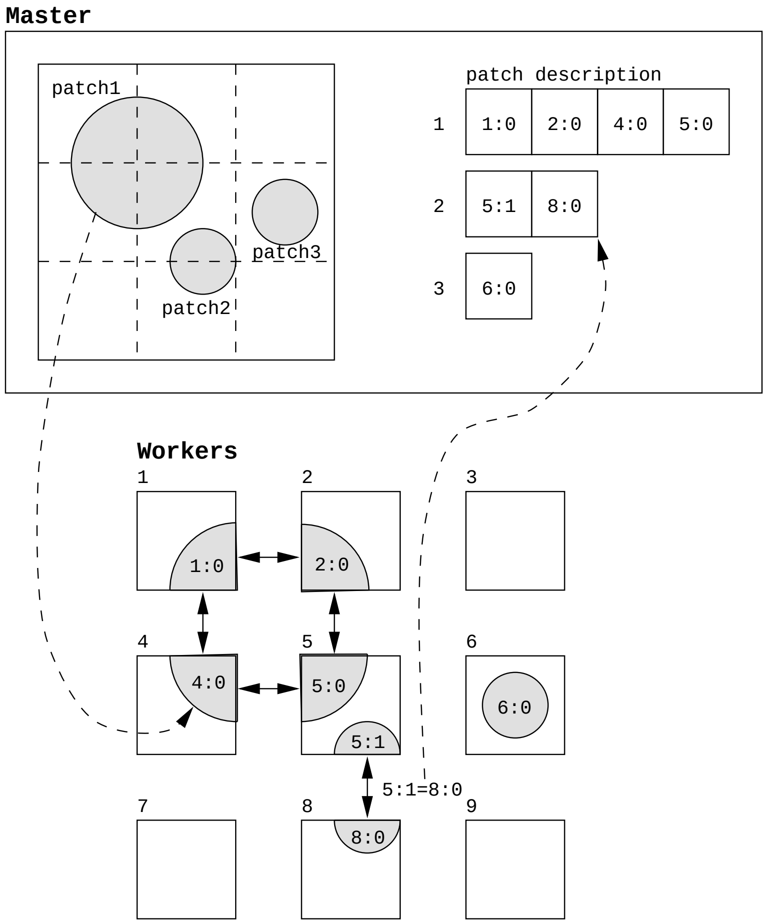


Figure 7.92: Schematic of HK Segmentation Algorithm

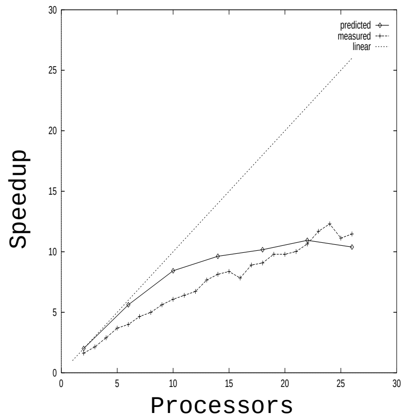


Figure 7.93: Speedup for HK Segmentation Implementation

7.2.7 Region-growing from the Coarse Segmentation (RG2)

Again, this is a direct implementation of the SML prototype described in Section 6.3.6, with the seeding phase and the growing phase both implemented in parallel. The seeding phase is implemented in parallel using the data farming skeleton (`DF`) such that the total number of samples required for each patch is divided up into chunks to give control over the granularity. The master keeps track of the best fit for each patch. Region-growing is implemented in parallel using the task farming with task spawning skeleton (`TF`). Here, the initial seed regions are treated as tasks and a fine-grain geometric decomposition of the image is used to define task boundaries. Region-growing is performed within these subimages and when growing crosses a geometric boundary a list of external seed points is maintained. Upon completion within the subimage, these points are partitioned according to their subimage number and new tasks are formed from the combination of these points plus their subimage number which are returned to the monitor for queuing. Again there is an optimal granularity for this process, too fine and communications dominates, too coarse and there is not enough parallelism.

Figure 7.94 shows the top-level Occam2 fold for the master processor. The task farm is mostly managed in the multiplexors, the global value `todocnt` gives the total number of pending tasks. When a new task is returned from a worker it is added to the pending task list by the master's input multiplexor. Apart from this, the task farm operates very much like a data farm. Note that the `RefitPatches` function is run in parallel (one patch per processor).

Figure 7.95 illustrates the top-level Occam2 fold for the worker processors. Here, the `RGGETSAMPLES` code is a simplified growing routine seeded with random patch points. The `GrowRegion2` participates in the task farming implementation, the "split up boundary" fold takes the grown points which intersect a subimage boundary, generates new tasks from this data and sends the tasks back to the master processor. Termination is very much like the `GDIW` implementation in Figure 7.91.

Table 7.22 shows the parameterisation for both phases. Figure 7.96 illustrates the basic process and Figure 7.97 shows the speedup achieved for a suitable level of granularity, (the image is divided up into 30×30 rectangular subimages). This agrees well with the prediction from the prototype.

7.3 Building the Complete System

Section 7.2 described the implementation of the major parallel components of the segmentation system. This section discusses how these components were combined to form a complete

```
-- Master code
{{{  single pass RG2 region-growing
  cmsg.initRG2
    SEQ
      ...  transmit RG2 parameters
      {{{  get RG2 patch samples
      ...  initialise
      SEQ p = 0 FOR maxpatch
        IF
          (patchhkcode[p] = 0) AND (Ppcnt[p] > 0)
            SEQ f = 0 FOR SampleFactor
              ...  farm out one value
          TRUE
            SKIP
      WHILE freeprocs(dprocstart,dprocs) < dprocs
        SKIP
      }}}
      {{{  farm out RG2 divisions
      todocnt,maxtodocnt,donecnt := 0,0,0
      ...  scan seed regions, create task list
      ...  reset patches
      WHILE (todocnt > 0) OR (freeprocs(dprocstart,dprocs) < dprocs)
        SEQ
          IF
            todocnt > 0
              {{{  farm out one value
              SEQ
                ...  grab free proc
                TxBuf[0] := psDepRG2GrowDivision
                ...  install division, patch
                ...  install fit parameters
                ...  install seed points
                SendMux(1,dp,[TxBuf FROM 0 FOR base + (todopts * 4)])
              }}}
            TRUE
              SKIP
      RefitPatches()
      }}}
      ...  calculate normals and centroids
}}}
```

Figure 7.94: The Occam2 Implementation of the RG2 Phase Master

```
-- Worker code
{{{  PD.RGGETSAMPLES
  PD.RGGETSAMPLES;patch;samples;fit
    SEQ
      SetupFit(fit)
      bestchisq,bestchisqpp := BIG64,BIG64
      GetPatchPoints(patch,pts,xpts,ypts,sig)
      WorkBitRect(0,0,wCols,wRows,#7FFFFFFF,#00000000)
      BestSample(patch,samples)
      ...  send result to master
}}}
{{{  PD.RG2GROWDIVISION
  PD.RG2GROWDIVISION;xdiv;ydiv;patch
  {{{  PROC GrowRegion2(patch,xbest,ybest,sigbest,bestspts,refit)
    PROC GrowRegion2(VAL INT patch,
                     []INT xbest,ybest,[]REAL64 sigbest,INT bestspts,
                     VAL BOOL refit)
      SEQ
        SetupFit(patchfnum[patch])
        ...  initialise point lists
        ...  run growing algorithm
        ...  split up boundary, send to master
        ...  update patch fit
        ...  send grown points to master
        ...  send termination message
    :
  }}}
    SEQ
      a := coeffs64[patch]
      gdXmin,gdXmax,gdYmin,gdYmax := divtogdxy(xdiv,ydiv)
      GdInitMask(#FFFFFFFF,#C0001F00,#3FFFFFFF,#00000000,
                 gdXmin,gdYmin,gdXmax,gdYmax)
      SetWorkPatch(patch,xbest,ybest,bestspts)
      GrowRegion2(patch,xbest,ybest,sigbest,bestspts,TRUE)
}}}
```

Figure 7.95: The Occam2 Implementation of the RG2 Phase Worker

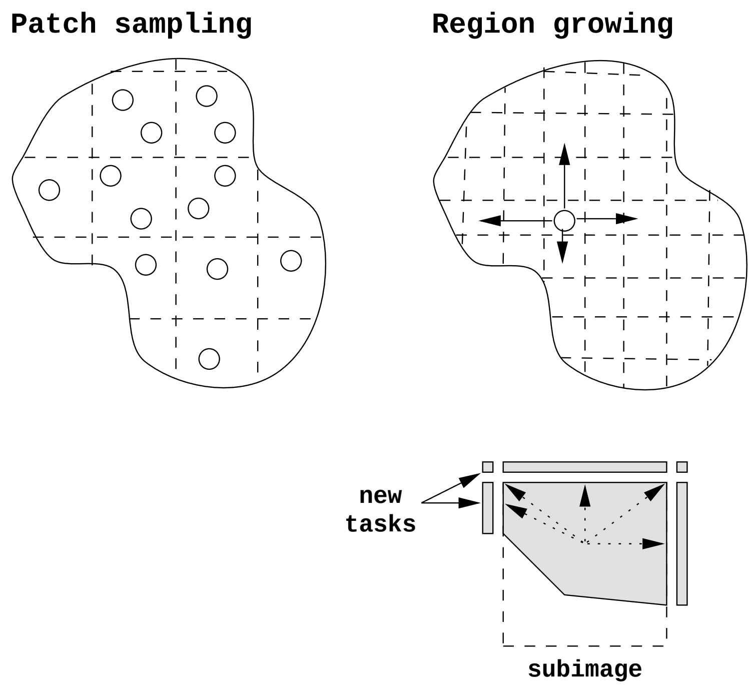


Figure 7.96: Illustration of Region-growing Algorithm

Global parameters	NumSamples	Number of samples per patch
	SampleSize	Maximum number of points per sample
	SampleFactor	Granularity control for sampling
	FitThL	Lower fitting threshold
	FitThH	Upper fitting threshold
	GrowThresh	Growing residual threshold
Phase 1: Patch sampling		
Skeleton	DF	Data farming
Global fixed data	xpatch,ypatch	List of patch (x,y) co-ordinates
Instantiation	GetSamples	Sample and grow within patch
Input	patch	patch number
	samples	number of samples
Output	chisq	χ^2 for best sample
	xbest,ybest	Points for best sample
	coeffs	Fit coefficients
Phase 2: Region-growing		
Skeleton	TF	Task farming with task spawning
Global fixed data	xpatch,ypatch	List of patch (x,y) co-ordinates
	gD	Geometric decomposition
Instantiation	GrowRegion	Grow from seed within subimage
Input	patch	Patch number
	gDnum	Subimage number
	xbest,ybest	Seed points
	coeffs	Fit coefficients
Output	xpatch,ypatch	List of added patch points
	patch,gDnum,xbest,ybest	List of new tasks

Table 7.22: Region-growing Parameterisation

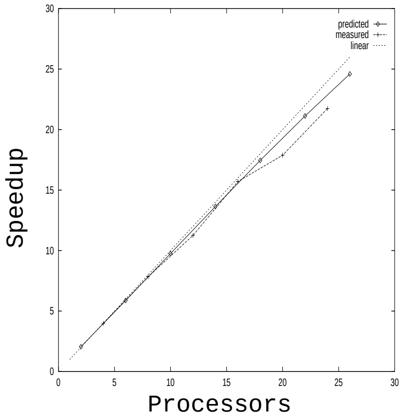


Figure 7.97: Speedup for RG2 (FarmDivisions) Growing Implementation

Operation	Type
send parameters to pose clustering	c
processor counts	n
initialise control data	n
load image and calibration data onto workers	c
transmit general control parameters	c
distribute geometric decomposition	nc
read model data and send to pose clustering	nc

The codes are; **n** for computation and **c** for data transmission.

Table 7.23: Sequence of Events During System Initialisation

system. In particular, data rearrangements and any other "glue" routines required to compose the modules are pointed out.

7.3.1 System Organisation

System Initialisation

Table 7.23 shows the sequence of events that occur prior to running the main segmentation software. Some of these are substantial, particularly reading image, calibration and model files and sending this data on to other processors. Some of these activities take part of the burden off the individual calculations (sending parameters, image and decomposition data) and are present in the initialisation code for efficiency reasons. The overall effect on the system is a beneficial one but the performances of the individual calculations are probably slightly flattered.

Depth Route

Table 7.24 shows the complete sequence of events in the depth route. All the parallel implementations discussed give reasonable parallel performance. The main limitation on the parallel performance as a whole is the data fitting routines. These are implemented partially in parallel in that each patch is sent to a separate worker for processing and the results are returned to the master processor for collection. Unfortunately, it is quite common for one large patch to be present in the image which limits the parallel performance. Since the fitting involves inverting a large matrix (up to 5,000 by 11 for a large cylindrical patch) this constitutes a major bottleneck in the system which could only be solved by implementing the fitting process in parallel.

The other major limitation is the data rearrangement phases which occur between some of the main parallel components. This is symptomatic of the general nature of the data being

Phase	Operations	Type
HK	initialise work frame	s
	run HK GD skeleton	p
	redistribute HK data to all workers	r
HKSEG	run HKSEG GDIW skeleton	p
	renumber patch labels to be consecutive	s
	send patch labels	r
	upload iconic patch data	r
	redistribute iconic patch data to all workers	r
	refit all patches	p*
	scrub small patches	s
	merge patches with matching normals	s
	redistribute iconic patch data to all workers	r
RG2	run patch sampling DF skeleton	p
	run region-growing TF skeleton	p
	clean up patch data	s
	refit all patches	p*
	calculate normals and centroids	s
	install in pose cluster format	s

The complete schedule for depth image processing. The task codes are; **p** is a parallel process, mostly skeleton implementations, ***** is a parallel process such as fitting patches in parallel (this is not efficient parallelism but speeds up the system slightly), **s** is a sequential overhead, and **r** is a data rearrangement.

Table 7.24: Sequence of Events in Depth Route

processed and the policy of using preloading extensively throughout the system. However, it should be remembered that these data rearrangements actually reduce the overall communications burden of the system. Without implementations of non-preloaded algorithms this remains purely a conjecture but experience with such algorithms has shown that for larger numbers of processors communications loads can become a limiting factor.

Intensity Route

Table 7.25 shows the complete structure of the intensity route. Again, all the main items of computation are implemented in parallel and the main overhead is updating with global data. The equivalent of the region-growing algorithm for the depth route is the post-processing phase at the end. This scans lines using Scher's algorithm [298] to cluster collinear points into line segments and coincidentally calculate the line endpoints which are required by the pose clustering algorithm. This phase is implemented partially in parallel inasmuch as each line is processed by a separate worker. Unlike the depth phase fitting routines, however, there are usually more lines than workers so the data farming skeleton DF is used here.

Phase	Operations	Type
CANNY	initialise work frame	s
	run CANNY GD skeleton	p
	redistribute CANNY data to all workers	r
ET	send ET parameters	s
	run ET tracking DF skeleton	p
	run ET line merge DF skeleton	p
	scrub short lines	s
POST	post-process lines	p*
	install in pose cluster format	s

The complete schedule for intensity image processing. The task codes are; **p** is a parallel process, mostly skeleton implementations, ***** is a parallel process such as fitting lines in parallel (this is not efficient parallelism but speeds up the system slightly), **s** is a sequential overhead, and **r** is a data rearrangement.

Table 7.25: Sequence of Events in Intensity Route

Message	Routine	Workload
IncCounter	increment general counter	l
IncTime	increment time counter	l
WorkBits	read in image segment	h
PatchFit	read and initialise patch header	m
PatchPts	read patch points into patch data	h
HKPatchCorrespond	maintain patch table	m
RG2bestsample	record best incoming sample	m
RG2boundpoints	handle RG2 TF incoming tasks	m
LineFit	read and initialise line header	m
LinePts	read line points into line data	h

These are the messages which perform processing. The Workload codes are; **l**ight, **m**oderate and **h**eavy loads. These loadings are purely subjective and vary from incrementing a single integer (light) to dumping point data into buffers (heavy for depth data, moderate for intensity data). Unfortunately, these cannot be quantified more precisely without running substantial profiling experiments.

Table 7.26: Computational Loads for Incoming Messages

Communications Input Routines

The communications handling routines described in Section 5.2.1 hide quite a lot of computation, that is significant computation is carried out by these routines independently of the processes which initiate them. To ensure the system does not back up, each message must be processed quickly but the sum total of the calculations carried out by these routines can be quite substantial if large numbers of messages are received. This is taken into account by the speedup calculation and is effectively a sequential overhead. Table 7.26 shows a summary of the messages with such computation.

7.3.2 Cylinders

The system has the capability to segment cylinders from depth data. These were added mainly to improve the quality of the pose estimation produced by the pose clustering routines but were not implemented in parallel due to time considerations.

Cylinders are detected at the coarse HK segmentation phase. When a patch is gathered on a worker it is classified by the HK sign code. For planar patches both H and K are thresholded at zero. Cylindrical patches have a positive H value for convex cylinders and a negative value for concave cylinders although none of the test images have concave cylindrical data. Planar patches are fitted with a planar fit and accepted or rejected on the basis of the fit quality, cylindrical patches are fitted using a general quadric and a separate threshold from the planar threshold is used to accept/reject the cylinder. Cylinders are not subsequently grown using the region-growing algorithm since an unconstrained quadric tends to "wrap" over other neighbouring surfaces. The coarse HK determined patch is fitted and the radius and centre of axis projections determined for transmission to the pose clustering. The radius calculation involves a high-complexity, rather naive nonlinear function minimisation which is a substantial overhead to the whole system but could be implemented in parallel.

7.3.3 Overall System

Figure 7.98 shows the top level Occam2 fold for running the segmentation system and illustrates how the components described in the previous sections are combined in a top level `PROC`[1] within the Occam2 implementation. After initialisation the communications handlers (one for segmentation workers and one to communicate with pose clustering) are run in parallel with the control process which drives the whole system and the two main processing routes which are slave tasks for the control process. In addition, both the screen and the

[1] A `PROC` is an Occam2 procedure.

channels to the workers have to be multiplexed which requires several additional processes provided by the `multiplexors` fold. During operation all computations are timed, both from start to finish and the combined sequential time for parallel operations. These are used to provide performance outputs which are dumped to the screen. The arguments to this `PROC` are the numbers of depth and intensity processors and a flag that signals loading of images onto the workers (the image data is retained between runs).

```
PROC RunPs(VAL INT dprocs,iprocs,VAL BOOL preload)

  ...  local variables
  ...  local procs
  SEQ
    ...  initialise
    PRI PAR
      ...  read in data from workers
      PAR
        ...  communicate with pose clustering
        ...  control process
        ...  depth route
        ...  intensity route
        ...  multiplexors
    ...  performance printouts

:
```

Figure 7.98: Top Level Fold for Complete System Master Processor

This `PROC` is called by a command line interface which allows activities such as defining image filenames, setting the parameters for the various phases along with other system-wide parameters, loading and saving image data from/to files and saving the resultant segmentation to files and also some debug utilities such as dumping buffers to the screen.

The top level Occam2 fold for the slave processors is illustrated in Figure 7.99. Here there are five main processes, two for handling input message (broadcast messages are sent separately from other messages), one each for the depth and intensity processes and one for input/output multiplexors. The `WorkDataRequest` function is a low-level data transfer routine, `RGFIT` is the `RefitPatches` worker function and `EMTEST` is the emulated farm discussed in Sections 4.4.3 and 4.5.3. The other functions are all vision processing routines. Due to the mixture of broadcast and point-to-point messages used in the system, broadcast messages are received on a separate channel from other messages resulting in the two message handling functions. All code for both the depth and intensity routes is present on all the processors but notice that the `Work` parameter to these processes is the same in each case so the two cannot be active simultaneously. Each of the depth and intensity routes is a passive process which does nothing until it receives a message from one of the `messageHandler` processes so at their top level they consist of a large `CASE` construct indexing all of the route functions.

```
PROC psDep(CHAN OF PSDEP fromPs,toPs,CHAN OF ANY toDebug,[]BYTE Work)
  ...  local variables
  ...  local procs
  SEQ
    ...  initialise
    WHILE psDepRun
      SEQ
        fromPs ? CASE
          ...  PD.WorkDataRequest
          ...  PD.HK
          ...  PD.HKSEG
          ...  PD.HKSEGGATHER
          ...  PD.RGFIT
          ...  PD.RGGETSAMPLES
          ...  PD.RG2GROWDIVISION
:

PROC psInt(CHAN OF PSINT fromPs,toPs,CHAN OF ANY toDebug,[]BYTE Work)
  ...  local variables
  ...  local procs
  SEQ
    ...  initialise
    WHILE psIntRun
      SEQ
        fromPs ? CASE
          ...  PI.GFFILTER
          ...  PI.CANNY
          ...  PI.EDGETRACK
          ...  PI.ETMRGLINES
          ...  PI.ETPOSTPROC
          ...  PI.EMTEST
:

PROC psSlave([]CHAN OF ANY fromTiny,toTiny,CHAN OF ANY fromDebug,toDebug,
             VAL []INT taskIfMask,VAL INT taskId,numTasks)
  ...  local variables
  ...  local procs
  SEQ
    ...  initialise
    PAR
      ...  messageHandler
      ...  messageHandlerCast
      ...  psDep
      ...  psInt
      ...  multiplexors
:
```

Figure 7.99: Top Level Fold for Complete System Worker Processor

7.3.4 Module Workload

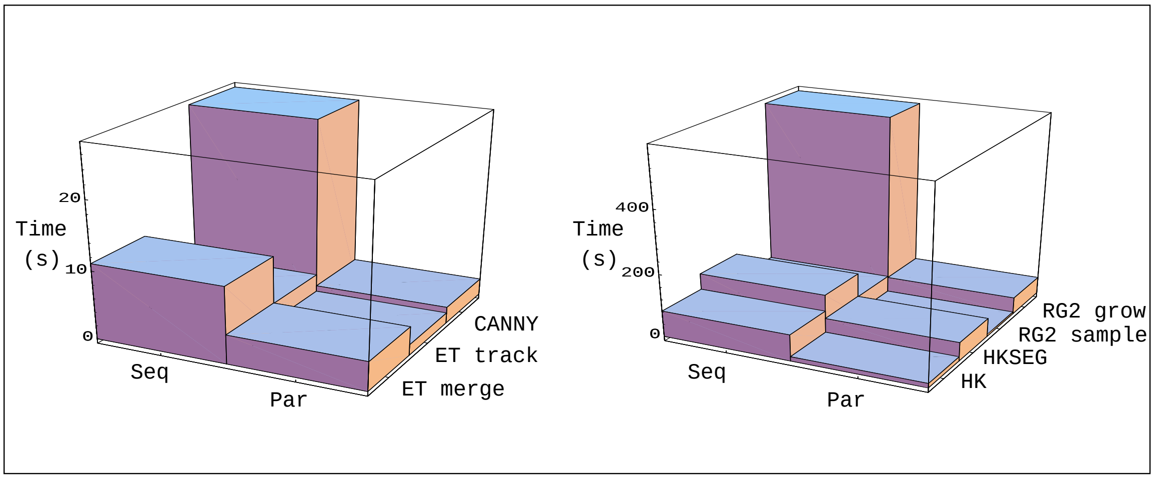


Figure 7.100: Sequential and Parallel Processing Time for Segmentation Modules

Figure 7.100 shows the relative processing times for each of the major parallel modules in terms of both sequential and parallel processing times. There is a great imbalance in the workload between the various modules. For a single image, with each phase running to completion before the next phase starts, the imbalance within each route does not present any problems. If the system were to operate in pipelined mode with each module running in parallel and images being pumped through the system, then creating a balanced pipeline could prove difficult.

More seriously, there is a huge imbalance between the intensity and depth routes, of the order of about 8:1 which makes it difficult to balance the two routes (which run concurrently). The ability to reallocate processors when the intensity route terminates would be advantageous but is impractical given the amount of intermediate data that would have to be transferred between the processors. This is a consequence (and a possible drawback) of using preloaded data prior to computation.

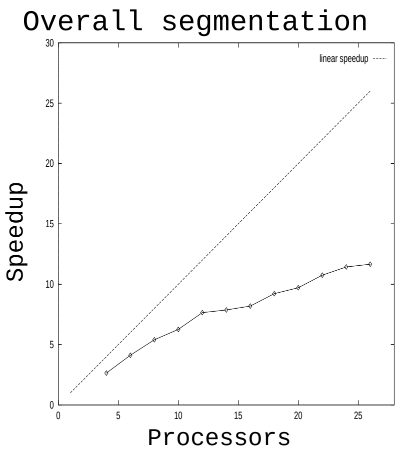


Figure 7.101: Speedup for Overall Segmentation System

Figure 7.101 shows the approximate speedup achieved for the overall system assuming the optimal balance between the depth and intensity routes. Note that one intensity processor was used for 4 to 10 overall processors and two intensity processors were used for 12 to 26 total processors. At the top end of this plot, three intensity processors give similar performance to two. Note also that this performance includes downloading of the raw images onto the worker processors at the start of computation (but not reading them from files). It also includes several data rearrangements and redistributions throughout the calculation and some post-processing on the resulting data, for instance, calculating centroids and normals for planes.

The performance is much lower than the individual components due to these overheads and is dominated by the major workload modules, particularly the RG2 calculation routine.

7.4 Summary and Conclusions

In this chapter, the implementation of a complete and parallel multisource segmentation system from SML prototypes was presented. On the whole this was a fairly mechanical process once the details of the sequential algorithm and the mode of parallelism were decided during prototype construction. To summarise the achievements of this chapter:

- Five modules were implemented in parallel and all gave reasonable performance, some gave excellent performance. There was not scope to discuss the specific development of each module as some went through several incarnations before being finalised, but an overview of each module's instantiation from the prototype and subsequent performance was presented and a typical performance profile shown. The use of algorithmic skeletons in this process was absolutely vital. It is unlikely that a system of this size and complexity could have been designed and hand-coded in the available time without a core abstraction concept to focus the design process and guide the implementation. If the skeletons concept could be realised in the form of a parallelising compiler then the system could not only be developed much more quickly but much more optimisation would be possible.

- The major problem associated with the system was memory usage. The design decisions led to a system which required large amounts of local storage on the workers and global data on the master. In retrospect, it might have been better to have used a more distributed approach to alleviate this difficulty but this would have made the analysis and construction of the code more difficult and placed an added burden on the communications. Considering previous applications had tended to be communications bound and considering the limited manpower available to the project, the current approach was probably the best one, despite the memory problems.

- The main obstacle to refining the existing system is the amount of hand coding required to implement optimisations. Very little optimisation was carried out other than to allocate processors to achieve balance between the two routes. Again, this could be supported by an automated compiler but this raises questions about what effect optimisations may have on global system performance. The entire system would have to be uniformly developed using the compiler, including all the support and linking code which would have implications for the computational complexity of the optimisation process.

- The completed system was built by composing the modules together under the auspices of a control system. A surprising amount of code was required to link these modules together. This was mostly concerned with gathering and distributing of input/output data from the modules and involved reindexing and other data restructuring.

In this chapter, the parallel performance of the individual modules was presented and the overall performance characteristics were illustrated for one image. A more complete characterisation of the system is deferred to the next chapter.

Chapter 8

Parallel System Characterisation

8.1 Introduction

In this chapter the complete visual recognition system is characterised. The main point of interest is the *parallel performance.* Note that the focus is on the performance of the complete system rather than individual modules. The quality of the segmentation and the quality of the resulting pose estimation from the matching phase are also briefly discussed.

A small set of test images is used and various experiments are carried out designed to test the system in as many different situations as possible. These situations are not chosen to flatter the system. Both the parallel performance and segmentation quality will break down if the system is used outside its useful range. This, however, is a property of all visual systems, including the human eye which is the ultimate benchmark for all such systems; consider for example finding a familiar face in a large crowd, which may take a long time, or reading a traffic sign in heavy fog. This chapter is designed to find where the limits of the system are rather than to compare with other visual recognition systems. A brief comparison, however, is made with some of the systems reviewed in Section 2.2.

8.2 Registered Images

The data required by the system is a perspective intensity image, an orthogonal depth image, a calibration file and a model description file. The depth and intensity images are required to be *registered*; this means that a transformation matrix between the two images is provided along with the camera position and parameters for the perspective intensity image.

The intensity and depth data are in HIPS format files [265], these files contain basic image information such as numbers of rows and columns, pixel format and colour information. The

Sequence Description field in the header for the depth data contains additional information including the camera parameters. The depth data is in iconic form and the additional header information contains scaling information which has to be applied to the x and y pixel co-ordinates to get world (x, y, z) co-ordinates (the z value is the pixel's value in the iconic image).

The calibration file contains the calibration information from the depth sensor. This data includes both intrinsic and extrinsic parameters estimated according to techniques by Tsai [299]. These include camera position, orientation, focal length, radial lens distortion and image scanning parameters. Additional information is the transformation matrix from world co-ordinates to camera co-ordinates.

The model data needs to be accurate and is compiled off-line to include some calculated parameters not found in the standard model (SXF) format.

8.3 Test Images

Three synthetic and four real images were used to test the system. Note that the real data contains measurement errors which are manifested as noise and distortion in the images and as inaccuracies in the calibration parameters. The registration parameters also have errors associated with them. In practice these two sets of parameters are combined into a single set of parameters as if the images were generated in a registered state. The synthetic data should be perfectly registered up to quantisation errors.

The four real scenes are similar. These contain a single object, mostly made up of orthogonal planes but with one significantly sized cylindrical patch. The blocks have been painted white to minimise erroneous depth measurements due to reflections and to accentuate the edges in the intensity images. In addition, the blocks are presented to the cameras such that significant portions of planes that are orthogonal to each other are in view. This aids the matching process since a point vector model is used in the pose clustering. Vectors with significant angular differences allow better positional accuracy.

Figures 8.102, 8.103, 8.104 and 8.105 show these four scenes, their intensity images, depth images (rendered by surface normal shading) and a sample segmentation and matching. The depth images have been preprocessed using a least median of squares smoothing algorithm [49]. This both smoothes surface noise within the planes and interpolates missing data when there is sufficient neighbourhood evidence. In addition, the background plane is removed when the iconic depth image is generated from lists of (x, y, z) points.

The synthetic images contain one (Figure 8.106), two (Figure 8.107) and three (Figure 8.108)

Phase	Description
CANNY	Canny edge detector
ET	edge tracker
HK	mean/Gaussian curvature calculation
HKSEG	coarse HK segmentation
RG2	region-growing

Table 8.27: Phase Mnemonics for Parallel System

Phase	Description
Seq	The depth and intensity routes are run consecutively.
Par	The depth and intensity routes are run concurrently.
Icyl	When TRUE the segmentation system attempts to identify cylinders in the depth image, when FALSE cylinders are ignored.
iprocs	Refers to the number of intensity processors used during a run.
dprocs	Refers to the number of depth processors allocated.
1x2	Pose clustering is run on a 2D array of 1 by 2 processors.
2x2	Pose clustering runs on 4 processors in a 2 by 2 array.

Table 8.28: Mnemonics for System Characterisation Chapter

objects. These were generated from the models used for images 1 and 4. Cylinders are rendered using faceted surfaces so cylindrical segmentation is not used in these images.

This chapter is divided up into four main sections. Section 8.5 discusses the segmentation quality, Section 8.6 presents timing diagrams for the various system components, Section 8.7 presents some experiments showing the parallel performance.

8.4 Terminology

Note that the system is divided up into the phases discussed in the previous chapter. For brevity the mnemonics used in the last chapter are used throughout this chapter, these are illustrated in Table 8.27.

A complete list of the system parameters is presented in Appendix B, those parameters which are varied are discussed in the context of the relevant section in this chapter. Some other mnemonics, specifically used in this chapter are in Table 8.28.

8.5 Segmentation Quality

8.5.1 Synthetic Images

Note that the synthetic images are all generated from the data found in the model file so there are no inaccuracies between the model data and the synthetic image data. One problem,

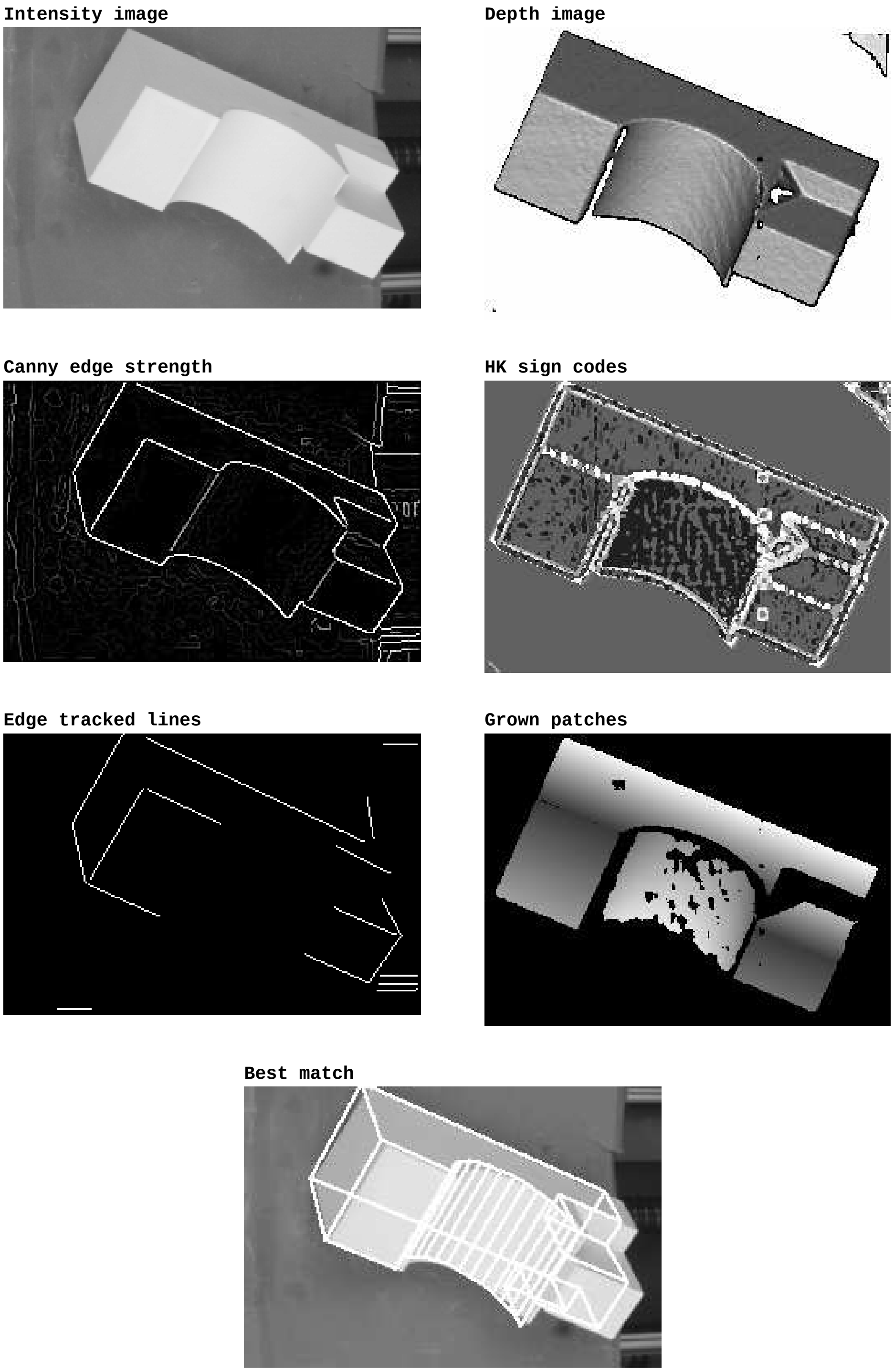


Figure 8.102: Image 1: Sample Segmentation and Matching

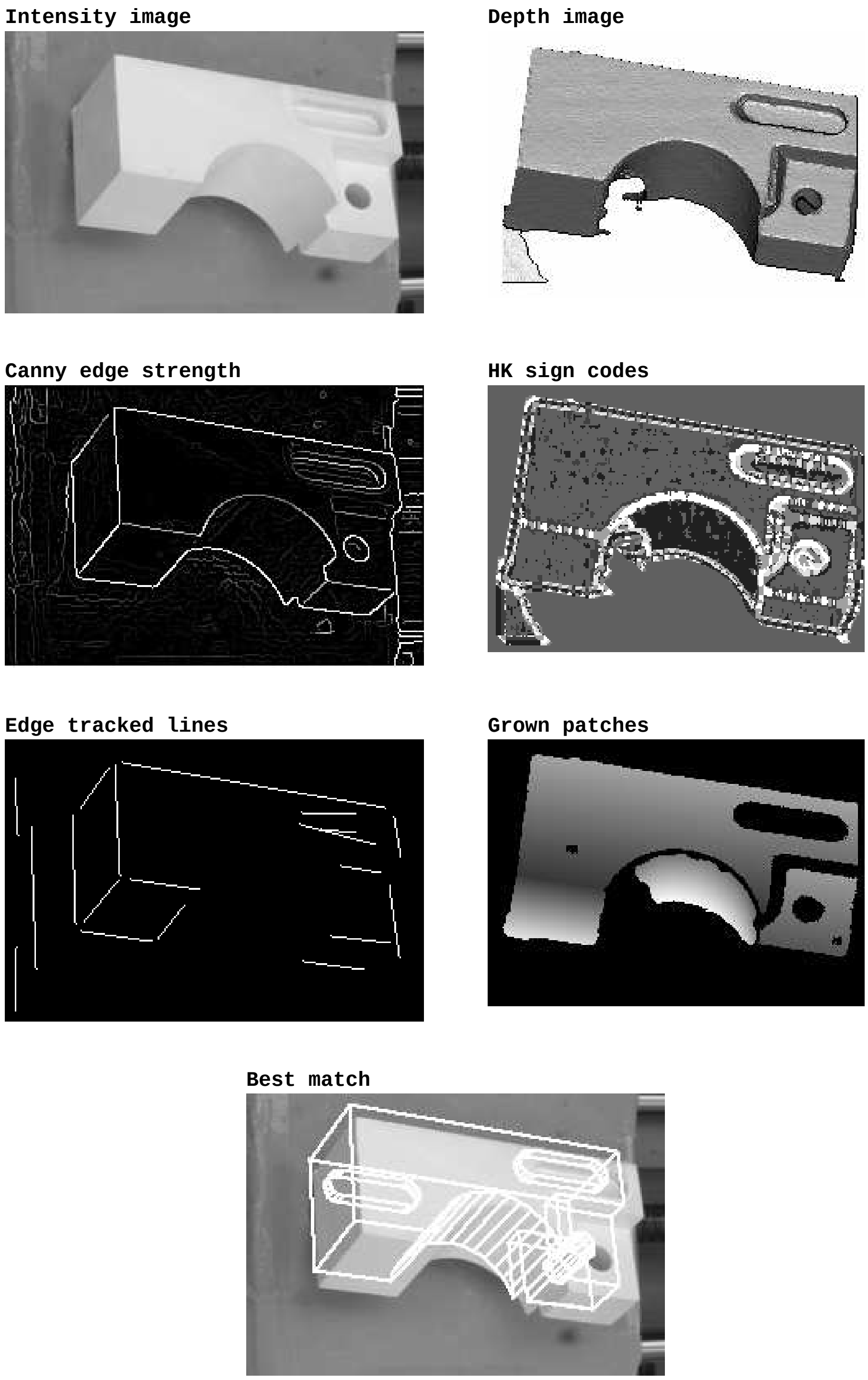


Figure 8.103: Image 2: Sample Segmentation and Matching

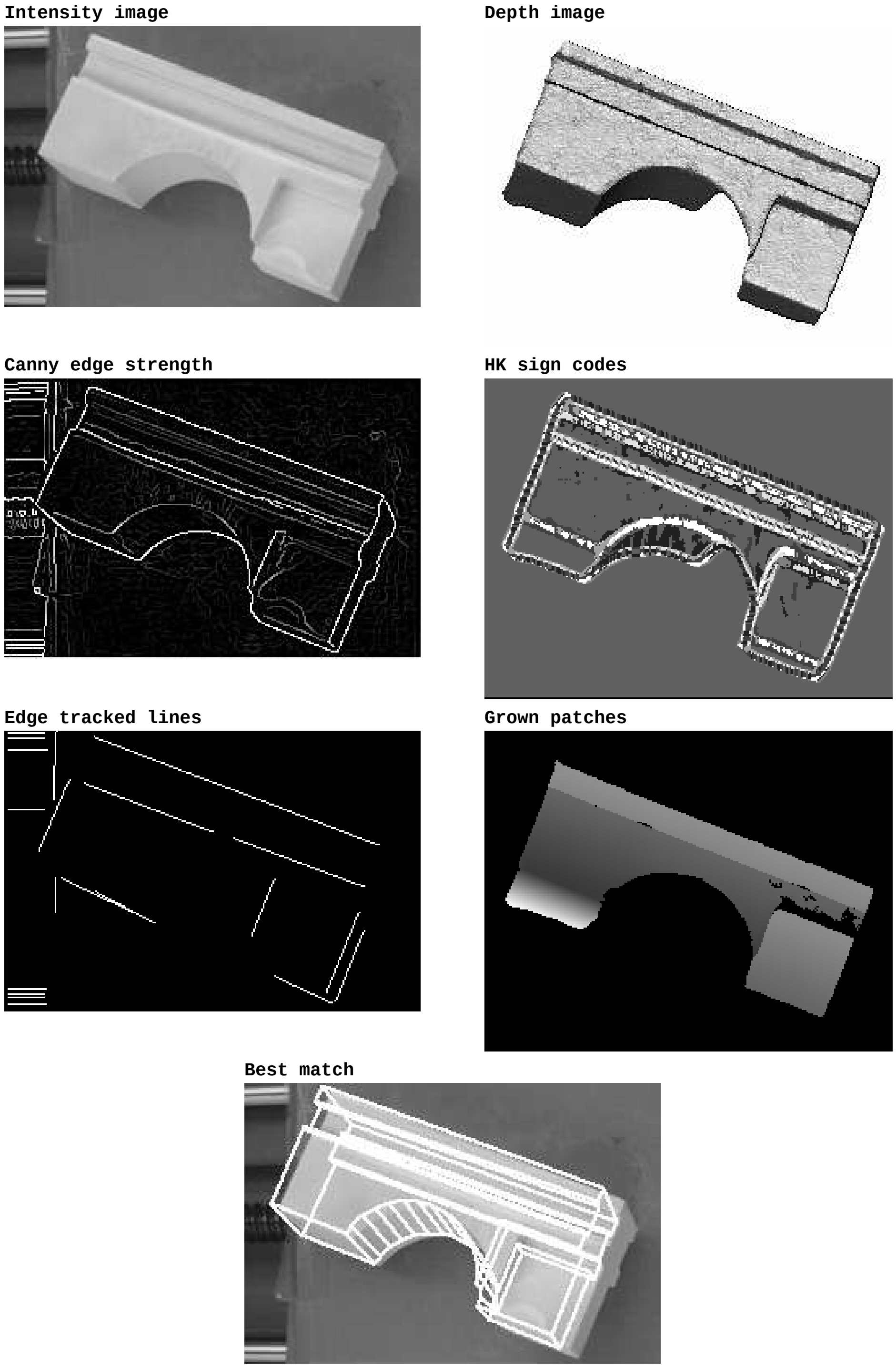


Figure 8.104: Image 3: Sample Segmentation and Matching

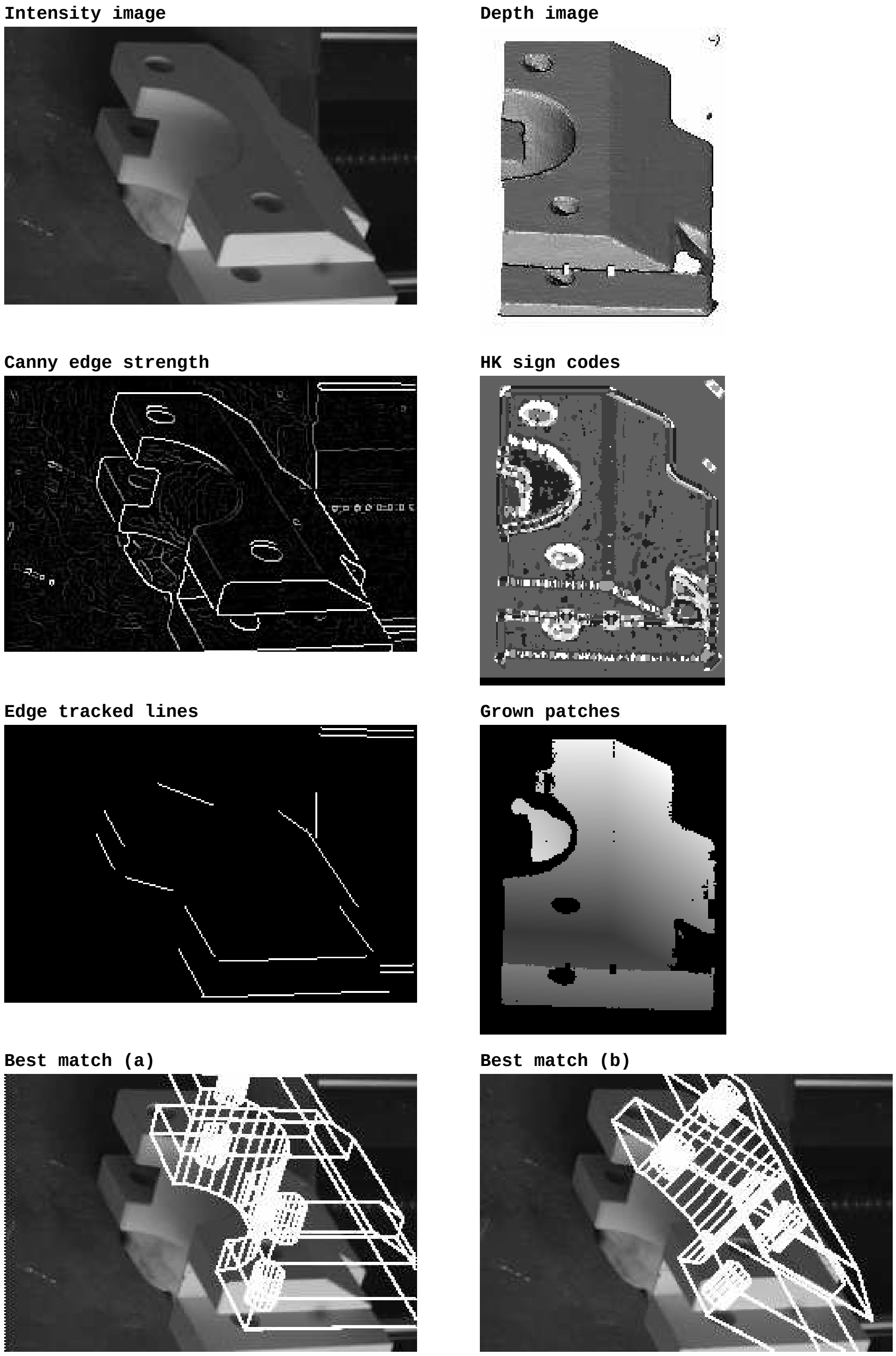


Figure 8.105: Image 4: Sample Segmentation and Matching

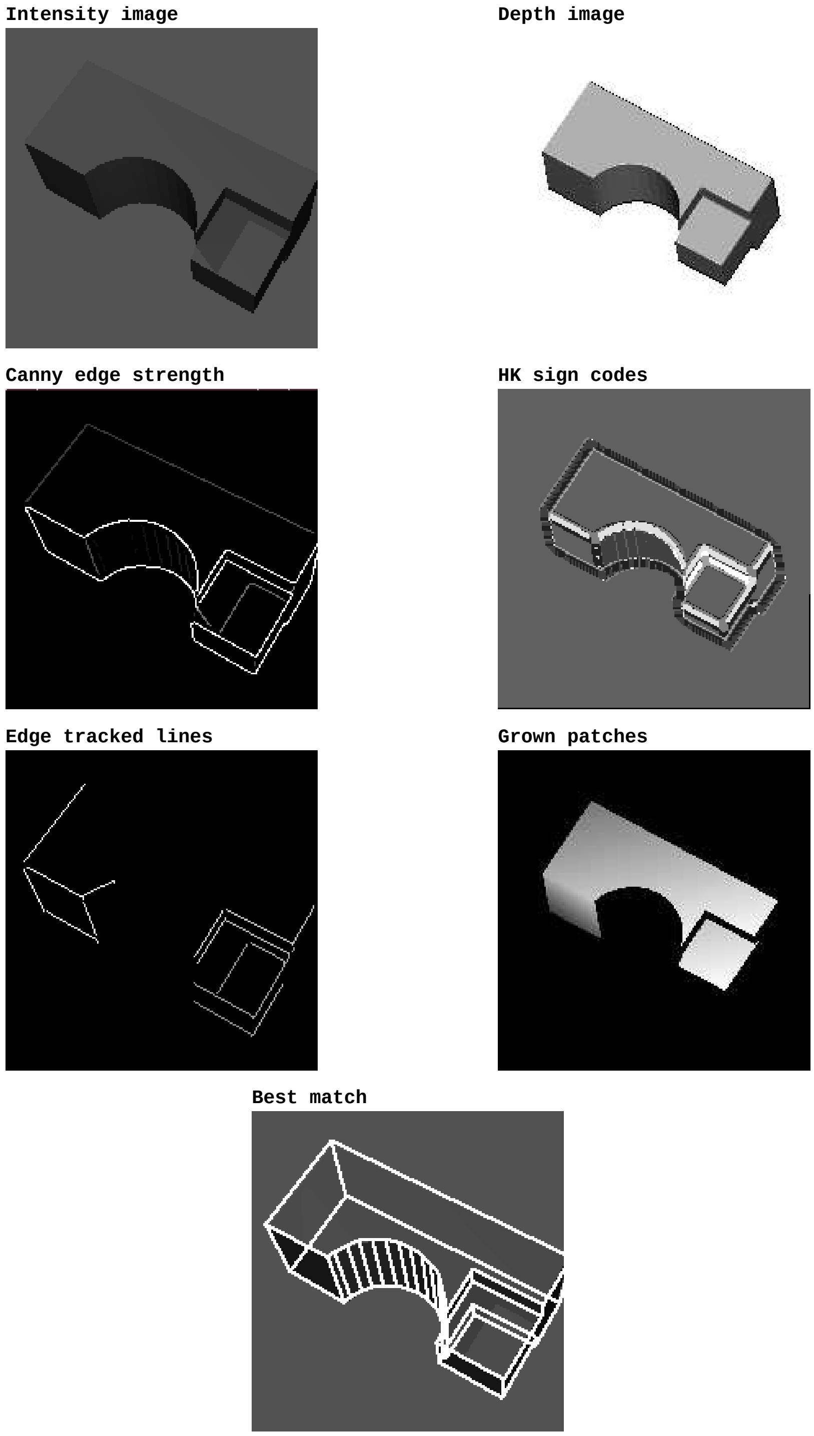


Figure 8.106: Image 5: Sample Segmentation and Matching

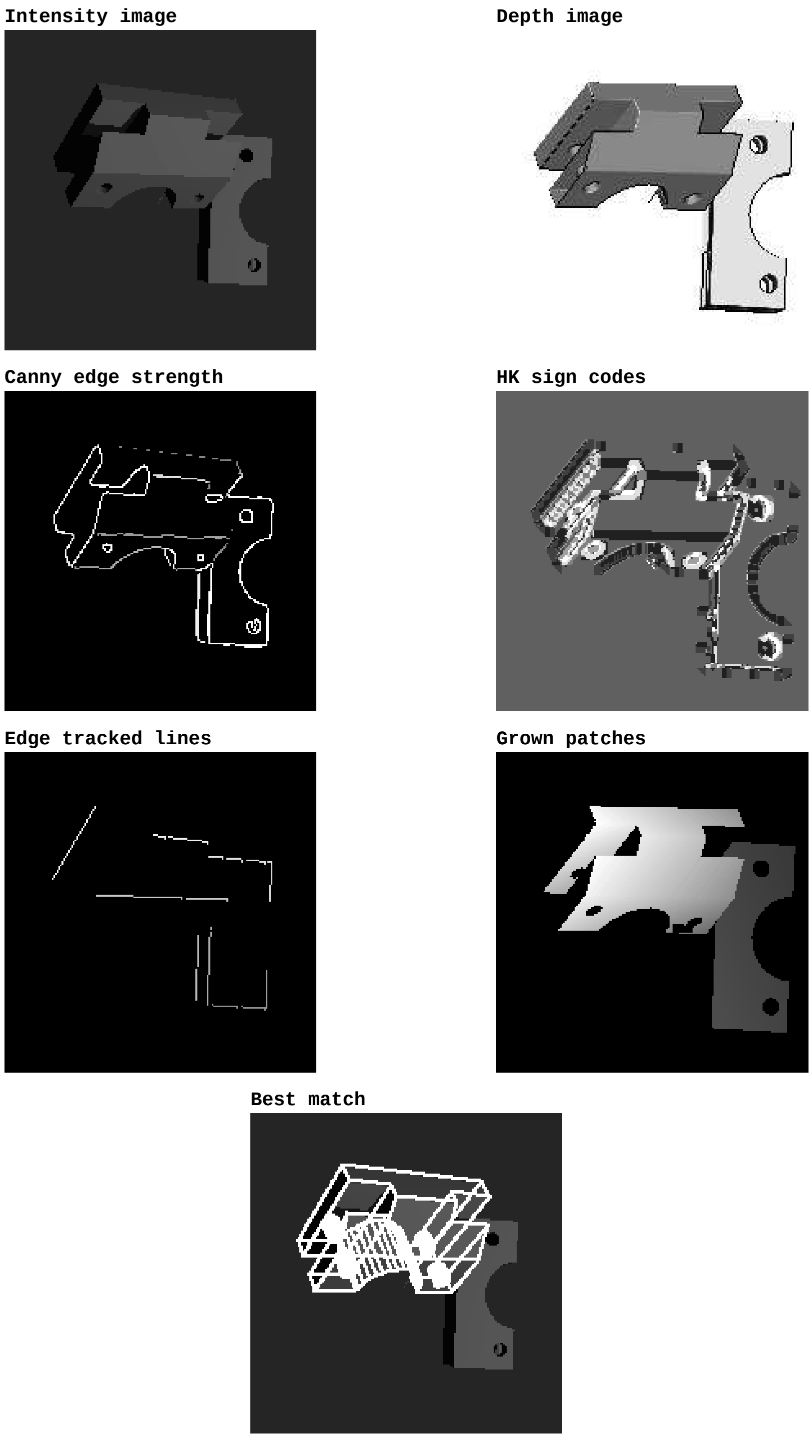


Figure 8.107: Image 6: Sample Segmentation and Matching

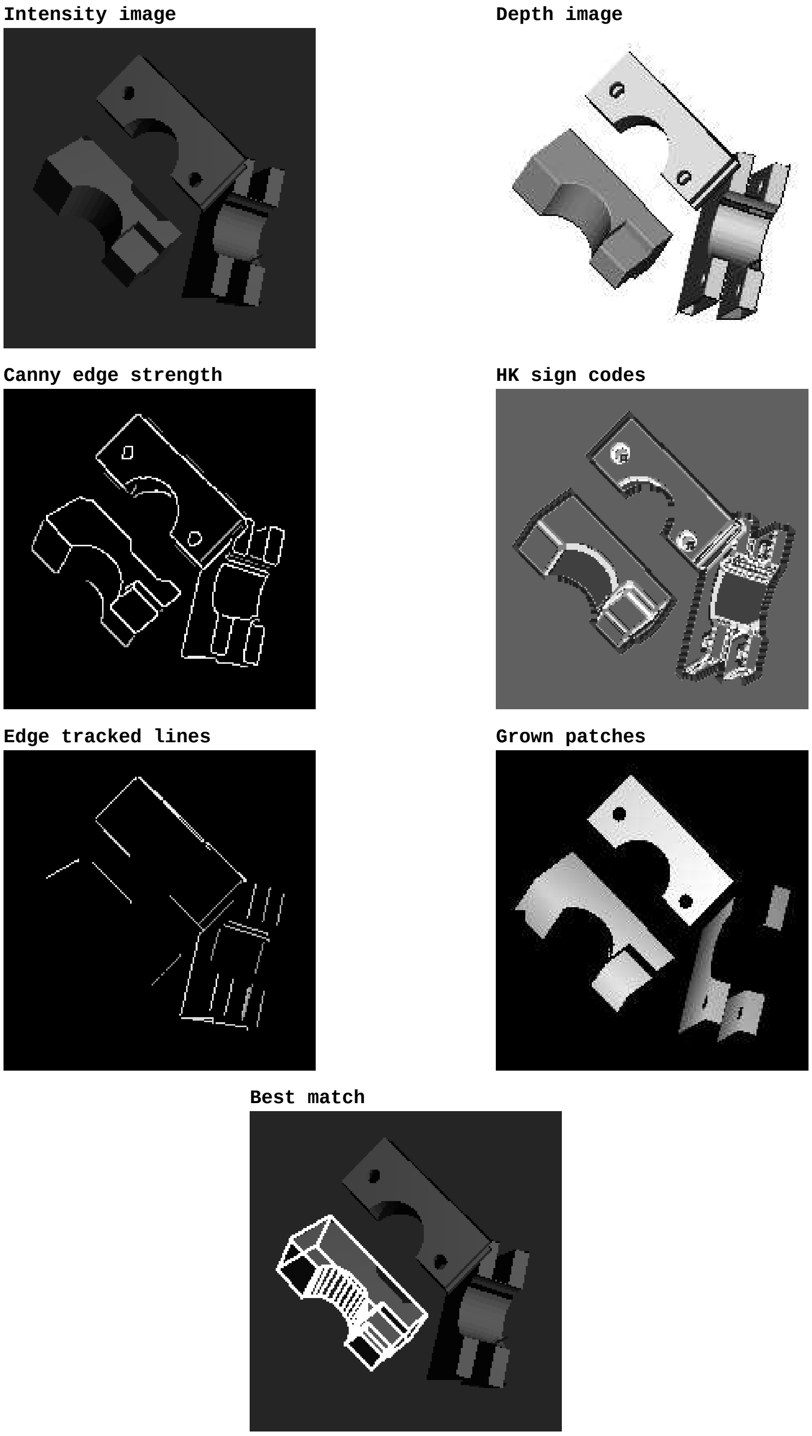


Figure 8.108: Image 7: Sample Segmentation and Matching

however, is that cylindrical patches are rendered using facets which prevents the segmentation system from being able to distinguish cylinders from flat planes. The synthetic images were processed with cylinder detection disabled.

All the synthetic images exhibit good segmentations, however. Image 5 (Figure 8.106) is the single object scene and all major features are detected apart from some lines in the intensity image which has poor contrast. The best pose shows an accurate match and verifies that the segmentation system and pose clustering are using the correct co-ordinate systems.

Image 6 (Figure 8.107) has two objects, one in the foreground in a good orientation for matching, the other slightly obscured and very flat to the viewpoint. The foreground object can be accurately matched with the equivalent model but the background object could not be matched at all with its equivalent model. In fact, this model gives a variety of false matches with the data from the foreground object.

Image 7 (Figure 8.108) shows a cluttered scene with three objects. Again, a good match is available with the most prominent object. A match with the object to the right was not possible since the combined complexity of the object and the scene caused too many possibilities for the pose clustering to handle.

The synthetic images are useful for testing and debugging the system under artificial conditions but are not realistic tests of the system's performance. Noise could have been added to the synthetic images to assess the system's degradation under noisy conditions. This was not done since the system did not have any noise-immunity built into the design and such immunity is not an objective of this work.

8.5.2 Real Images

The four real images are intended to be representative of the type of data the system would be presented with in a real application.

Image 1 (Figure 8.102) is probably typical of the best quality data such a system could see in practice.

The intensity image is such that the target object is in high contrast with the image background and in general all fold edges are in high contrast due to the lighting conditions. The Canny edge data is typical for this type of image, most major lines are detected but there is a fair degree of background noise. This has an important bearing upon the resulting segmentation, only the main line components are present in order to "filter" out the background noise. This compromise can be controlled by varying the edge thresholds and the minimum line size parameter. The best balance still includes quite a lot of noise since the noise in this image is quite significant relative to the object features.

The depth image is of a very high quality. Most of the surface data is present, apart from a few corner regions where noise is to be expected in range images of this type. Due to the local nature of HK segmentation the surface noise follows through into the HK sign code calculation. This limits the quality of the connected component segmentation. There is a compromise between the missing small patch and the cylindrical patch. If the threshold on Gaussian curvature (K) is set too low then the surface "misclassification" noise on planar patches increases. This causes small or narrow patches to be split up. If it is set too high the HK sign code for the cylindrical patch tends to become planar.

Another compromise is apparent in the depth image. The window size used to generate the HK values results in *peaks* and *troughs* being identified in the HK codes (white in the images shown here) around plane junctions. This causes two problems. Firstly, it eats into the planar surface making it difficult to identify patches which have a dimension which is small relative to the minimum patch size parameter. Secondly, these peak and trough regions can actually become big enough to be recognised as patches themselves. A threshold on patch fitting quality allows these to be removed.

The balance between the H and K thresholds is also a tricky one to decide for a particular image. Theoretical results [300] suggest that the K threshold should be about the square root of the H threshold:

$$K_{thresh} \approx \sqrt{H_{thresh}} \tag{8.38}$$

Practically, noise prohibits such small values of the K threshold. It has been found by systematic experiment that a value of one tenth of the H threshold works well with these images:

$$K_{thresh} \approx 0.1 \times H_{thresh} \tag{8.39}$$

The resulting segmentation is of comparable quality to other methods [51, 301]. There are two main limitations. Firstly, there is a minimum patch width that can be detected. This is caused by the system's inability to handle noise properly. Secondly, the cylindrical patches are not grown. There is enough data to get a decent radius estimate but this will not be true where a lower percentage of the cylinder's circumference is visible.

Figure 8.102 shows the model superimposed on the intensity image for a correct match. Note that the pose clustering generates hundreds of matches ranked roughly in order of cluster size. Provided there is at least one "correct" match in the first dozen or so poses, a verification process could select the correct match more accurately.

The match shown here is not perfect. Since perfect matches are obtainable from the synthetic data and since the real images have been used in other vision processing systems [51] the most likely source of this discrepancy is a registration error between the depth and intensity images.

Image 2 (Figure 8.103) shows a view of a similar block to Image 1 but with some additional surface features, a milled groove on each side and a hole drilled through one section. These present no problems to the segmentation process.

The intensity segmentation is very similar to Image 1. Some of the lines around the milled groove are very poor, being thrown out by noise from inside the groove. There is still a sufficient number of good features as a basis for matching.

The depth segmentation is again very good for the main features but again the small and narrow patches are broken up by noise. Provided a patch has a "clear" region, i.e. one which is large with respect to the sample window size, the growing process can negotiate surface features. The small plane within the groove was detectable at some parameter settings but again both this patch and the cylindrical patch could not be detected at the same time.

The match shown in Figure 8.103 is similar in quality to that for Image 1.

Image 3 (Figure 8.104) shows a slightly harder problem than the first two images. The object is aligned with the camera axis in such a way that most of the visible planes are perpendicular to the line of sight. This object also contains a major discontinuity on the large side plane.

The intensity image segmentation is very similar to the previous two images. There are problems around the ends, however, due to the short lines visible in this viewpoint and there are also problems around the milled channel where there is poor contrast.

Image 4 (Figure 8.105) illustrates a case where the system essentially fails, despite there still being reasonably good data. The intensity image has quite poor contrast so that many of the major features are down in the noise "floor". In addition, the depth view is relatively "flat", the two main patches are now only at 135° to one another. There also appears to be more noise in some regions of this data. The two best matches are shown but these are well down in the rankings and not well localised.

8.6 System Timing

Figure 8.109 shows a timing diagram for system operation which includes processing of cylinders. This was carried out on Image 1 using 16 depth processors, 3 intensity processors and

4 pose clustering processors. The system was run in two distinct modes, one in which the depth and intensity routes were run consecutively and one in which they were run concurrently. Figure 8.110 shows the timings under the same conditions but excluding cylinder processing.

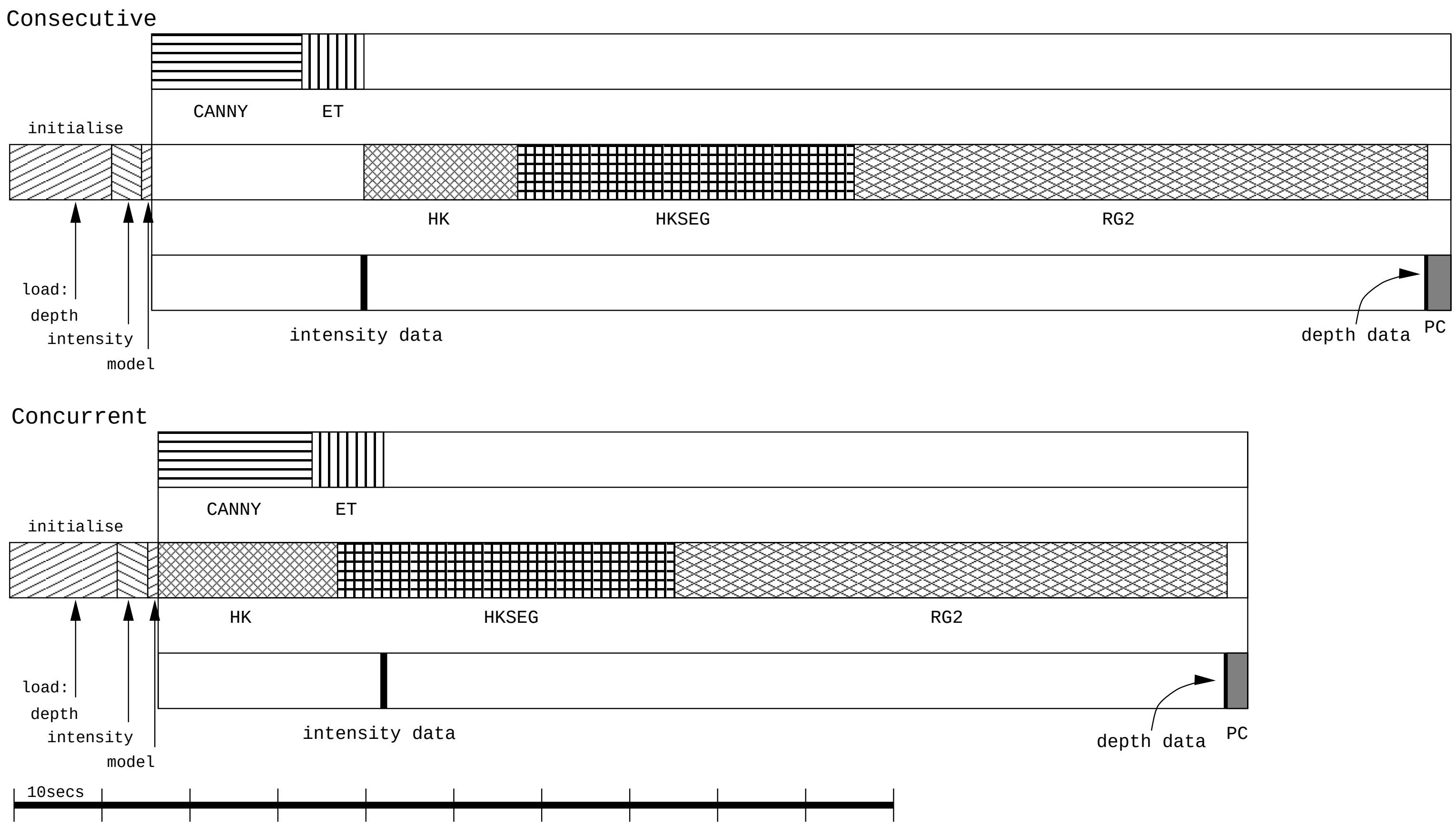


Figure 8.109: Image 1: Timing Diagram for System Including Cylinder Processing

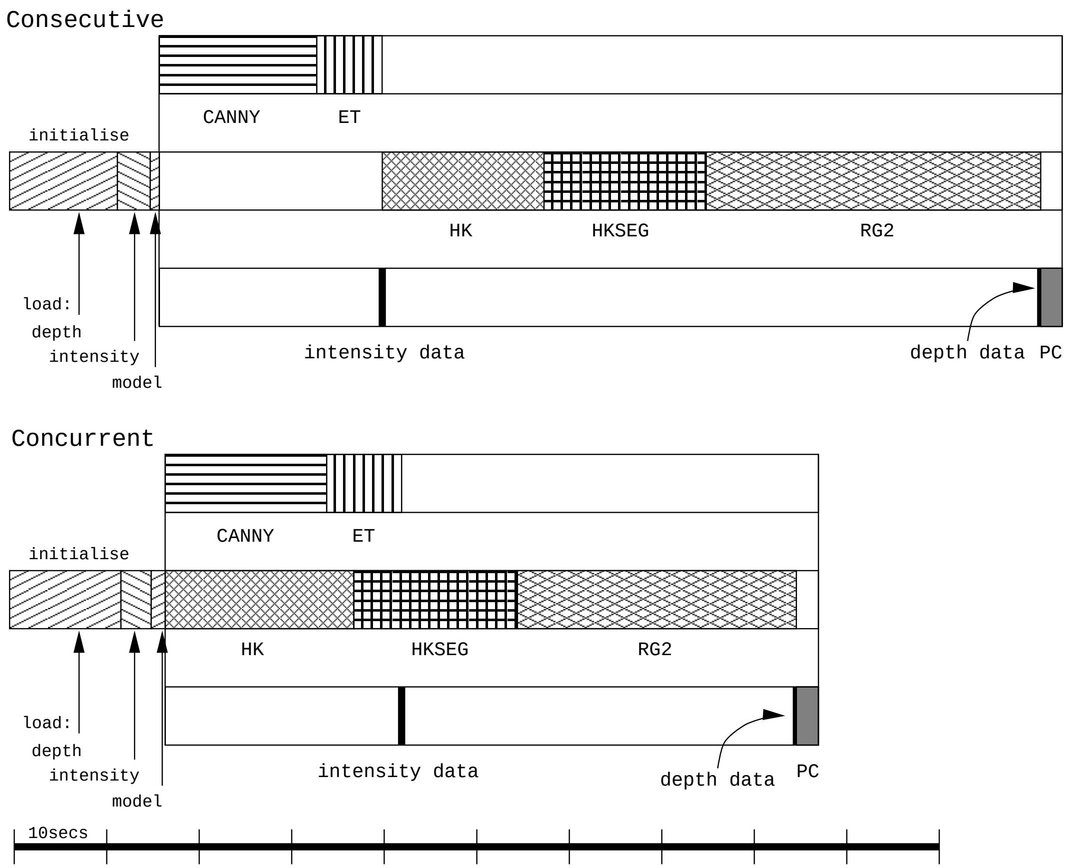


Figure 8.110: Image 1: Timing Diagram for System Excluding Cylinder Processing

Depth/Intensity	Cylinders	
	Yes	No
Seq	164	114
Par	141	88

Seq/Par refers to whether the depth and intensity routes were run sequentially with one another or in parallel. Cylinders refers to whether cylinder processing was included or not.

Table 8.29: Total Processing Times (Seconds) for Image 1

Both Figures show the significant performance gain available by processing both routes together. It should be noted, however, that if the system were reconfigured when one of the routes terminates there would also be a performance improvement. This can be implemented, for instance, by converting the intensity processors into depth processors. The differences between the two Figures show how cylinder processing affects the overall performance. As already indicated, this is computationally intensive and implemented sequentially so it is perhaps unfair to judge the overall performance of the system with these included. Note that including cylinders does not significantly affect the pose clustering times. This is due to the facts that in the point/vector model cylinders and planes have the same representation and that there is only one cylinder in the image.

Table 8.29 shows the total processing time for the system under the conditions in Figures 8.109 and 8.110. These show the two factors of concurrency and excluding cylindrical processing combine to halve the overall parallel processing time.

Note that the reason these figures have been presented for both sequential and parallel depth and intensity routes is that there are communications problems with the parallel version. When the data rearrangements occur between two phases the network is under heavy load. If both routes perform data rearrangements at the same time the network crashes. This might be solvable using a different communications model or providing larger buffers for the routing harness but this was not attempted. Using a different model would require substantial restructuring and recoding and no more memory was available for the routing harness. The performance figures in the following section are presented for the sequential case because complete sets of data could be gathered. It is assumed that the behaviour of the parallel case is inferable from these figures.

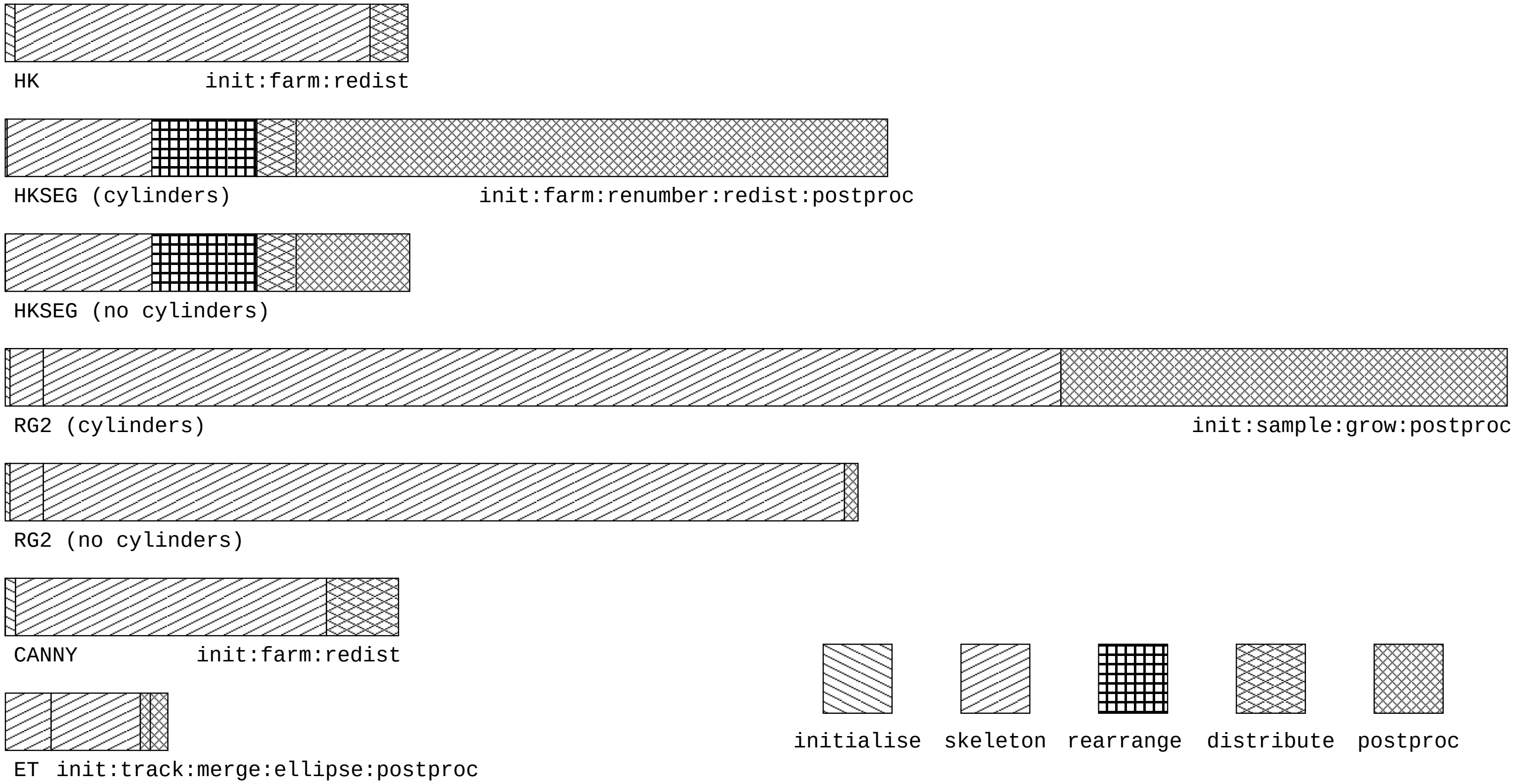


Figure 8.111: Image 1: Timing Diagram for Individual Phases

8.7 Parallel Performance

This section presents various experiments that were carried out on the completed parallel object recognition system. These are intended to characterise the behaviour of the system. The two main criteria for judging the system are the parallel performance and the segmentation/pose estimation quality. The experimental variables are the allocation of processors to the depth, intensity and pose clustering activities and the system control variables described in Appendix B. Note that throughout this chapter the variable `iprocs` refers to the number of intensity processors and `dprocs` is the number of depth processors.

8.7.1 Behaviour Under Different Processor Allocations

The allocations of processors to the three main activities were varied for each of the four test images. The numbers of intensity processors were varied from 1 to 3. Depth processors were varied from 6 to 18 in steps of 2. Pose clustering processors were used in 1×2 and 2×2 arrangements. In addition, each combination of processor allocations was carried out both including and excluding cylinder processing. For each system run, all the timing and segmentation/clustering data were stored but there is not enough room here to present all this data. The performance is summarised by plotting overall time for segmentation and for pose clustering. The speedup for the segmentation system is also plotted.

Image 1

Figure 8.112 shows the parallel run times for Image 1 for segmentation and pose clustering. Several points can be made about this data:

- The configuration of the pose clustering processors does not in any way affect the segmentation processing times. This is as expected.
- The pose clustering times are consistent across variations in the segmentation processor allocation. This is also as expected since the pose clustering is only affected by the number of features passed through and not by the processing time. This is an important verification of the stability of the segmentation with respect to number of processors. Remember there is a slight non-determinacy in the sampling process.
- The number of intensity processors affects the overall processing time slightly with about a 20 second reduction from 1 to 2 and about a 10 second reduction from 2 to 3 processors. More than 3 intensity processors are unlikely to improve this by much and would increase processing time by taking processors away from the depth route.

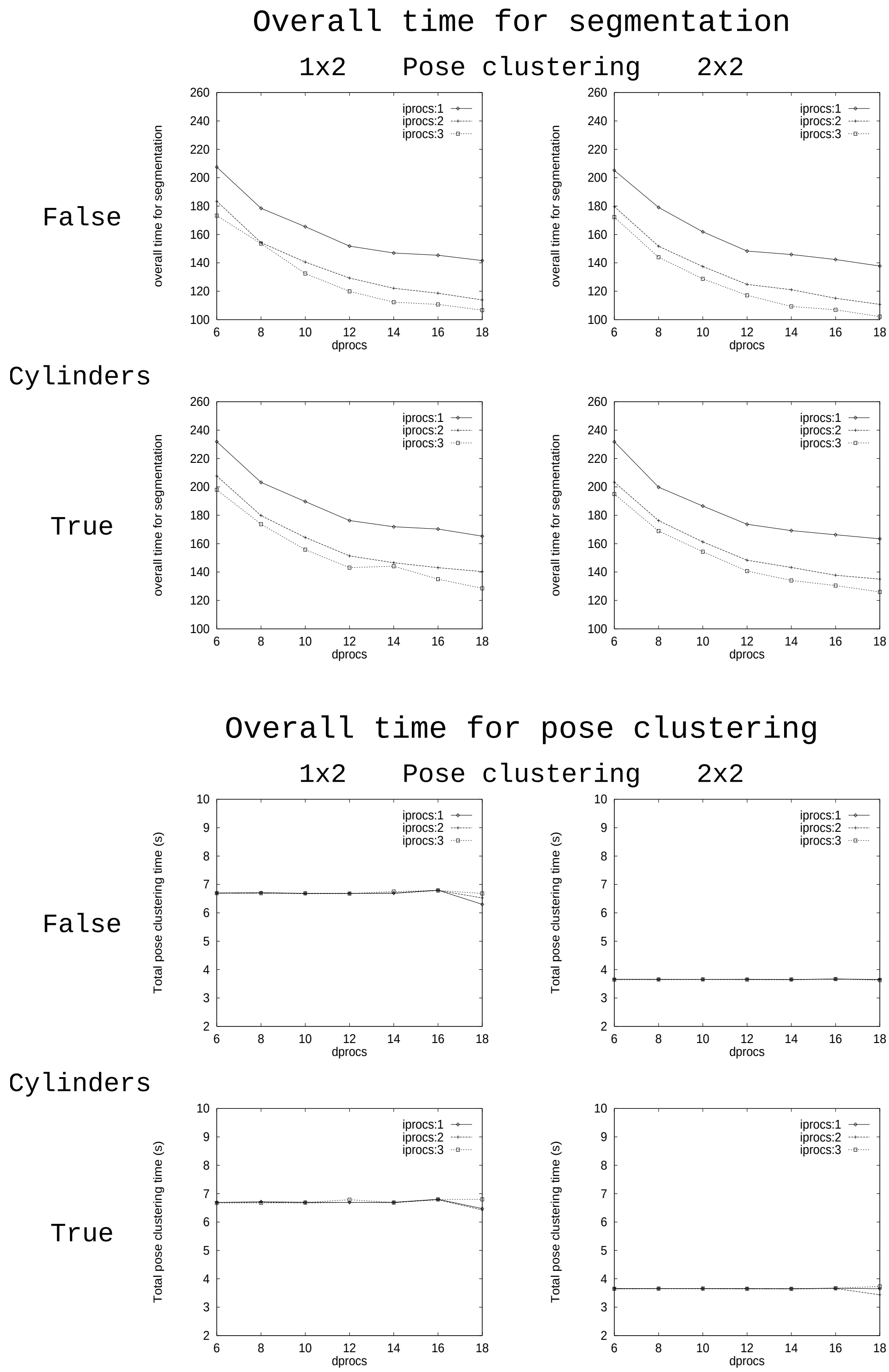


Figure 8.112: Image 1: Parallel Run Time for Segmentation and Pose Clustering

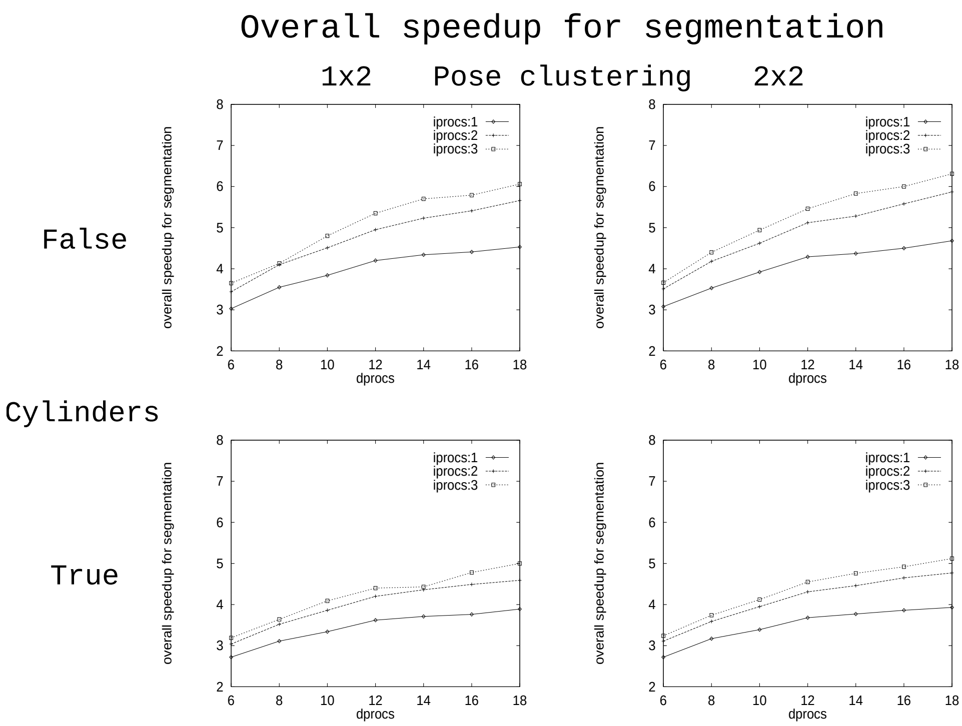


Figure 8.113: Image 1: Speedup for Segmentation

- Processing the one cylinder in this image increases the processing time by about 20 seconds. More cylinders would increase this still further.

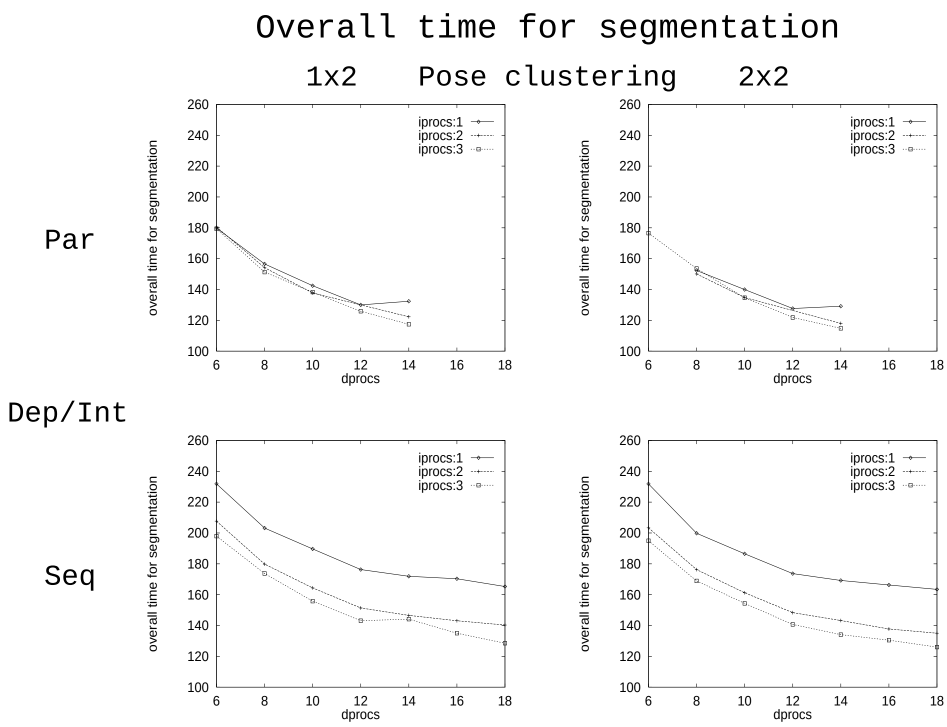
- Increasing the depth processors from 6 to 18 reduces the overall processing time by about 70 seconds. The total time is over 200 seconds for 6 processors and between about 100 and 160 seconds for 18 processors. These are not respectable figures for a modern computer vision system but are typical for the types of vision algorithms employed in this system.

Figure 8.113 presents the speedup figures for the segmentation system. A maximum speedup of 6 without cylinders and 5 including cylinders for a total of 22 processors is modest. The speedup figures for the modules implemented using skeletons are very good (see Section 7.2) but the total overheads to the system are sufficiently large to reduce the speedup figures for the overall system.

For comparison, the overall processing time for the segmentation system is presented for concurrent and consecutive intensity and depth routes in Figure 8.114. This illustrates two points:

- There is a substantial performance gain by running the two routes concurrently. This could be improved still further by reallocating processors when one route terminates.

- The intensity processor allocation does not affect the overall time. This is because it is defined by the maximum processing time between the two routes.



This shows the effect of running the depth and intensity routes consecutively and concurrently. `Par` refers to concurrent depth and intensity, `Seq` refers to consecutive. Only partial data is available for concurrent runs.

Figure 8.114: Image 1: Comparison Between Consecutive/Concurrent Depth and Intensity

Images 2, 3 and 4

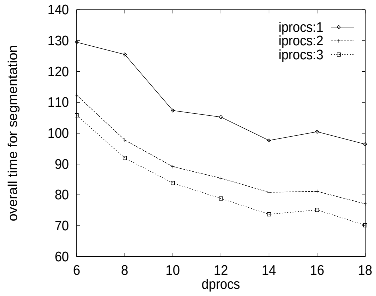
The results for these images are almost identical to those for Image 1. This is perhaps not surprising considering the fact that they are similar to Image 1 in terms of number and size of patches, lighting conditions and image noise. Image 2 is practically identical to Image 1 in terms of performance. Image 3 has more surface noise than Image 1 and this is manifested in slight variations in processing times. Otherwise, the performance with this image closely follows that for Image 1. The performance for Image 4 is slightly different from the other images. The processing times are slightly longer but the speedup is better, peaking at about 7 for 18 depth and 3 intensity processors. This is attributable to the fact that the actual depth data covers a greater percentage of the image, improving the performance of the HKSEG phase.

Images 5, 6 and 7

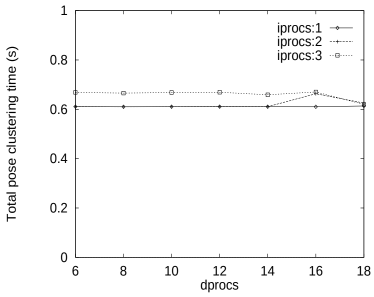
Figure 8.115 shows a summary of the parallel performance for the first synthetic scene. These exclude cylindrical processing and use only the 2×2 array of pose clustering processors.

Comparing these results with the results for Image 1 it can be seen that the overall processing time is less (peaking at about 130 seconds versus 200 seconds) and the speedup is slightly

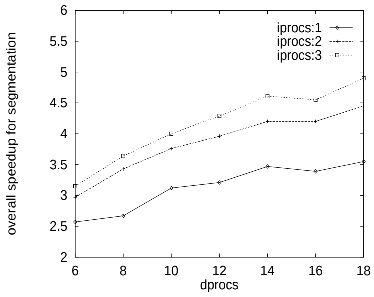
Overall time for segmentation



Overall time for pose clustering



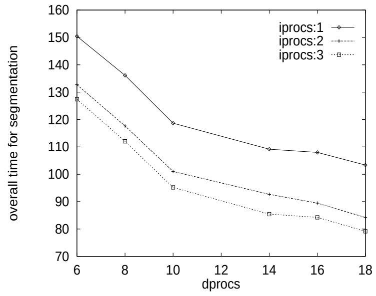
Overall speedup for segmentation



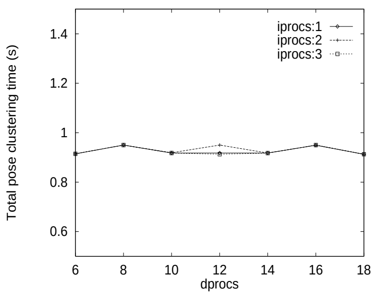
Due to the faceted nature of rendered cylindrical surfaces no cylindrical processing is used. These results are for the 2×2 array of pose clustering processors.

Figure 8.115: Image 5: Parallel Run Time for Segmentation and Pose Clustering and Overall Speedup for Segmentation

Overall time for segmentation



Overall time for pose clustering



Overall speedup for segmentation

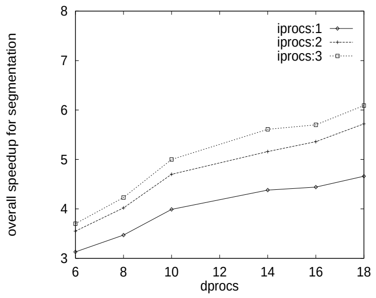


Figure 8.116: Image 6: Parallel Run Time for Segmentation and Pose Clustering and Overall Speedup for Segmentation

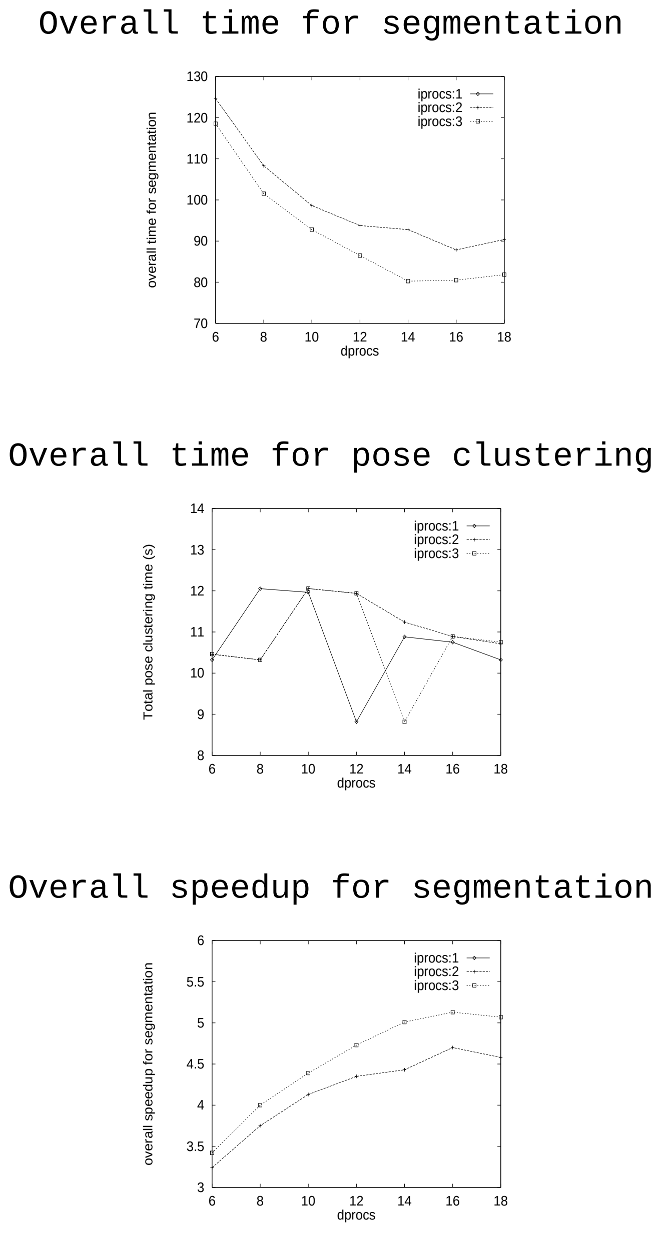


Figure 8.117: Image 7: Parallel Run Time for Segmentation and Pose Clustering and Overall Speedup for Segmentation

less (a maximum of just under 5 as opposed to over 6). These observations can be attributed to the facts that the synthetic image is slightly smaller (192×192 against 170×256) and slightly more sparse than the real images.

Images 6 and 7 exhibit a larger variation in performance compared to Image 5 than was seen in the real images. There is a much greater variation in the amount and quality of data between the synthetic images. Image 7 in particular consists of a larger number of smaller planes than any of the other test images. This is manifested in the pose clustering time which is about 0.6 seconds for Image 5, 0.9 seconds for Image 6 and up to 12 seconds for Image 7.

8.8 Comparison with Other Systems

8.8.1 Parallel Vision Systems

There are few complete vision systems implemented in parallel using methods such as the algorithmic skeletons approach used in this thesis. There are, however, a number of systems for specialised applications mostly using algorithms dedicated to the vision problem implemented on highly optimised (and usually hybrid) hardware. Some of these were reviewed in Section 2.2.

One such system is the PARADOX system by Wang and Brady [84]. It is difficult, however, to make a direct comparison between PARADOX and the system in this thesis. The PARADOX system is more specific and only has one image source. It also uses a hybrid architecture as opposed to the uniform transputer architecture used here. The performance of PARADOX is very good but this is helped by specialised hardware. Another difference is that PARADOX processes continuous sequences of images where the system in this thesis processes one set of multisource image data. The major difference, however, is in the acquisition and transfer of image data via high speed hardware, the lack of provision of any video rate input for the system in this thesis is a serious limitation but not one which invalidates the methods employed.

Another system is MARVIN, by Rygol *et al.* [85, 86]. Again comparisons between MARVIN and the system in this thesis have to be treated with caution. MARVIN bears some resemblance to the PARADOX system above, but is also dependent on specialised hardware to speed up low-level processing and for communication of image data. The combined segmentation and matching are comparable with the system here but use completely different data sources and vision techniques. The central processing cores are very similar in terms of the topology and type of processors but MARVIN is far better organised in terms of handling image data and data locality.

The GDM model of Wang *et al.* [70] bears some resemblance to the GD skeleton used here. Both are based on geometric decomposition of the images and have relatively general communications models (if the variants of the GD skeleton are taken into account). The GDM implementation, however, includes a range of optimisations, such as load balancing and data reorganisation which have to be handled explicitly by the programmer in the simple skeletons methodology used here.

The BDFP model of Alexander *et al.* [74] also has a passing resemblance to a skeleton with an equivalent parallel implementation (in this case the BDPA architecture). Again, however, the emphasis is on optimised parallel performance and most of the implementation issues are related to translating a particular solution to a programming problem into a representation which can be handled by the programming model and attendant architecture. The method described does seem to be very powerful within the domain of digital signal processing and image processing but would have problems with higher level vision tasks.

Perhaps the closest in terms of methodology to the skeletons approach is the vector model of Little *et al.* [80]. This uses a set of primitive operations such as *scan* and *permutation* operations which have implementations on fine-grained parallel machines with acceptable performance penalties. A set of higher-level operations are developed and used to construct a respectable list of vision applications, although actual performance figures are a bit thin. This method complements the development of the individual algorithms in this thesis but does not address the problems of building a complete system.

The Apply and Adapt languages by Webb and others [88, 89, 91] bear a very close resemblance to the work in this thesis. Their implementation of the equivalence-table based connected components algorithm is highly reminiscent of the HKSEG algorithm presented here. It is stated that implementing their algorithm on a non-row-based decomposition would be difficult but the HKSEG algorithm effectively achieves this. Another close correspondence is between the Canny implementation in Apply and the CANNY phase presented here. The Apply algorithm uses a pipelined approach to local window operations whereas the CANNY phase uses the GD skeleton with preloaded data. It is difficult to compare the performance of the two systems without more data than is available in the cited papers but results look comparable. In terms of the overall method, Apply and Adapt concentrate on providing efficient parallel models and using compiler support to manage some of the data placement and architectural decisions whereas the work in this thesis advocates the use of performance prediction as a means of selecting suitable centres of parallelism.

8.8.2 Algorithmic Skeletons

Many of the methods reviewed in Section 2.3 are highly theoretical in nature, many others are not in a sufficient state of development for use in a system such as that constructed here but the influence of previous skeletons work on the current system should not be underestimated.

The most direct influence was that of previous work by Waugh [16, 53, 54] using simple skeletons such as `map` and `filter` to construct a high-level vision application. Another was the work of Bratvold [40] into automated parallelisation of SML programs, including a ray tracing example. This work used a novel method of prototyping which was too detailed for the requirements of this project but was the basis for many of the skeletons ideas which were vital during construction.

The work of several other algorithmic skeletons researchers influenced the design of the skeletons in this system, for example Cole's *task queue* skeleton [11] bears some resemblance to the task farm skeleton here. Another example is the *local window* (LW) skeleton used by Deldarie *et al.* [10] for image processing which is again similar to the GD skeleton and is also used for convolution operations.

8.9 Summary and Conclusions

This chapter presented a brief characterisation of the parallel object recognition system, concentrating on the parallel performance. The following concluding remarks can be made:

1. The four real and three synthetic images were not sufficient to cover all the conditions with which the system could be presented. All the real images represented similar scenes in terms of complexity, ambient conditions, noise and size. The three synthetic images verified some aspects of the system's operation and provided scenes of differing complexity. Synthetic images do not provide a realistic test of the system for practical use, however.

2. The four real images were all very good in terms of completeness of data and general noise. Some areas in Images 2 and 3 had higher levels of noise and these caused problems with the depth segmentation. It was possible to get a good segmentation from these images with careful tuning of the parameters.

3. The three synthetic images comprised from one to three objects in a single scene. These all gave good segmentations with very little difficulty. The orientation of some of the objects prevented matching with the equivalent models.

4. All seven test images gave segmentations of similar quality to other systems [51, 301]. These were sufficient to allow the matching phase to include correct poses in the list of generated poses. These were checked by superimposing the transformed model over the original intensity image.

5. The overall segmentation times (over 200 seconds per image) were too large for the system to be used in a real application. This was due to overcomplexity in the depth processing and the fact that the methods used were computationally intensive. This could be improved by streamlining the depth processing, for instance by directly seeding the growing process from random seeds in the original data as opposed to the HK data.

6. The distribution of processing among and within the modules highlights several sequential overheads which cause a reduction in parallel performance. One of these is cylindrical processing which is implemented sequentially and is computationally intensive. This is particularly true of cylindrical fitting. Other overheads include post-processing of data and data rearrangements. Initialisation of parallel phases was not a significant factor.

7. The overall parallel performance was modest, ranging from speedups of about 5 to 7 for 22 processors for all the test images. The reasons for this were elaborated in Section 7.3.4. This speedup is, however, consistent throughout the experiments for all the images under different processing conditions.

The overall performance of the system is not as good as one might expect considering the performance of the individual algorithms. It should be noted, however, that there is ample room for improvement by implementing some of the sequential overheads in parallel. One immediate improvement would be to implement fitting in parallel, although there are few implementations of the SVD calculation in parallel [302]. Another possibility is to implement cylinder processing in parallel.

Another point is that the system was designed as part of a wider investigation. The possibility of controlling the system and automatically deducing suitable parameters as well as feedback from the matching phase were built into the system. These have led to some of the inefficiencies in the completed system. The main point here is that the memory limitations arose from intermediate data being retained for later processing at all stages.

Chapter 9

Conclusions

This thesis concerned the design and implementation of a complete multisource scene segmentation system in parallel using functional language prototyping. The huge scope of this work has meant that many of the issues involved were not fully considered. Some aspects were dealt with in detail but the principal motivation was to devise and implement a workable solution. Inevitably, compromises were made to achieve this solution.

9.1 Achievements and Contribution

The thesis objectives and a set of subsidiary goals were given in Section 1.3. Achievements are discussed in terms of meeting the subsidiary goals:

1. Considerable effort was put into the initial system design. The motivation for building the system was considered along with other design factors. These included; ease of sequential implementation, ease of parallel implementation, overall memory requirements, suitability for skeleton parallelism, commonality of code and data, suitability for the chosen architecture, interfacing with external code and possibility of external control. A complete specification document was produced Austin [65] using ADTs as informal module specifications.

2. A set of well-studied algorithms were reviewed and suitable candidates selected. Those which admitted simple parallel implementations were given preference. These included convolution-based low-level processing and growing-based intermediate processing. The limitations of these algorithms were known in advance but other project goals (such as ease of parallelism) limited the choice of available algorithms.

3. Algorithmic skeletons were comprehensively reviewed. Although much of this work is

theoretical it strongly influenced the design and implementation of the parallel system. Many of the ideas from skeletons work were incorporated in a complete system.

4. A partial classification of vision algorithms was developed with working SML examples. This was to prove the usefulness of functional languages in developing model-based, multi-source 3D object recognition algorithms. The programs written here showed that there were no major problems in this respect. Indeed the speed with which they were constructed proves that functional languages have much to offer developers of such programs.

5. A methodology for the construction of parallel algorithms was developed.

   (a) A set of skeletons for parallel model-based, multi-source 3D object recognition was retrospectively devised from the parallel system. These were geometric decomposition (GD), data farm (DF), task farm (TF) and GD with interworker communications (GDIW). These skeletons are specific to the system developed here. It is not possible to say how many would be required for general computer vision but it is certain that the four skeletons developed would be useful in parallel vision work.

   (b) All of the skeletons were implemented in both SML and in Occam2 and the performance prediction methodology adapted in each case. The GD and DF skeletons were straightforward both for implementation in SML and for instrumentation and performance prediction. The TF and GDIW skeletons presented some difficulties, these required imperative features in the SML prototype and the performance modelling did not exactly reflect the behaviour of the parallel equivalent. Reasonable predictions were obtained in both cases, however, given some restrictions upon the instantiation.

   (c) A method of predicting the performance of a parallel implementation from a functional language prototype was devised. This is a "cheap and cheerful" method which uses numerical analysis methods to summarise the performance of a dummy skeleton construct. This construct and the numerical description are parametrised according to the values that can be measured from the prototype. Accurate predictions were achieved in a small number of cases but the method is too application dependent for general purpose use.

   (d) Five parallel modules were implemented using the methodology. The main points here are that the skeletons were used directly to construct each module (some used more than one skeleton in combination) and all of these modules gave adequate parallel performance.

(e) The methodology itself was not evaluated thoroughly. Its use in the construction of the complete system was, however, a good indication of its practicality. The main features which stood out were the ease of use and the parallel performance achieved.

6. The methodology was applied in the construction of a complete system and fitted in well with the overall system design. The complete system was not fully prototyped but a complete working prototype with some of the final features missing (such as cylinder identification) was built and run on one image. The completed parallel system was based very closely on the prototype, especially in the sources of parallelism for the core modules.

7. The parallel modules were composed into an integrated system using suitable interface components such as data rearrangements and other post-processing operations. Some of these components proved to be computationally or communications intensive. This led to modest performance for the overall system. The requirements of the specification are satisfied, however, and there is plenty of room for improvement by identifying areas which could be implemented in parallel. This could be done using the prototyping and performance modelling method.

8. The completed system was tested on four real registered images and three synthetic images which were generated in a registered form. The parallel performance was evaluated. Although this was not a thorough evaluation the basic characteristics were determined. The parallel performance of the overall system was found to be consistent but modest for the seven images tested here.

This work has brought together several different areas of research into a single project. The contributions of this project are:

- Functional prototyping *for parallelism* has not previously been employed in such a large project. This has led to important lessons in the development of parallel model-based, multi-source 3D object recognition algorithms in functional languages.

  1. Handling of 2D or 3D data requires different techniques from lists which are commonly found in parallel functional languages and applications. This mostly concerns data access in the prototype and data placement in the parallel version.
  2. Many of the issues in the parallel implementation can be addressed at the prototype stage. This is true of functional prototyping in general but is particularly useful for

parallel vision applications which can be both communication and computationally intensive.

3. In a modular system, parallelisation of individual components does not in itself lead to good overall performance. Care has to be taken to analyse the system as a whole, for instance spotting and analysing sequential overheads between the phases.

- A method for the design and implementation of a system for the segmentation of multisource data has been developed. The algorithms to which the methodology has been applied are not very sophisticated by present standards but a *complete* system has been implemented in *parallel* for multisource data. The overall parallel performance is less than might be expected given the performance of the individual phases but the system as it exists could be improved in future developments.

9.2 Limitations and Discussion

There are several points about what has been achieved with the parallel system and the methods used to construct it:

- The methodology of prototyping in a functional language, predicting the performance using numerical analysis techniques and translating into Occam2 has limitations. There are many issues in developing vision algorithms in a functional language that have to be clarified, particularly with regards to non-associative `fold` operations and the use of arrays. This was particularly apparent in several early versions of the edge tracking routine. Program transformation also has to be investigated in this context. Numerical performance modelling works well within restricted domains but has not been tested comprehensively enough to assess its accuracy for general use. The translation process from SML to Occam2 has caused too many problems for it to be proposed as a suitable method for general parallel program development.

- All the examples in this thesis were hand-coded in parallel. This has limited what could be done to optimise the system's performance. The whole methodology needs to be supported by programming tools, possibly even a semi-automatic parallelising compiler. The modest performance which resulted from incomplete parallelisation stems directly from this and indicates the importance of support software for development of such systems. Unfortunately, the development of such tools is a major exercise in its own right and there was no scope in the present work to attempt this. An automated parallelising compiler is presently under development which incorporates many of the methods proposed in this thesis [296].

- There were many problems with system resource allocation. This has restricted the size of the images that could be processed. It also constrained the types of parallelism that could be exploited. Part of the reason for this is that the memory model was too static, for instance planes are stored in a 2D array of planes of points. A more dynamic memory allocation would have made code development harder. Since all coding was by hand it seemed prudent to keep a memory model which was "natural" to Occam2. Exploiting different kinds of parallelism, for instance using distributed data rather than duplicated data across processors would alleviate this problem. This would again cause coding difficulties since the algorithms would need more internal synchronisation.

- The processing times for object recognition are not small enough for any real applications to be built using this system. This is a result of the algorithms chosen and the incomplete parallelism rather than any fundamental flaw in the methods used. Similarly the quality of segmentation is not reliable enough to be used in a situation where there were variations in ambient conditions or recognition of arbitrary lists of objects was required. The system developed here, however, is more generic than would be required in many applications. A specific application may only require one particular setup or even part of the system to perform useful functions. More seriously, the reliance upon registered range and intensity data could be a problem with real vision applications. Although 3D imaging systems are commercially available, there are many technical difficulties in generating such registered images which could cause problems during commissioning and maintenance in an industrial setting.

The overall conclusion is that the system has not been fully developed. There are many areas which could be improved on or be the basis for future developments.

9.3 Future Work

There are many promising aspects of this work which could be developed further:

- The most immediate priority with the system at present is to test it on a wider range of images, both synthetic and real. This will better characterise areas which could be reworked or rewritten. Some debugging work will probably result from this since pushing the system to its limits is bound to expose software faults. It may even be desirable to rework some components. For instance, the depth route could be simplified by scrapping the HK segmentation altogether and replacing it with a random sampling process.

- Once the system is fully characterised it would be desirable to implement some of the automatic parameter control features that were planned as part of the original project specification but not implemented. This would involve online reprocessing of both depth and intensity data, either independently or by commands from the pose clustering. The system is set up to do this but there will be substantial new communication code in the full implementation. The full characterisation of the system would be the basis for system identification to suggest control methods. Due to the unpredictable nature of some of the variables it is likely that a set of *production rules* or an *expert system* would work better than simple PID (proportional, integral, derivative) control or adaptive control systems.

- Sensor fusion is another aspect of the original specification which was not implemented. This is due to the fact that there are no simple discrete algorithms for fused data as there are for single source data. Fusion could occur at low-level, for instance intensity-guided depth segmentation, or at intermediate-level where planes and lines are matched between images, prior to model matching. High-level fusion already exists since pose clustering implements a uniform cluster space for depth and intensity data.

- The methodology needs more work. Numerical performance modelling worked well but needs to be tested for more general use. This would involve characterising several basic skeletons over the *full range* of their parameters, possibly splitting the performance predictors up into different parameter domains. An interesting idea would be to build the model predictors into the skeleton implementations so they could initialise the parameters prior to or during a program run. It is also essential that an efficient translation from prototype code to parallel implementation is developed. Even just a translator from sequential prototype code to sequential implementation code would drastically speed up development. Libraries of skeletons could then be instantiated with this code to generate parallel implementations.

- The handling of algorithmic skeletons needs to be made more explicit and user friendly. A graphical interface allowing the controlled selection and instantiation of a library of skeletons should be provided with full documentation and performance information. The literature is full of skeletons which are very similar to one another and these probably represent a fair amount of duplication of work which could be alleviated by a central depository for known skeletons and their characteristics.

Appendix A

SML Library definitions

A.1 Introduction

This appendix outlines some of the SML routines commonly used throughout the thesis, these are arranged in lexicographic order within each library, correct compilation of these requires a different order since they are interdependent.

A.2 Library Structures

The libraries are in Standard ML `structure` format.

A.2.1 `Gen`

```
fun t1f2 (a,b) = a
fun t2f2 (a,b) = b
...
```

In general `tXfY` extracts the `X`th value from a tuple of `Y` elements.

A.2.2 `Lists`

```
fun flatten nil = nil
  | flatten (h::t) = h @ (flatten t)
```

`flatten` takes a list of lists and returns a list comprised of the inner lists appended in order.

```
fun fxlist _ 0  _ = nil
  | fxlist x xn ff = (ff x)::(fxlist (x + 1) (xn - 1) ff)
```

`fxlist` generates a list by applying the function `ff` to each integer from `x` to `x+xn`.

```
fun head _ [] = []
  | head 0 _ = []
  | head n (h::t) = h::(head (n - 1) t)
```

`head` extracts the first `n` elements from the head of a list.

```
fun ilist 0 _ = nil
  | ilist n init = init::(ilist (n - 1) init)
```

`ilist` generates a list of length `n` of value `init`.

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

`map` applies a function to each element of a list.

```
fun map2 ff (h1::t1) (h2::t2) = (ff h1 h2)::(map2 ff t1 t2)
  | map2 _ _ _ = nil
```

`map2` maps a function over two lists simultaneously, the function is a binary function and is passed one element from each list.

```
fun mapixy _ _ nil = nil
  | mapixy ff x (h::t) = (ff x h)::(mapixy ff (x + 1) t)
```

`mapixy` maps a function over a list but passes the position of the element in the list along with the element to the function.

```
fun map2ixy _ _ _ nil = nil
  | map2ixy _ _ nil _ = nil
  | map2ixy ff x (h1::t1) (h2::t2) =
    (ff x h1 h2)::(map2ixy ff (x + 1) t1 t2)
```

`map2ixy` combines the functionality of `map2` and `mapixy`.

```
fun seg _ _ [] = []
  | seg 0 0 _ = []
  | seg 0 e (h::t) = h::(seg 0 (e - 1) t)
  | seg s e (h::t) = seg (s - 1) e t
```

`seg` extracts a segment, start `s`, length `e` from a list.

```
fun tabulate _ nil = nil
  | tabulate a (h::t) = (a,h)::(tabulate a t)
```

`tabulate` creates a list of tuples with first element `a` and second element from a list.

```
fun tail _ [] = []
  | tail 0 l = l
  | tail n (h::t) = tail (n - 1) t
```

`tail` removes the first `n` elements from the head of a list.

```
fun unzip nil = (nil,nil)
  | unzip ((h1,h2)::t) =
    let
      val (t1,t2) = unzip t
    in
      (h1::t1,h2::t2)
    end
```

`unzip` takes a list of tuples and returns a tuple of lists.

```
fun zip (h1::t1) (h2::t2) = (h1,h2)::(zip t1 t2)
  | zip _ _ = nil
```

`zip` takes two lists and creates a list of tuples from equivalent elements between the two lists.

A.2.3 LLists

```
fun fxylist _ _  _  0 _ = nil
  | fxylist x xn y yn ff =
    (Lists.fxlist x xn (ff y))::(fxylist x xn (y + 1) (yn - 1) ff)
```

`fxylist` generates a list of lists from `(x,y)` to `(x+xn,y+yn)` by applying the function `ff` to the `(x,y)` index of each element.

```
val mapll = map o map
```

`mapll` is a `map` over a list of lists.

```
fun map2ixyll _ _ _ _ nil = nil
  | map2ixyll _ _ _ nil _ = nil
  | map2ixyll ff x y (h1::t1) (h2::t2) =
    (map2ixyll' ff x y h1 h2)::(map2ixyll ff x (y + 1) t1 t2)

and map2ixyll' _ _ _ _ nil = nil
  | map2ixyll' _ _ _ nil _ = nil
  | map2ixyll' ff x y (h1::t1) (h2::t2) =
        (ff (x,y,h1,h2))::(map2ixyll' ff (x + 1) y t1 t2)
```

`map2ixyll` is a version of `map2`, a map over two lists of lists, maintaining an (x, y) index for each element which is passed into the application function.

```
fun padll def wd img =
  let
    fun zerol x = Lists.ilist x def
    fun zeroll x y = Lists.ilist y (Lists.ilist x def)
    val tb = zeroll ((length (hd img)) + (2 * wd)) wd
    fun padline line = (zerol wd)@line@(zerol wd)
  in
    tb@(map padline img)@tb
  end
```

`padll` grows a list of lists in all directions by `wd` elements, filling in the blank elements with value `def`. The inverse operation is `unpadll`.

```
fun segll (s1,e1) (s2,e2) ll = map (Lists.seg s1 e1) (Lists.seg s2 e2 ll)
```

`segll` extracts a 2D segment from a list of lists, top left is `(s1,s2)`, dimensions are `(e1,e2)`.

```
fun shiftll def xs ys img =
  let
    fun zeroll x y = Lists.ilist y (Lists.ilist x def)
    fun shiftline xs line =
      if xs = 0
        then line
        else
          if xs < 0
            then (Lists.tail (~xs) line)@(Lists.ilist (~xs) def)
            else (Lists.ilist xs def)@(Lists.head ((length line) - xs) line)
    val xsimg = map (shiftline xs) img
    val xsz = length (nth(img,0))
  in
    if ys = 0
      then xsimg
      else
        if ys < 0
          then (Lists.tail (~ys) xsimg)@(zeroll xsz (~ys))
          else (zeroll xsz ys)@(Lists.head ((length img) - ys) xsimg)
  end
```

`shiftll` shifts a list of lists by `(xs,ys)` where `xs` and `ys` can both be positive (right shift) or negative (left shift). The blank values are padded with the value `def`.

```
fun tabulatell nil _ = nil
  | tabulatell (h::t) l1 = (Lists.tabulate h l1)::(tabulatell t l1)
```

`tabulatell` is effectively the outer product in `list list` format of two lists.

```
fun unpadll wd img =
  let
    fun unpadline wd line = Lists.seg wd ((length line) - (2 * wd)) line
  in
    map (unpadline wd) (Lists.seg wd ((length img) - (2 * wd)) img)
  end
```

`unpadll` shrinks a list of lists in all directions by `wd` elements, this is the inverse of `padll`.

```
fun zipll (h1::t1) (h2::t2) = (Lists.zip h1 h2)::(zipll t1 t2)
  | zipll _ _ = nil
```

`zipll` takes two lists of lists and generates a list of list of tuples.

```
fun unzipll nil = (nil,nil)
  | unzipll (h::t) =
      let
        val (l1,l2) = unzip h
        val (t1,t2) = unzipll t
      in
        (l1::t1,l2::t2)
      end
```

`unzipll` takes a list of lists of tuples and generates a tuple of lists of lists.

Appendix B

Description of control variables

Parameters can be of several types; `BOOL` which takes value `TRUE` or `FALSE`; `INT`; `REAL32` and `REAL64` (all in standard Occam2 syntax). The variables are declared in fold `System parameters` of file `common1.tsr`. The default values listed here are those in use at the present time but may be changed by altering the `setup defaults/system parameters` folds in file `psmon.tsr` and recompiling.

B.1 General Control

Name	Type	Default	Description
Bcast	`BOOL`	`TRUE`	When `TRUE` broadcast message will be sent using `tiny`'s broadcast facility, when `FALSE` broadcast messages will be sent by point to point messages, looping round all processors. This was to overcome early problems with broadcast messages and is currently redundant.
Print	`BOOL`	`TRUE`	When `TRUE` progress reports will be printed out during the course of calculation. This can be voluminous but can be saved by `rian`. When measuring performance results with the `Perf` flag, this parameter should be set to `FALSE`.
Perf	`BOOL`	`TRUE`	When `TRUE` performance measurements are printed out at the end of each run. Note that performance results for most of the modules are normally compiled out of the program, see the `performance printouts` fold in `PROC RunPs` of file `psmon.tsr`.
Debug	`BOOL`	`FALSE`	This controls low-level debug messages and should be set to `FALSE`.
Loads	`BOOL`	`FALSE`	When `TRUE` the segmentation results will be read directly from the segmentation file (see `files` command) and will be sent directly to the pose clustering software.

B.2 HK Calculation

Name	Type	Default	Description
HKN	`INT`	`7`	The size (in both X and Y dimensions of the convolution window. This must be an odd number and is forced to the next odd number if an even number is given.
Hthresh	`REAL32`	`0.006`	This is the zero threshold for the mean curvature calculation, note that it is separate from the Gaussian curvature threshold.
Kthresh	`REAL32`	`0.0006`	The zero threshold for the Gaussian curvature calculation, this should be about $\texttt{Hthresh}^2$ but the values shown here work quite well in practice.

B.3 HKSEG Segmentation

Name	Type	Default	Description
HKSEGN	`INT`	`7`	The size of the window for median smoothing of the HK sign map for use in the connected components algorithm.
maxDepthDiscrepancy	`REAL32`	`10.0`	When growing a patch during the connected components algorithms, pixels separated in depth by more than this value are assumed not to be compatible even if they have the same HK sign code.
minDepth	`REAL32`	`140.0`	The minimum depth for valid data, not of any value but should remove really bad rogue points, if any are present.
maxDepth	`REAL32`	`10.0`	The maximum depth for valid data. This was to exclude baseplane data but is not really used any more since the baseplane data can be removed by `rim`.
minPatchSize	`INT`	`500`	This value is used throughout the depth processing routines. Once a complete patch is available, either by connected components, by merging patches or by growing and fitting, it is discarded if it contains fewer points than this parameter. **Note** that this parameter is critical to the successful operation of the depth segmentation, it is also short circuited temporarily at various points in the program to allow aggregation of small patches, where possible.
FitThL	`REAL64`	0.1	This value is currently used by the HKSEG postprocessing operation to decide whether to keep or discard a planar patch. If the fit residual is greater than this value then the patch is discarded.
FitThH	`REAL64`	5.0	This is currently used as a keep/discard threshold on the fit residual for cylindrical patches.

B.4 RG2 Region-growing

Name	Type	Default	Description
NumSamples	INT	10	The number of samples to take from each HK patch to use as a seed for the growing process.
SampleSize	INT	100	The size of each sample to take during patch sampling. If a patch is less than this size then the entire patch is used as a single sample.
SampleWindow	INT	10	This parameter is obsolete.
SampleFactor	INT	4	This parameter is used to divide the NumSamples value for each patch to give sufficient parallelism and does not affect the program output.
GrowThreshP	REAL64	0.5	This is the threshold on the residuals used to determine compatible points for growing planar patches. This is a critical parameter for the region-growing process.
GrowThreshC	REAL64	4.0	This is the growing threshold for cylindrical patches although these patches are not currently grown due to problems with growing into neighbouring patches, at present the entire patch from the HK segmentation is used.
FitThL	REAL64	0.1	This is the lower fit threshold for determining whether a fit equation matches the point data. If the χ^2 value is less than this then a good fit is assumed, above this a higher order fit is attempted. **Note** that at present the switch to a higher order fit is disabled, all patches are tested separately for cylindricity prior to the growing process. This value is also somewhat arbitrary and is probably very different for different images.
FitThH	REAL64	5.0	This is the upper fit threshold. Above this value any data is discarded. At present this is not implemented.
Xdivs	INT	30	The region-growing algorithm currently operates by dividing the image up into a large number of geometrically defined subimages with no overlap. Growing then occurs within each subimage, spawning futher tasks when growing proceeds over a geometric boundary. This value is the number of divisions in the X dimension and is effectively a granularity control for the growing process.
Ydivs	INT	30	The number of divisions in the Y dimension.

B.5 CANNY Edge Detector

Name	Type	Default	Description
CannyN	`INT`	`7`	This is the size of the convolution window used during smoothing and edge enhancement.
CannyLow	`INT`	`10`	A lower threshold on the output edge strength data, a point is ignored if it has an edge strength lower than this value.
CannyHigh	`INT`	`255`	An upper threshold on the output edge strength value, values above this are set to this value.
CannySigma	`REAL32`	`1.0`	The sigma value for the Gaussian and derivative of Gaussian masks used during smoothing and edge enhancement.
CannyScale	`REAL32`	`20.0`	The calculated edge strength values are multiplied by this value prior to limiting with the `CannyHigh` value, this effectively operates as a primitive scaling factor for the intensity values.

B.6 ET Edge Tracking

The edge tracking routine operates by tracking within subimages, followed by merging of partial lines. The merging of partial lines is also defined by the following parameters which apply to the basic tracking process. If two lines have neighbouring points which are compatible by the growing metric then they are merged. Note also that the `Segment` routine used by the Hough transform process is also used to post-process edge tracked lines.

Name	Type	Default	Description
EdgeThresh	`INT`	`200`	The edge strength threshold for the edge tracking process, only pixels with edge strength greater than this value are considered.
AngleThresh	`INT`	`10`	Two neighbouring pixels are considered compatible if their edge orientation are within this threshold of each other (angles run from 0 to 255).
ETportions	`INT`	`25`	Like the region-growing algorithm the edge tracking routine achieves parallelism by dividing the image up into a set of rectangular subimages and runs the sequential algorithm within each image. This parameter defines the total number of subimages.

Appendix C

Algorithmic Details

This appendix adds some extra detail to the algorithms used in the parallel segmentation system.

C.1 Regional Segmentation

Here, the regional segmentation algorithms are described. The initial image smoothing phase is discussed, then the calculation of the mean and Gaussian curvature and then the curve fitting using the SVD method.

C.1.1 Image Smoothing

Nearly all range image segmentation algorithms are preceded by some method of smoothing due to the noise generated by range sensors. Simple Gaussian smoothing has been used in the past [303, 304] but the problem with range image noise is that it is often not Gaussian in nature. The problem is quite often due to spikes at discontinuities, particularly with laser-based sensors. The effect of Gaussian smoothing on such data is to spread the effect of the noise across a wider area giving poorer results than the underlying data are capable of producing.

One alternative is the adaptive smoothing filter of Saint-Marc and Medioni [305]. In this method the first derivative is iteratively smoothed by a monotonically decreasing function of probability of discontinuity. This method has the advantage of being applicable to both range and intensity images but has the disadvantage of the loss of spatial information. Re-integration to get the spatial information is likely to introduce additional errors.

The most desirable method for the parallel system is probably the iLMedS algorithm [49] which is an improved version of the Least Median of Squares method (LMedS) [306]. This method removes outliers, notably the spikes introduced by the range sensor, and is thus a statistically robust method. With robust smoothing it may not be necessary to introduce robust fitting methods in later processing given sufficiently good underlying data. The other advantage of this method is that it can easily be implemented in parallel using geometric decomposition and an existing implementation is available in parallel C. For real data a following Gaussian filter with a small σ value would also be applied to help with any residual statistical noise.

Another possibility is anisotropic diffusion smoothing [51, 307]. This behaves better near the boundaries of patches in the data but is computationally intensive. In practice, both types of smoothing were used, sometimes both together. The LMedS method allowed interpolation and removal of outliers whereas the diffusion smoothing worked better with surface noise.

C.1.2 Coarse Segmentation by Signs of Mean and Gaussian Curvature

Mean and Gaussian Curvature

This section is a summary of [247] and [269].

The segmentation process starts with a coarse segmentation based on the mean H and Gaussian K curvatures. These are based on results from differential geometry which describes surfaces (or curves) in terms of the properties of the differentials of the surfaces rather than their position in space. The general parametric form of a surface is:

$$S = \Big\{(x, y, z) : x = d(u, v), y = e(u, v), z = f(u, v), (u, v) \in \mathbf{R}^2\Big\} \tag{C.40}$$

where u and v define a parameter space and $\mathbf{R}$ is the set of real numbers. For depth images, this form can be simplified as $x = u$, $y = v$ and $z = f(u, v)$, called graph surfaces or Monge patches.

There are two mathematical constructs used to describe such surfaces in differential geometry, one is the First Fundamental Form:

$$I(u, v, du, dv) = d\mathbf{x} \cdot d\mathbf{x} = \begin{bmatrix} du & dv \end{bmatrix} \begin{bmatrix} g_{11} & g_{12} \\ g_{21} & g_{22} \end{bmatrix} \begin{bmatrix} du \\ dv \end{bmatrix} \tag{C.41}$$

where:

$$
\begin{aligned}
g_{11} &= E = \mathbf{x}_u \cdot \mathbf{x}_u \\
g_{12} = g_{21} &= F = \mathbf{x}_u \cdot \mathbf{x}_v \\
g_{22} &= G = \mathbf{x}_v \cdot \mathbf{x}_v
\end{aligned} \tag{C.42}
$$

$$
\begin{aligned}
\mathbf{x}_u &= \frac{\partial \mathbf{x}}{\partial u} \\
\mathbf{x}_v &= \frac{\partial \mathbf{x}}{\partial v}
\end{aligned} \tag{C.43}
$$

and $\mathbf{x}$ is a unit vector in the tangential plane at (u, v). This measures the rate of change of $|d\mathbf{x}|^2$ with $d\mathbf{u}$. The Second Fundamental Form is:

$$
II(u, v, du, dv) = -d\mathbf{x} \cdot d\mathbf{n} = \begin{bmatrix} du & dv \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} \begin{bmatrix} du \\ dv \end{bmatrix} \tag{C.44}
$$

where:

$$
\begin{aligned}
b_{11} &= L = \mathbf{x}_{uu} \cdot \mathbf{n} \\
b_{12} = b_{21} &= M = \mathbf{x}_{uv} \cdot \mathbf{n} \\
b_{22} &= N = \mathbf{x}_{vv} \cdot \mathbf{n}
\end{aligned} \tag{C.45}
$$

$$
\begin{aligned}
\mathbf{x}_{uu} &= \frac{\partial^2 \mathbf{x}}{\partial u^2} \\
\mathbf{x}_{uv} &= \frac{\partial^2 \mathbf{x}}{\partial u \partial v} \\
\mathbf{x}_{vv} &= \frac{\partial^2 \mathbf{x}}{\partial v^2}
\end{aligned} \tag{C.46}
$$

and $\mathbf{n}$ is the unit normal to the surface at (u, v). This measures how the normal changes with respect to the position on the surface.

Note that these quantities can be defined (and calculated) by the partial derivatives at the surface and can also be defined in terms of the $\mathbf{G} = g_{ij}$ and $\mathbf{B} = b_{ij}$ fundamental form matrices. The E, F, G, L, M and N Gaussian notation is also common in the literature. Note also that if $d\mathbf{n}$ and $d\mathbf{x}$ are aligned for a particular $d\mathbf{u}$ then this defines a principal direction and that the extrema of the normal curvature function defined by $II(u, v, du, dv)/I(u, v, du, dv)$ are in this direction and define the principal curvatures κ_1 and κ_2.

The six functions in the two fundamental forms characterise a surface uniquely but it is desirable to summarise this information in a more convenient form. The matrix product $\mathbf{G}^{-1}\mathbf{B}$ is called the Weingarten mapping, the Gaussian curvature K is the determinant of this matrix:

$$
K = det\left(\mathbf{G}^{-1}\mathbf{B}\right) \tag{C.47}
$$

The mean curvature H is half the trace:

$$H = \frac{1}{2} tr \left(\mathbf{G}^{-1} \mathbf{B} \right) \tag{C.48}$$

These are the natural scalar algebraic invariants of the surface and, when used in combination, can be used to classify surfaces. One of the most important properties of these values is that they are invariant to the (u, v) parameterisation (provided the Jacobian remains non-zero) and are invariant under rotation and translation and thus view independent.

Due to the difficulty of accurately determining differentials (particularly second-order differentials) from noisy, discrete representations of surfaces, the absolute values of H and K obtained from such data are not reliable enough to do matching. The signs (+,-,0), however, of the H and K curvatures are accurate enough to classify a surface point into one of eight types (K cannot be positive when H is zero); peak ($H < 0$, $K > 0$), flat ($H = K = 0$), pit ($H > 0, K > 0$), minimal ($H = 0, K < 0$), ridge ($H < 0, K = 0$), saddle ridge ($H < 0, K < 0$), valley ($H > 0, K = 0$) and saddle valley ($H > 0, K < 0$).

Evaluating Mean and Gaussian Curvature

There are a number of definitions of the mean and Gaussian curvatures that can be used to compute these curvatures. Some are more useful for calculation from parametric equations for surfaces, others are more useful for estimating the curvatures from range data.

Probably the most useful one for range data is the definition of the curvatures in terms of the elements of the fundamental form matrices:

$$K = \frac{det(b)}{det(g)} = \frac{b_{11} b_{22} - (b_{12})^2}{g_{11} g_{22} - (g_{12})^2} = \frac{LM - N^2}{EG - F^2} \tag{C.49}$$

$$H = \frac{g_{11} b_{22} + g_{22} b_{11} - 2 g_{12} b_{12}}{2 \left(g_{11} g_{22} - (g_{12})^2 \right)} = \frac{EN + GL - 2FM}{2 \, (EG - F_2)} \tag{C.50}$$

Since range data is a graph surface of the form $\mathbf{x}(u, v) = [u \, v \, f(u, v)]^T$ the derivatives are quite easily calculated:

$$\mathbf{x}_u = [1 \, 0 \, f_u]^T \quad \mathbf{x}_v = [0 \, 1 \, f_v]^T \tag{C.51}$$

$$\mathbf{x}_{uu} = [0 \, 0 \, f_{uu}]^T \quad \mathbf{x}_{vv} = [0 \, 0 \, f_{vv}]^T \quad \mathbf{x}_{uv} = [0 \, 0 \, f_{uv}]^T \tag{C.52}$$

$$\mathbf{n} = \frac{1}{\sqrt{1 + f_u^2 + f_v^2}}[-f_u - f_v \, 1]^T \tag{C.53}$$

These definitions can be used in conjunction with the dot product definitions of the matrix elements (Equations C.42 and C.45) to give:

$$\begin{array}{lll} g_{11} = 1 + f_u^2 & g_{12} = f_u f_v & g_{22} = 1 + f_v^2 \\ b_{11} = \frac{f_{uu}}{\sqrt{1+f_u^2+f_v^2}} & b_{12} = \frac{f_{uv}}{\sqrt{1+f_u^2+f_v^2}} & b_{22} = \frac{f_{vv}}{\sqrt{1+f_u^2+f_v^2}} \end{array} \tag{C.54}$$

Using these definitions for the matrix elements allows the curvatures to be expressed in terms of the differentials of f, the graph surface definition function:

$$K = \frac{f_{uu} f_{vv} - f_{uv}^2}{(1 + f_u^2 + f_v^2)^2} \tag{C.55}$$

$$H = \frac{f_{uu} + f_{vv} + f_{uu} f_v^2 + f_{vv} f_u^2 - 2 f_u f_v f_{uv}}{2(1 + f_u^2 + f_v^2)^{3/2}} \tag{C.56}$$

These equations reduce the problem of evaluating H and K to evaluating the surface partial derivatives. These could, for example, be calculated using numerical differentiation. A much simpler method, proposed by Besl and Jain [269], makes use of masks to perform a local, quadratic least squares fit within a small $N \times N$ window. Simple equations are then used to extract the partial derivatives.

An $N \times N$ mask, with N odd, indexed by values from $u = -(N - 1)/2$ to $u = (N - 1)/2$, is defined by the normalised polynomials:

$$b_0(u) = \frac{1}{N} \tag{C.57}$$

$$b_1(u) = \frac{3}{M(M + 1)(2M + 1)} u \tag{C.58}$$

$$b_2(u) = \frac{1}{P(M)} \left(u^2 - \frac{M(M + 1)}{3} \right) \tag{C.59}$$

where $M = (N - 1)/2$ and:

$$P(M) = \frac{8}{45} M^5 + \frac{4}{9} M^4 + \frac{2}{9} M^3 - \frac{1}{9} M^2 - \frac{1}{15} M \tag{C.60}$$

Using these particular masks, the coefficients of the quadratic fit a_{ij}, $(0 \leq i, j \leq 2)$ can be calculated from:

$$a_{ij} = \sum_{u,v=-M}^{M} f(u,v) b_i(u) b_j(v) \tag{C.61}$$

and hence the partial derivatives from:

$$f_u = a_{10} \quad f_v = a_{01} \quad f_{uu} = 2a_{20} \quad f_{uv} = a_{11} \quad f_{vv} = 2a_{02} \tag{C.62}$$

Since the expression for the coefficients is separable in u and v each partial derivative can be calculated by a separate convolution.

There are several disadvantages of this approach. Firstly, the least squares fit is unweighted which could lead to problems with outliers and will cause the operator to break down at discontinuities. Secondly, a different fit is applied at each point on the surface with no continuity constraints. Thirdly, the window size could be an important parameter without obvious means to guide the size selection.

In addition to providing a means of calculating the mean and Gaussian curvature, the partial derivatives can also be used to calculate other, potentially useful, properties:

1. Critical points. These are first or second-order discontinuities on the surface, such as ridges or ramp edges. They can be calculated quite simply by analysing the zero crossings of the first derivative and can be combined with the mean and Gaussian curvatures to improve the coarse segmentation achieved by $H - K$ classification.

2. Metric determinant. This is the square root of the determinant of the first fundamental form matrix:

$$\sqrt{g} = \sqrt{EG - F^2} = \sqrt{1 + f_u^2 + f_v^2} \tag{C.63}$$

   This can be summed over a surface to give the surface area and can also be thresholded to give an edge map.

3. Quadratic variation. This is defined by:

$$Q = f_{uu}^2 + 2f_{uv}^2 + f_{vv}^2 \tag{C.64}$$

   This can be integrated over a surface to give a measure of the flatness of a region.

C.1.3 General Multidimensional Linear Least Squares Fitting by Singular Value Decomposition

The following section is primarily a summary of chapter 15 of the the Numerical Recipes book [76].

The general least squares method can be summarised as follows, assume there are N data points (x_i, y_i), $i = 1, \ldots, N$ and a model $y(x; a_1, \ldots, a_M)$ with M adjustable parameters a_j, $j = 1, \ldots, M$. The least squares fitting method to find the best set of parameters a_j to fit the data points (x_i, y_j) under the given model minimises:

$$\sum_{i=1}^{N} [y_i - y(x_i; a_1, \ldots, a_M)]^2 \tag{C.65}$$

with respect to a_j.

The Gaussian distribution assumption has problems with certain types of low-probability events ("tail" events) which are given lower probabilities than they deserve. Range and intensity data are likely to suffer from this type of anomaly. However, since this phenomenon tends to manifest itself as large numbers of outliers, robust methods can be called into play if there are such effects during image processing. One point to note is that fitting only removes statistical errors, not systematic ones such as calibration errors.

One immediate generalisation is the relaxation of the "constant σ" assumption, leading to weighted least squares or chi-squared fitting:

$$\chi^2 = \sum_{i=1}^{N} \left[\frac{y_i - y(x_i; a_1, \ldots, a_M)}{\sigma_i}\right]^2 \tag{C.66}$$

Note that this is not to be confused with the chi-squared distribution, which assumes linearity in $a_1, \ldots, a_M$. If the σ_i values are not known in advance then an arbitrary single value for $\sigma_i = \sigma$ can be assumed, χ^2 can be minimised and then σ recomputed using:

$$\sigma^2 = \sum_{i=1}^{N} \frac{[y_i - y(x_i)]^2}{N - M} \tag{C.67}$$

This, however, assumes a good fit between the model and the data and so prohibits any goodness-of-fit assessment. For the image data to be analysed, here, it is hoped that reasonably reliable estimates of error are available from the sensor.

The generalised problem is obtained by differentiating Equation C.66, giving the M equations:

$$\sum_{i=1}^{N} \left(\frac{y_i - y(x_i)}{\sigma_i^2} \right) \left(\frac{\partial y(x_i; \ldots, a_k, \ldots)}{\partial a_k} \right) = 0 \tag{C.68}$$

where $k = 1, \ldots, M$.

One potential problem with image data is that both the dependent and independent variables are subject to errors. In this case, Equation C.66 should strictly be:

$$\chi^2 = \sum_{i=1}^{N} \left[\frac{y_i - y(x_i; a_1, \ldots, a_M)}{Var(y_i - y(x_i; a_1, \ldots, a_M))} \right]^2 \tag{C.69}$$

Even for simple models such as straight lines, this introduces nonlinearities which require sophisticated, iterative methods for minimising general functions.

General Linear Least Squares

A common specialisation of Equation C.66 is to assume that $y(x_i; a_1, \ldots, a_M)$ is a linear combination of M, potentially nonlinear, functions $X_k(x)$:

$$y(x) = \sum_{k=1}^{M} a_k X_k(x) \tag{C.70}$$

The $X_k(x)$ functions are called the basis functions and in this case the cost function is defined as:

$$\chi^2 = \sum_{i=1}^{N} \left[\frac{y_i - \sum_{k=1}^{M} a_k X_k(x_i)}{\sigma_i} \right]^2 \tag{C.71}$$

Rearranging the summations allows this expression to be written in matrix form:

$$\chi^2 = \mathbf{b} - \mathbf{A} \cdot \mathbf{a} \tag{C.72}$$

where:

$$A_{ij} = \frac{X_j(x_i)}{\sigma_i} \tag{C.73}$$

$$b_i = \frac{y_i}{\sigma_i} \tag{C.74}$$

The vector $\mathbf{a}$ is composed of the model parameters a_i. The matrix $\mathbf{A}$ is known as the design matrix of the problem and consists of the basis functions applied in columns to the data points in rows.

Differentiating Equation C.71 gives the following M equations for the minimum of χ^2:

$$\sum_{i=1}^{N} \frac{1}{\sigma_i^2} \left[y_i - \sum_{j=1}^{M} a_j X_j(x_i) \right] X_k(x_i) = 0 \tag{C.75}$$

where $k = 1, \ldots, M$. Interchanging the summation order allows the matrix form for this equation to be written:

$$\left(\mathbf{A}^T \cdot \mathbf{A} \right) = \mathbf{A}^T \cdot \mathbf{b} \tag{C.76}$$

This can be solved by numerical inversion of the $\mathbf{A}^T \cdot \mathbf{A}$ equations. A much better solution, however, involves the use of singular value decomposition.

C.1.4 Least Squares by Singular Value Decomposition

One of the advantages of the SVD is that singular matrices can be handled very effectively. If there is any degeneracy in the basis functions then the columns of the design matrix can be linearly dependent making the design matrix singular. Methods that solve Equation C.76 by inversion methods such as LU decomposition or Gauss-Jordan elimination cannot solve truly singular systems at all. For matrices that are almost singular, relative to the floating point representation of the machine, solutions are prone to high degrees of roundoff error due to unstably balanced fitting parameters a_k.

The singular value decomposition of a $M \times N$ matrix $\mathbf{A}$, where $M >= N$, can be expressed as the product of three matrices $\mathbf{U}$, $\mathbf{W}$ and $\mathbf{V}$ such that:

$$\mathbf{A} = \mathbf{U} \cdot \mathbf{W} \cdot \mathbf{V}^T \tag{C.77}$$

where $\mathbf{U}$ is a $M \times N$ column-orthogonal matrix, $\mathbf{V}$ is a $N \times N$ orthogonal matrix:

$$\mathbf{U}^T \cdot \mathbf{U} = \mathbf{V}^T \cdot \mathbf{V} = \mathbf{I} \tag{C.78}$$

and $\mathbf{W}$ is a diagonal $N \times N$ matrix with elements $w_1, \ldots, w_N$ known as the singular values of matrix $\mathbf{A}$. It turns out that all matrices have a singular value decomposition, whether N is less than, equal to or greater than M. For square matrices, the inversion is trivial:

$$\mathbf{V} \cdot [diag(1/w_j)] \cdot \mathbf{U}^T \tag{C.79}$$

This matrix is called the *pseudoinverse* of $\mathbf{A}$. Additionally, approximate inversions of singular matrices are possible. A singular matrix is indicated by a zero value for one or more of the

w_i values and inversion is performed by zeroing the $1/w_i$ values derived from zero w_i values. This corresponds to the addition of zero multiples of linear combinations of degenerate basis functions, rather than the large multiples introduced by inversion methods, thus alleviating instability and roundoff problems.

To understand the relevance to least squares methods, consider the simultaneous equations:

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \tag{C.80}$$

This set of equations can be solved by computing the SVD of matrix $\mathbf{A}$ and solving for $\mathbf{x}$ by:

$$\mathbf{x} = \mathbf{V} \cdot [diag(1/w_j)] \cdot \left(\mathbf{U}^T \cdot \mathbf{b}\right) \tag{C.81}$$

If the equations are underdetermined then it can be shown that the process of setting $1/w_j = 0$ is equivalent to minimising the length of the $\mathbf{x}$ vector and that for overdetermined systems the solution vector $\mathbf{x}$ is the best approximation in the least squares sense, i.e. it finds the vector $\mathbf{x}$ such that:

$$|\mathbf{A} \cdot \mathbf{x} - \mathbf{b}| \tag{C.82}$$

is a minimum. Compare this with the linear least squares problem where it is required to find the parameters $\mathbf{a}$ that minimise:

$$\chi^2 = |\mathbf{A} \cdot \mathbf{a} - \mathbf{b}|^2 \tag{C.83}$$

If $\mathbf{U}_{(i)}$ is the i-th column of $\mathbf{U}$ and $\mathbf{V}_{(i)}$ is the i-th column of $\mathbf{V}$ then the solution to the general linear least squares is:

$$\mathbf{a} = \sum_{i=1}^{M} \left(\frac{\mathbf{U}_{(i)} \cdot \mathbf{b}}{w_i}\right) \mathbf{V}_{(i)} \tag{C.84}$$

One incidental benefit of this is that the errors in the a_k parameters are also linear combinations of the columns of $\mathbf{V}$ so that the variances can be calculated by:

$$\sigma^2(a_j) = \sum_{i=1}^{M} \left(\frac{V_{ji}}{w_i}\right)^2 \tag{C.85}$$

and the covariances can be calculated:

$$Cov(a_j, a_k) = \sum_{i=1}^{M} \left(\frac{V_{ji} V_{ki}}{w_i^2}\right) \tag{C.86}$$

Another benefit is that singular values that are large enough not to cause problems with roundoff error can also be edited out preventing combinations of variables that do not contribute to reducing χ^2 from increasing the errors in the a_k values.

The main disadvantage of SVD methods over other methods is that it requires extra storage space and runs significantly slower. Considering the benefits obtained, the performance is more than compensated for.

C.1.5 SVD Fitting in Intensity and Range Data

The equations above relate directly to intensity data where straight lines, ellipses and other parametric shapes can be handled with simple basis functions.

The situation for range data is slightly complicated by having basis functions of two variables. The foregoing comments, however, can be directly extended to functions of any number of variables:

$$\chi^2 = \sum_{i=1}^{N} \left[\frac{z_i - \sum_{k=1}^{M} a_k X_k(x_i, y_i)}{\sigma_i} \right]^2 \tag{C.87}$$

There is also a potential problem with errors in the independent variables x_i and y_i. The merit function in this case is:

$$\chi^2 = \sum_{i=1}^{N} \left[\frac{z_i - \sum_{k=1}^{M} a_k X_k(x_i, y_i)}{Var(z_i - \sum_{k=1}^{M} a_k X_k(x_i, y_i))} \right]^2 \tag{C.88}$$

The problem with this is that the differentials $\frac{\partial \chi^2}{\partial a_k}$ are nonlinear in the a_k terms. To solve nonlinear systems such as this requires advanced methods for minimising nonlinear functions such as the Levenberg-Marquardt method [308], which are iterative in nature. It also means giving up SVD as a means of solving the equations as they are no longer in matrix form and the Levenberg-Marquardt method is itself an extension of the Normal Equations method ([76], chapter 15) with all the attendant problems of singular matrices.

A simple but partial solution might be to assume initial values for the a_k parameters and calculate:

$$\sigma_i^2 = Var(z_i - \sum_{k=1}^{M} a_k X_k(x_i, y_i)) \tag{C.89}$$

The SVD fitting procedure could be run to give more accurate values of a_k and the process repeated until there is no further reduction in χ^2. The stability and convergence of this method are not guaranteed but the merit function ensures that divergence will not occur and the SVD method ensures at least partial stability of the intermediate solutions.

For image processing using the SVD approach the ability to spot degenerate basis functions is potentially very useful so it would be desirable to stick with the SVD method if at all possible. One disadvantage, however, is that there are very few implementations of SVD calculations in parallel reported in the literature [302, 309]. Most of these are based on the methods of Golub and Reinsch [310] or Chan [311] and around Householder reduction to bidiagonal form. How these could be implemented on MIMD parallel machines is not obvious.

C.2 Canny Edge Detection Scheme

C.2.1 Overview

The Canny edge detector [248] is a well-studied method which combines several image processing functions into a single operator. The method as a whole was chosen to satisfy criteria for an ideal edge detector proposed by Canny. As implemented in this thesis, the Canny edge detection algorithm comprises four phases:

1. *Image smoothing.* This operator is designed to reduce the effect of noise on the edge detection phase. The image is convolved with a Gaussian filter, the degree of smoothing being controlled by the σ in the Gaussian distribution.

2. *Edge enhancement.* This is accomplished by convolving the image with the derivative of the Gaussian distribution. In practice, since this method forms a 'zero-crossing' at edge positions, orientation information can be extracted. The image is convolved separately with two directional filters to give edge strength components in the x and y direction. The angular information is extracted by resolving these two directional components.

3. *Edge thinning.* This algorithm is based on non-maxima suppression. The gradients of the edge strength are computed in both directions normal to the edge and if the edge strength is not a local maximum in this direction then the edge is suppressed. The gradients are calculated by a simple linear interpolation within a nine pixel neighbourhood.

4. *Hysteresis thresholding.* All points in the image are retained that are above an upper threshold. In addition, all neighbouring points in a nine pixel neighbourhood are tested. If they are above a lower threshold and their orientation is within an angular threshold of the central point's orientation then they are also retained. This process is then repeated in a recursive manner.

C.2.2 Canny Phases

The individual phases of the Canny operator are encapsulated in the following functions:

Image Smoothing

The raw image is supplied with a σ value for the Gaussians. Two smoothed images are generated, one smoothed using a 1D filter in the x direction and the other smoothed similarly in the y direction.

Firstly a 1D filter is generated from the following function:

$$G(n_i) = \frac{1}{2\pi\sigma^2} \int_{n_i-0.5}^{n_i+0.5} e^{-\frac{x^2}{2\sigma^2}} dx$$

where:

$$n_i = \{-n_{max}, -n_{max}+1, ..., n_{max}\}$$

and:

$$G(n_i) = G(n_{-i})$$

Note that Simpson's approximation is used to evaluate the integral:

$$\int_a^b f(x)dx \approx \frac{b-a}{6}\left[f(a) + 4f(\frac{a+b}{2}) + f(b)\right]$$

The image is then convolved separately in x and y directions using the 1D Gaussian filter:

$$S_x(x,y) = \sum_{k=-n_{max}}^{n_{max}} G(k)\ I(x+k,y)$$

and:

$$S_y(x,y) = \sum_{k=-n_{max}}^{n_{max}} G(k)\ I(x,y+k)$$

where $I(x,y)$ is the intensity value of the image at (x,y).

Edge detection

The images smoothed in x and y directions are enhanced in the y and x directions, respectively. Firstly, the derivative of the Gaussian distribution is formed approximately:

$$G'(n_i) \approx \frac{1}{2\pi\sigma^2}\left[e^{-\frac{(n_i+0.5)^2}{2\sigma^2}} e^{-\frac{(n_i-0.5)^2}{2\sigma^2}}\right]$$

This can be achieved simultaneously with the calculation of the 1D Gaussian filter since the exponentials are simply the $f(a)$ and $f(b)$ terms in the Simpson's approximation equation. Note that in this case, however, $G(n_i) = -G(n_{-i})$.

The smoothed images are then convolved with $G'(n_i)$ to produce edge strength components in the x and y directions, note that the x edges are calculated from the image smoothed in the y direction so the net effect is smoothing using a 2D Gaussian filter:

$$E_x(x,y) = \sum_{k=-n_{max}}^{n_{max}} G'(k)\ S_y(x+k,y)$$

and:

$$E_y(x,y) = \sum_{k=-n_{max}}^{n_{max}} G'(k)\ S_x(x,y+k)$$

Non-Maxima Suppression

The edge components in the x and y directions are resolved into edge strength and orientation components. Non-maxima points are set to zero and a scaling factor is provided for low intensity images.

At each point in the image, the gradient of the edge strength is calculated in both directions normal to the line. If the gradients are such that if either of the neighbouring points in the directions normal to the line would have a greater strength than the central point then the point is rejected and set to zero.

The edge strength of neighbouring points is determined by linear interpolation. For example, if the line orientation at the centre of a nine pixel neighbourhood is such that the line is at approximately 150° to the horizontal (where the magnitude of the x component is greater than the magnitude of the y component) then the edge strength of the conceptual neighbouring pixel in the NNE (north-north-east) direction can be estimated as:

$$\frac{E_x(x,y)}{E_y(x,y)}H(x+1,y+1) + \left(1 - \frac{E_x(x,y)}{E_y(x,y)}\right)H(x,y+1)$$

and in the SSW (south-south-west) direction by:

$$\frac{E_x(x,y)}{E_y(x,y)}H(x-1,y-1) + \left(1 - \frac{E_x(x,y)}{E_y(x,y)}\right)H(x,y-1)$$

where:

$$H(x,y) \;=\; \sqrt{(E_x(x,y)^2 \;+\; E_y(x,y)^2)}$$

Similar equations exist in each of the eight possible directions. There are degenerate cases where:

$$E_x(x,y) \;=\; 0 \quad \textit{and} \quad E_y(x,y) \;=\; 0$$

For all edges that are retained, the edge strength is generated from the hypotenuse and the orientation in the range $-\pi$ to $+\pi$ from the inverse tangent:

$$E_s(x,y) \;=\; H(x,y)$$

$$E_o(x,y) \;=\; \tan^{-1}\frac{E_y(x,y)}{E_x(x,y)}$$

where $E_s(x,y)$ and $E_o(x,y)$ are the edge strength and orientation components, respectively.

Hysteresis Thresholding

Hysteresis thresholding can be implemented in a simple recursive routine, summarised in Figure C.118. The image is scanned and all points above the upper threshold are marked as having been examined and are retained. Then, for each other point, all eight neighbouring points are examined and any that are above the low threshold are retained if their orientation lies within the angular threshold of the central point's orientation. Note that parallel lines should already have been rejected by the non-maxima suppression phase.

```
for each point strength(x,y) in the input image
  if strength(x,y) >= high
    then track_line(x,y)
    else output_strength(x,y) = 0

track_line(x,y)
  if (x,y) has not been analysed
    then
      mark (x,y) as having been analysed
      output_strength(x,y) = strength(x,y)
      for each neighbour at (x + j,y + k), -1 <= j,k <= 1
        if strength(x,y) >= low and difference between orientation(x,y)
        and orientation(x + j,y + k) < angular threshold
          then
            track_line(x + j,y + k)
```

Figure C.118: Recursive Hysteresis Thresholding Algorithm

Bibliography

[1] A. M. Wallace. Industrial applications of computer vision since 1982 . *IEE Proceedings*, 135(E)(3):17–136, 1988.

[2] M. J. Quinn. *Parallel Computing: Theory and Practice*. McGraw-Hill, 1994.

[3] S. Gorn. Letter to the Editor. *CACM*, 1(1):2–4, 1958.

[4] P. Pepper. *Deductive Derivation of Parallel Programs,* In [312], pages 1–53. Kluwer Academic, 1993.

[5] D. B. Skillicorn. The Bird-Meertens Formalism as a Parallel Model. In J. S. Kowalik and L. Grandinetti, editors, *NATO ASI Workshop on Software for Parallel Computation*, volume 106 of *F*, Cetrato, Italy, Jun 1993. Springer-Verlag.

[6] J. P. Wray, S. Fitzpatrick, M. Clint, and P. L. Kilpatrick. Deriving Distributed MIMD Implementations from Functional Specifications. *Advances in Parallel Computing*, 9:353–358, 1993.

[7] 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.

[8] D. B. Skillicorn and W. Cai. A Cost Calculus for Parallel Functional Programming. *Journal of Parallel and Distributed Programming*, 28(1):65–84, 1995.

[9] M. Kesseler. Constructing Skeletons in Clean The Bare Bones. In A. P. Wim Bom and J. T. Feo, editors, *High Performance Functional Computing*, pages 182–192. , Apr 1995.

[10] H. Deldari, J. R. Davy, and P. M. Dew. *The performance of parallel algorithmic skeletons,* In [313], pages 65–74. IOS Press, 1995.

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

[12] A. J. Field and P. G. Harrison. *Functional Programming*. Addison-Wesley, 1988.

[13] S. Atkins and A. S. Wagner. *Transputer Research and Applications*, volume 6. IOS Press, 1993.

[14] Inmos Ltd. *Occam2 Reference Manual*, 1988.

[15] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[16] K. Waugh, P. A. McAndrew, and G.J. Michaelson. Parallel Implementations from Functional Prototypes: A Case Study. Technical Report TR90/4, Heriot-Watt University, Riccarton, Edinburgh, 1990.

[17] J. R. Parker. *Algorithms for Image Processing and Computer Vision*. Wiley, Oct 1996.

[18] M. Maresca, M. A. Lavin, and H. Li. Parallel Architectures for Vision. *Proceedings of the IEEE*, 76(8):970–981, Aug 1988.

[19] M. Yasrebi. A Review of Various Philosophies on the Design and Analysis of Parallel Architectures for Vision. *The Australian Computer Journal*, 24(3):139–151, Nov 1992.

[20] R. Milner and M. Tofte. *Commentary on Standard ML*. MIT Press, 1991.

[21] A. Church. *The Calculi of Lambda-Conversion*. Princeton Univ. Press, NJ, 1941.

[22] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, Arrhus University, Denmark, Sep 1981.

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

[24] C. A. R. Hoare. Communicating Sequential Processes. *CACM*, 21(8):666–677, 1978.

[25] C. Hoare. *Communicating Sequential Processes.* Prentice-Hall, Englewood Cliffs, NJ, 1985.

[26] Boleslaw K Szymanski, editor. *Parallel Functional Languages and Compilers.* Frontier Series. ACM Press, New York, 1991.

[27] Kevin Hammond. Parallel Functional Programming: An Introduction (invited paper). In *Proceedings of PaSCo94, Linz, Austria.* World Scientific Publishing Company, September 1994.

[28] A. N. Choudhary and R. Ponnusamy. Run-time Data Decomposition for Parallel Implementation of Image Processing and Computer Vision Tasks. *Concurrency: Practice and Experience*, 4(4):313–334, Jun 1992.

[29] S. Yalamanchili and J. K. Aggarwal. Parallel-processing Methodologies for Image Processing and Computer Vision. *Advances in Electronics and Electron Physics*, 87:259–300, 1994.

[30] Y. Kozato. *Lazy Image Processing: An Investigation into Applications of Lazy Functional Languages in Image Processing.* PhD thesis, University of London, 1994.

[31] E. Luque, R. Suppi, and J. Sorribes. Designing Parallel Systems - A Performance Prediction Problem. *Microprocessors and Microsystems*, 16(1):25–35, 1992.

[32] F. Sotz. *A Method for Performance Prediction of Parallel Programs*, volume 457 of *LNCS*, pages 98–107. Springer-Verlag, 1990.

[33] R. M. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. *JACM*, 24(1):44–67, Jan 1977.

[34] H. Partsch and R. Steinbruggen. Program Transformation Systems. *ACM Computing Surveys*, 15(3):199–236, Sep 1983.

[35] J. M. Boyle and M. N. Muralidharan. Program Reuse Through Program Transformation. *IEEE Transactions on Software Engineering*, SE-10(5):574–588, 1984.

[36] P. G. Harrison and H. Khoshnevisan. Algebraic Transformation Techniques for Functional Languages. *Computer J.*, 31(3):229–242, 1988.

[37] M. Tanik and R. T. Yeh. Guest Editor's Introduction - Rapid Prototyping in Software Development. *IEEE Computer*, 22(5):9–10, 1989.

[38] H. Lichter, M. Schneider-Hufschmidt, and H. Zullighoven. Prototyping in Industrial Software Projects - Bridging the Gap Between Theory and Practice. *IEEE Transactions on Software Engineering*, 20(11):825–832, Nov 1994.

[39] J. Schwarz and F. A. Rabhi. *A Case Study in the Parallel Implementation of Functional Languages Using Skeletons,* In [314], volume 797 of *LNCS*, pages 357–362. Springer-Verlag, Munich, Apr 1994.

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

[41] R. Rangaswami. HOPP - A Higher-Order Parallel Programming Model. Internal report, Department of Computer Science, University of Edinburgh, 1995.

[42] H. W. To. *Optimising the Parallel Behaviour of Combinations of Program Components.* PhD thesis, Department of Computing, Imperial College of Science, Technology and Medicine, London, 1995.

[43] P.J. Besl and R.C. Jain. Three-dimensional object recognition. *ACM Computing Surveys*, 17(1):75–145, 1985.

[44] F. Arman and J.K. Aggarwal. Model-Based Object Recognition in Dense Range Images: A Review. *ACM Computing Surveys*, 25(1):5–43, Mar 1993.

[45] G. Zhang, J. Clark, and A. M. Wallace. Registered Depth and Intensity Data from an Integrated Vision Sensor. In *Imaging and Illumination for Metrology and Inspection*, volume 2348, pages 239–250, 1995.

[46] J. Clark and A. M. Wallace. Depth Sensing by Variable Baseline Triangulation. In *Proceedings of the 6th British Machine Vision Conference*, volume 1 and 2, pages 227–236, 1995.

[47] G. Zhang and A. M. Wallace. Edge classification and depth reconstruction by fusion of range and intensity edge data. In G. Sandini, editor, *Proc. Second European Conf. on Comput. Vision*, volume 588 of *LNCS*, pages 744–748. Springer-Verlag, 1992.

[48] G. Zhang, E. Thirion, and A.M. Wallace. Model matching using multiple data sources. In *Proc. of SPIE Conference on Machine Vision Applications in Industrial Inspection*, pages 2–12, San Jose, February, 1993.

[49] G. Zhang, M. Umasuthan, and A. M. Wallace. An Efficient Outlier Removal Algorithm. In *Proceedings SPIE 92, Non-linear Image Processing*, 1992.

[50] W. J. Austin, A. M. Wallace, and V. Fraitot. Parallel Algorithms for Plane Detection using an Adaptive Hough Transform. *Image and Vision Computing*, 9(6):372–384, 1991.

[51] M. Umasuthan. *Recognition and Position Estimation of 3D Objects from Range Images using Algebraic and Moment Invariants.* PhD thesis, Heriot-Watt University, Riccarton, Edinburgh, EH14 4AS, 1995.

[52] S. Hopkins, G. J. Michaelson, and A. M. Wallace. Parallel Imperative and Functional Approaches to Visual Scene Labelling. *Image and Vision Computing*, 7(3):178–193, 1989.

[53] K. G. Waugh and G. J. Michaelson. Parallel Process Placement from Functional Prototype Instrumentation. In *Proceedings TENCON 91, Energy, Computer, Communication and Control Systems*, volume 2, pages 275–282, New Delhi, Aug 1991. IEEE.

[54] K. G. Waugh. Parallel Imperative Programs from Functional Prototypes. In *Proceedings of the Workshop on the Parallel Implementation of Functional Languages*, pages 75–88. Dept. Electronics and Computer Science, University of Southampton, Jun 1991.

[55] A. M. Wallace, G. J. Michaelson, P. McAndrew, K. Waugh, and W. Austin. Dynamic Control and Prototyping of Parallel Algorithms for Intermediate and High-level Vision. *IEEE Computer*, 25(2):43–53, Feb 1992.

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

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

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

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

[60] T. Bratvold. A Skeleton-Based Parallelising Compiler for ML. In R. Plasmeijer and M. van Eekelen, editors, *Proceedings of the Fifth International Workshop on Implementation of Functional Languages, Nijmegen, the Netherlands*, pages 23–33, Sep 1993.

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

[62] N. R. Scaife, G. J. Michaelson, and A. M. Wallace. Prototyping Parallel Algorithms using Standard ML. In D. Pycock, editor, *Proceedings of the 6th British Machine Vision Conference*, volume 2, pages 671–680, University of Birmingham, Birmingham, Sep 1995.

[63] N. R. Scaife, G. J. Michaelson, and A. M. Wallace. Four Skeletons for Computer Vision. In *Implementation of Functional Languages '97*, Sep 1997.

[64] A. M. Wallace, G. J. Michaelson, N. Scaife, and W. J. Austin. A Dual Source, Parallel Architecture for Computer Vision. *The Journal of Supercomputing*, 12(1/2):37–56, Jan/Feb 1998.

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

[66] N. R. Scaife. Design and Implementation of a Parallel Depth and Intensity Segmentation System. Technical Report RM/96/3, Department of Computing and Electrical Engineering, Heriot-Watt University, Riccarton, Edinburgh, Dec 1995.

[67] N. R. Scaife, G. J. Michaelson, and A. M. Wallace. A Method for Developing Parallel Vision Algorithms with an Example of Edge Tracking. Technical Report RM/96/1, Department of Computing and Electrical Engineering, Heriot-Watt University, Riccarton, Edinburgh, Jan 1996.

[68] C. H. Chen. *Handbook of pattern recognition and computer vision*. World Scientific, 1993.

[69] R. C. Jain, R. Kasturi, and B. G. Schunck. *Introduction to Machine Vision*. McGraw-Hill, 1995.

[70] C.-L. Wang, P. B. Bhat, and V. K. Prasanna. High-Performance Computing for Vision. *Proceedings of the IEEE*, 84(7):931–945, Jul 1996.

[71] R. Cypher and J. L. C. Sanz. SIMD Architectures and Algorithms for Image Processing and Computer Vision. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 37(12):2158–2174, Dec 1989.

[72] V. Chaudhary and J. K. Aggarwal. *Parallelism in Computer Vision: A Review,* In [315], pages 271–309. Springer-Verlag, 1990.

[73] P. Baglietto, M. Maresca, M. Migliardi, and N. Zingirian. Image Processing on High-Performance RISC Systems. *Proceedings of the IEEE*, 84(7):917–929, Jul 1996.

[74] W. E. Alexander, D. S. Reeves, and C. S. Gloster. Parallel Image Procesing with the Block Data Parallel Architecture. *Proceedings of the IEEE*, 84(7):947–967, Jul 1996.

[75] B. G. Sherlock and D. M. Monro. Algorithm 749: Fast Discrete Cosine Transform. *ACM Transactions on Mathematical Software*, 21(4):372–378, Dec 1995.

[76] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C.* CUP, 2nd edition, 1992.

[77] M. I. Gillespie and D. D. Olesky. Ordering Givens Rotations for Sparse QR Factorisation. *SIAM Journal on Matrix Analysis and Applications*, 16(3):1024–1041, 1995.

[78] A. Krikelis and R. M. Lea. A Modular Massively Parallel Computing Approach to Image-Related Processing. *Proceedings of the IEEE*, 84(7):988–1004, Jul 1996.

[79] C. S. Burrus, H. Guo, and R. A. Gopinath. *Introduction to Wavelets and Wavelet Transforms: A Primer*. Prentice Hall, Aug 1997.

[80] J. J. Little, G. E. Blelloch, and T. A. Cass. Algorithmic Techniques for Computer Vision on a Fine-Grained Parallel Machine. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(3):244–257, Mar 1989.

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

[82] R. A. Jarvis. On the identification of the convex hull of a finite set of points in the plane. *Inform. Process. Lett.*, 2:18–21, 1973.

[83] Pinaki Mitra and Subhas C. Nandy. Efficient Computation of Rectilinear Geodesic Voronoi Neighbor in the Presence of Obstacles. *Journal of Algorithms*, 28(2):315–338, August 1998.

[84] H. Wang and M. Brady. PARADOX - A Heterogeneous Machine for the 3D Vision Algorithm. *Concurrency: Practice and Experience*, 7(4):257–272, Jun 1995.

[85] M. Rygol, S. Pollard, and C. Brown. MARVIN and TINA: A Multiprocessor 3-D Vision System. *Concurrency: Practice and Experience*, 3(4):333–356, Aug 1991.

[86] M. Rygol, S. Pollard, and C. Brown. Multiprocessor 3D Vision System for Pick and Place. *Image and Vision Computing*, 9(1):33–38, Feb 1991.

[87] S.B. Pollard, J.E.W. Mayhew, and J.P. Frisby. Pmf: A stereo correspondence algorithm using a disparity gradient limit. *Perception*, 14:449–470, 1985.

[88] J. A. Webb. Steps Toward Architecture Independent Image Processing. *IEEE Computer*, 25(2):21–31, Feb 1992.

[89] H. Wang, P. M. Dew, and J. Webb. Implementation of Apply on a transputer array. *Concurrency: Practice and Experience*, 3(1):43–54, Feb 1991.

[90] D. Gelernter. Generative Communications in Linda. *ACM TOPLAS*, 7(1):80–112, Jan 1985.

[91] J. A. Webb. Unifying Concepts in Architecture-Independent Parallel Image Processing in Adapt. In *Image Processing and Interchange: Implementation and Systems*, volume 1659, pages 228–239. SPIE, 1992.

[92] D. Parnas. A Technique for software module specification. *CACM*, 15(5):330–336, May 1972.

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

[94] B. Liskov. *Modular Programme Construction Using Abstraction,* In [316], volume 86 of *LNCS*, pages 354–389. Springer-Verlag, 1980.

[95] M. Stefik and D. Bobrow. Object-oriented programming: Themes and Variations. *AI Magazine*, 6(4):40–62, 1986.

[96] B. Shriver and P. Wegner. *Research Directions in Object-Oriented Programming.* MIT Press, 1987.

[97] W. Kim and F. H. Lochovsky. *Object-Oriented Concepts, Databases and Applications.* Addison-Wesley, 1989.

[98] T. Korson and J. D. McGregor. Understanding Object-Oriented: A Unifying Paradigm. *CACM*, 33(9):40–60, Sep 1990.

[99] R. J. Wirfs-Brock and R. E. Johnson. Surveying Current Research in Object-Oriented Design. *CACM*, 33(9):104–124, Sep 1990.

[100] B. J. Cox and A. J. Novobilski. *Object-Oriented Programming: An Evolutionary Approach.* Addison-Wesley, 2nd edn. edition, 1991.

[101] J. Martin and J. J. Odell. *Object-Oriented Methods: A Foundation.* Prentice-Hall, 1995.

[102] D. B. MacQueen. Modules for Standard ML. In *Proc. 1984 ACM Symposium on Lisp and Functional Programming*, 1984.

[103] P. Hudak, S. P. Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, M. M. Guzman, K. Hammond, J. Hughes, T. Johnsson, D. Kieburtz, R. Nikhil, W. Partain, and J. Peterson. Report on the Programming Language Haskell - A Nonstrict, Purely Functional Language Version 1.2. *SIGPLAN Notices*, 27(5):R1–R163, 1992.

[104] D. B. Skillicorn. Bulk Datatypes and General Purpose Computation. In *Bulk Datatypes for Architecture Independence.* British Computer Society, Parallel Processing Specialist Group, May 1994.

[105] R. Rangaswami. *A Cost Analysis for a Higher-Order Parallel Programming Model.* PhD thesis, University of Edinburgh, 1995.

[106] S. Nishimura and A. Ohori. A Calculus for Exploiting Data Parallelism on Recursively Defined Data. In *Proc. Intl. Workshop on Theory and Practice on Parallel Programming*, volume 907 of *LNCS*, 1994.

[107] M. Sudholt. *Data Distribution Algebras - A Formal Basis for Programming Using Skeletons,* In [317], pages 19–38. North-Holland, Jun 1994.

[108] A. Maaßen. Parallel Programming with Data Structures and Higher Order Functions. *Science of Computer Programming*, 18:1–38, 1992.

[109] A. Shafarenko. Symmetries in Data Parallelism. *Computer Journal*, 38(5):365–380, 1995.

[110] C. B. Jay. A Semantics for Shape. *Science of Computer Programming*, to appear., 1995.

[111] C. B. Jay. Shape analysis for parallel computing. In J. Darlington, editor, *Proceedings of the fourth international parallel computing workshop: Imperial College London, 25–26 September, 1995*, pages 287–298. Imperial College/Fujitsu Parallel Computing Research Centre, 1995.

[112] C. B. Jay. Programming in FISh. *International Journal on Software Tools for Technology Transfer*, 2, 1999.

[113] C. B. Jay. Denotational Semantics of Shape: Past, Present and Future. In Scedrov, A. and Jung, A., editor, *Proceedings of the Fifteenth Conference on the Mathematical Foundations of Programming Semantics (MFPS XV)*, volume 20, pages 197–210, Tulane University, New Orleans, LA USA, April-May 1999. Elsevier.

[114] C.B. Jay, M.I. Cole, M. Sekanina, and P.A. Steckler. A monadic calculus for parallel costing of a functional language of arrays. In C. Lengauer, M. Griebl, and S. Gorlatch, editors, *Euro-Par'97 Parallel Processing*, volume 1300 of *Lecture Notes in Computer Science*, pages 650–661. Springer, August 1997.

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

[116] J. H. Williams. On the Development of the Algebra of Functional Programmes. *ACM Transactions on Programming Languages and Systems*, 4(4):733–757, Oct 1982.

[117] P. G. Harrison. A Higher-Order Approach to Parallel Algorithms. *Computer Journal*, 35(6):555–566, 1992.

[118] M. Danelutto and S. Pelagatti. Parallel Implementation of FP using a Template-based Approach. In R. Plasmeijer and M. van Eekelen, editors, *Proc. 5th International Workshop on Implementation of Functional Languages*, Tech. Rep. No. 93-21, pages 7–21, Nijmegen, The Netherlands, Sep 1993.

[119] M. Danelutto and S. Pelagatti. Functional Programming on a Bunch of Processors. Technical Report TR 17/93, Universitá Degli Studi Di Pisa, Dipartimento Di Informatica, 1993.

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

[121] R. S. Bird. Algebraic Identities for Program Calculation. *Computer Journal*, 32(2):122–126, 1989.

[122] K. E. Iverson. *A Programming Language*. Wiley, New York, 1962.

[123] M. I. Cole. Parallel Programming, List Homomorphisms and the Maximum Segment Sum Problem. Tech. Rep. CSR-25-93, University of Edinburgh, May 1993.

[124] D B Skillicorn. Questions and Answers about Categorical Data Types. In *Bulk Data Types for Architecture Independence (Seminar Program)*, London, May 1994. British Computer Society, Parallel Processing Specialist Group.

[125] Z. N. Grant-Duff and P. G. Harrison. Skeletons, List Homomorphisms and Parallel Program Transformation. Tech. Rep. 94/14, Imperial College, London, 1994.

[126] K. G. Kumar and D. B. Skillicorn. Data Parallel Geometric Operations on Lists. *Parallel Computing*, 21:447–459, 1995.

[127] W. Cai and Skillicorn D. B. Calculating Recurrences Using the Bird-Meertens Formalism. *Parallel Processing Letters*, 5(2):179–190, June 1995.

[128] D. B. Skillicorn. Parallel Implementation of Tree Skeletons. Tech. Rep. ISSN-0836-0227-95-380, Dept. Computing and Information Science, Queen's University, Kingston, Canada, Mar 1995.

[129] K. Abrahamson, N. Dadoun, D. G. Kirkpatrick, and I. Przytycka. A Simple Parallel Tree Contraction Algorithm. *Journal of Algorithms*, 10(2):287–302, 1989.

[130] G. L. Miller and J. H. Reif. Parallel Tree Contraction .2. Further Applications. *SIAM Journal on Computing*, 20(6):1128–1147, 1991.

[131] W. J. Hsu and C. V. Page. Parallel Tree Contraction and Prefix Computations on a Large Family of Interconnected Topologies. *Acta Informatica*, 32(2):145–153, 1995.

[132] S. Gorlatch and C. Lengauer. Parallelization of Divide-and-Conquer in the Bird-Meertens Formalism. *Formal Aspects of Computing*, 7(6):663–682, 1995.

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

[134] S. Gorlatch. From Transformations to Methodology in Parallel Program Development: A Case Study. *Microprocessing and Microprogramming*, 41:571–588, 1996.

[135] T. Axford and M. Joy. List Processing Primitives for Parallel Computation. *Comput. Lang.*, 19(1):1–17, 1993.

[136] E. A. Boiten, A. M. Geerling, and H. A. Partsch. Transformational Derivation of (parallel) Programs using Skeletons. In *Computing Science in the Netherlands*, pages 97–108. , 1993.

[137] W.-N. Chin. Synthesizing Efficient Parallel Programs from Sequential Specifications. Technical Report TRA9/95, Dept. Informationn Systems and Computer Science, National University of Singapore, Sep 1995.

[138] W.-N. Chin. Towards and Automated Tupling Strategy. In *Proceedings 3rd ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 119–132, Copenhagen, Jun 1993.

[139] M. Barr and C. Wells. *Category Theory for Computing Science*. Prentice-Hall, 1990.

[140] Benjamin C. Pierce. *Basic Category Theory for computer Scientists*. Foundations of Computing Series. MIT Press, 1991.

[141] D. B. Skillicorn. Categorical Data Types. In J. R. Davy and P. M. Dew, editors, *Workshop on Abstract Models for Highly Parallel Computation*, pages 155–168. Oxford University Press, 1995.

[142] C. R. Banger and D. B. Skillicorn. Flat Arrays as a Categorical Data Type. Tech. rep., Dept. of Computing and Information Science, Queen's University, Kingston, Canada, Nov 1992.

[143] C. R. Banger and D. B. Skillicorn. A Foundation for Theories of Arrays. Tech. rep., Dept. of Computing and Information Science, Queen's University, Kingston, Canada, Jan 1993.

[144] E. G. J. M. H. Nocker, J. E. W. Smetsers, M. C. J. D. van Eekelen, and M. J. Plasmeijer. Concurrent Clean. In *Proc. Parallel Architectures and Languages Europe, PARLE'91*, volume 505 (II) of *LNCS*, pages 202–219. Springer, 1991.

[145] W. M. Gentleman. Some Complexity Results for Matrix Computations on Parallel Processors. *JACM*, 25:112–115, 1978.

[146] H. Kuchen, R. Plasmeijer, and H. Stoltze. Distributed Implementation of a Data Parallel Functional Language. In *Parallel Architectures and Languages Europe*, volume 817 of *LNCS*, pages 464–477. Springer-Verlag, 1994.

[147] R. Harrison. Parallel Programming with Pure Functional Languages. Unpublished, Dept. Electronics and Computer Science, University of Southampton, 1995.

[148] J.-P. Banâtre and D. Le Métayer. A New Computational Model and its Discipline of Programming. Research Report 566, INRIA, 1986.

[149] P. H. J. Kelly. *Functional Programming for Loosely-coupled Multiprocessors*. Research Monographs in Parallel and Distributed Computing. Pitman, 1989.

[150] A. C. Sodan and H. Bock. *Extracting Characteristics from Functional Programs for Mapping to Massively Parallel Machines,* In [318], pages 134–148. Apr 1995.

[151] P. H. Hartel, R. F. H. Hofman, K. G. Langendoen, H. L. Muller, W. G. Vree, and L. O. Hertzberger. A Toolkit for Parallel Functional Programming. *Concurrency: Practice and Experience*, 7(8):765–793, Dec 1995.

[152] W. G. Vree and P. H. Hartel. Communication Lifting: Fixed Point Computation for Parallelism. *Journal of Functional Programming*, 5(4):549–581, Oct 1995.

[153] W. G. Vree. *Design Considerations for a Parallel Reduction Machine*. PhD thesis, Department of Computer Systems, University of Amsterdam, Dec 1989.

[154] C. A. Herrmann, C. Lengauer, R. Gunz, J. Laitenberger, and C. Schaller. A Compiler for HDC. Technical Report MIP-9907, Fakultat fur Mathematik und Informatik, Universitat Passau, May 1999.

[155] C. A. Herrmann and C. Lengauer. The hdc compiler project. In A. Darte and G.-A. Silber and Y. Robert, editor, *Proc. Eighth Int. Workshop on Compilers for Parallel Computers (CPC 2000)*, pages 239–254. LIP, ENS Lyon, 2000.

[156] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Series in Computer Science and Information Processing. Addison-Wesley, 1974.

[157] H. Toivonen. *Discovery of Frequent Patterns in Large Data Collections*. PhD thesis, Department of Computer Science, University of Helsinki, 1996.

[158] G. H. Botorog and H. Kuchen. Skil: An Imperative Language with Algorithmic Skeletons for Efficient Distributed Programming. In *Proceedings of the Fifth International Symposium on High Performance Distributed Computing (HPDC-5)*, pages 243–252. IEEE Computer Society Press, 1996.

[159] T. Richert. Management of Distributed Dynamic Data with Algorithmic Skeletons. In *Proceedings of the International Conference on Parallel Computing 99*, Delft, The Netherlands, 1999.

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

[161] J. Darlington, M. Ghanem, and H. W. To. Structured Parallel Programming. In *Proc. Massively Parallel Programming Models Conference*, Berlin, Sep 1993.

[162] J. Darlington, M. Ghanem, H. W. To] J. Darlington, Y. Guo, H. W. To, Q. Wu, J. Yang, and M. Kohler. Fortran-S: A Uniform Functional Interface to Parallel Imperative Languages. In *Proc. 3rd Parallel Computing Workshop*, Kawasaki, Japan, 1994. Fujitsu Laboratories Ltd.

[163] S. Lucco and O. Sharp. Parallel Programming with Coordination Stuctures. In *Proc. 18th ACM POPL*, 1991.

[164] D. Gelernter and N. Carriero. Coordination languages and their Significance. *CACM*, 35(2):96–107, Feb 1992.

[165] J. Darlington, Y. Guo, H. W. To, and J. Yang. Functional Skeletons for Parallel Coordination. In *Proceedings of Europar 95*, 1995.

[166] J. H. Hur and K. Chon. *Overview of a Parallel Object-Oriented Language CLIX,* In [319], volume 276 of *LNCS*, pages 265–273. Springer-Verlag, 1987.

[167] J. Darlington, H. W. To, and Y. Jing. Skeletons for Structured Parallel Composition. In *Proc. 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1995.

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

[169] R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[170] S. Bokhari. Partitioning Problems in Parallel, Pipelined and Distributed Computing. *IEEE Transactions on Computers*, 37(1):48–57, Jan 1988.

[171] F. A. Rabhi. Exploiting Parallelism in Functional Languages: A "Paradigm-Oriented Approach". In J. R. Davy and P. M. Dew, editors, *Workshop on Abstract Machine Models for Highly Parallel Computers*, pages 118–139. Oxford University Press, 1995.

[172] D. I. Bevan *et al.* Design Principles of a Distributed Memory Architecture for Parallel Graph Reduction. *Computer Journal*, 32(5):461–469, 1989.

[173] V. S. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, 2(4):315–339, 1990.

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

[175] F. A. Rabhi. *A Parallel Programming Methodology Based on Paradigms,* In [320], pages 239–252. IOS Press, 1995.

[176] J. Schwarz and F.A. Rabhi. *A Skeleton-Based Implementation of Iterative Transformation Algorithms using Functional Languages,* In [321], pages 119–133. IOS Press, Apr 1996.

[177] R. Miller. A Constructive Theory of Multidimensional Arrays. Master's thesis, Oxford University, Feb 1993.

[178] D. Feldcamp, H. V. Sreekantaswamy, A. Wagner, and S. Chanson. *Towards a Skeleton-based Parallel Programming Environment,* In [322], pages 104–115. IOS Press, 1992.

[179] D. Feldcamp and A. Wagner. *Parsec - A Software Development Environment for Performance Oriented Parallel Programming,* In [13], pages 247–262. IOS Press, 1993.

[180] J. M. Nash, P. M. Dew, M. E. Dyer, and J. R. Davy. *Parallel Algorithm Design on the WPRAM Model,* In [323]. Oxford University Press, 1995.

[181] R H Bisseling and W F McColl. Scientific computing on bulk synchronous parallel architectures. Technical Report 836, Department of Mathematics, University of Utrecht, dec 1993.

[182] A. Y. Grama, A. Gupta, and V. Kumar. Isoefficiency: Measuring the Scalability of Parallel Algorithms and Architectures. *IEEE Parallel and Distributed Technology*, -(-):12–21, Aug 1993.

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

[184] M. Danelutto, R. Di Meglio, S. Orlando, S. Pelagatti, and M. Vanneschi. The P^3L Language: An Introduction. Technical Report HPL-PSC-91-29, Hewlett-Packard Laboratories, Pisa Science Center, Pisa, Italy, Dec 1991.

[185] F. Baiardi, M. Danelutto, R. Di Meglio, M. Jazayeri, M. Mackey, S. Pelagatti, F. Petrini, T. Sullivan, and M. Vanneschi. Pisa Parallel Processing Project on General-purpose Highly-parallel Computers. Technical Report HPL-PSC-91-23, Hewlett-Packard Laboratories, Pisa Science Center, Pisa, Italy, May 1991.

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

[187] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P(3)L - A Structured High-level Parallel Language and its Structured Support. *Concurrency: Practice and Experience*, 7(3):225–255, 1995.

[188] F. Baiardi, M. Jazayeri, M. Mackey, F. Petrini, and M. Vaneschi. An Abstract Architecture for Massively Parallel Machines. In *Workshop on Abstract Machine Models for Highly Parallel Computers*, Leeds, Apr 1991.

[189] E. A. Brewer and C. N. Dellarocas. Proteus User Documentation Version 0.2. Technical report, MIT, Oct 1991.

[190] S. Ciarpaglini, M. Danelutto, L. Folchi, C. Manconi, and S. Pelagatti. ANACLETO: a template-based p3l compiler. In *Proceedings of the Seventh Parallel Computing Workshop (PCW '97)*, pages P2–F–1–7, Australian National University, Canberra, August 1997.

[191] M. Danelutto, F. Pasqualetti, and S. Pelagatti. Skeletons for data parallelism in p3l. In C. Lengauer, M. Griebl, and S. Gorlatch, editors, *Proc. of EURO-PAR '97, Passau, Germany*, volume 1300 of *LNCS*, pages 619–628. Springer, August 1997.

[192] B. Bacci, B. Cantalupo, M. Danelutto, S. Orlando, D. Pasetto, S. Pelagatti, and M. Vanneschi. An environment for structured parallel programming. In L. Grandinetti, M. Kowalick, and M. Vaitersic, editors, *Advances in High Performance Computing*, pages 219–234. Kluwier, Dordrecht, The Netherlands, 1997.

[193] T. Bratvold. Functional Components for Parallel Programming. Master's thesis, Department of Computer Science, Heriot-Watt University, Edinburgh, September 1991.

[194] A. L. Davis and R. M. Keller. Dataflow Program Graphs. *IEEE Computer*, pages 26–41, Feb 1982.

[195] D. Fernández-Baca. Allocating Modules to Processors in a Distributed System. *IEEE Transactions on Software Engineering*, 15(11):1427–1436, Nov 1989.

[196] N. R. Scaife, G. J. Michaelson, and A. M. Wallace. Using Functional Prototypes to Develop Parallel Vision Algorithms: a Case Study of Edge Tracking. *Submitted*, 1996.

[197] J. Serot, G. Quenot, and B. Zavidovique. Functional programming on a data flow architecture: Applications in real-time image processing. *Machine Vision and Applications*, 7:44–56, 1993.

[198] Dominique Ginhac. *Prototypage rapide d'applications parallèles de vision artificelle par sqelettes fonctionnels.* PhD thesis, Université Blaise Pascal - Clermont II, 1999.

[199] J. M. Anderson, S. P. Amarasinge, and M. S. Lam. Data and Computation Transformations for Multiprocessors. In *Proceedings of the 5th ACM Symposium on Principles and Practice of Parallel Programming*, Jul 1995.

[200] L. A. Crowl and T. J. Leblanc. Parallel Programming with Control Abstraction. *ACM TOPLAS*, 16(3):524–576, May 1994.

[201] I. Foster and S. Taylor. A Compiler Approach to Scalable Concurrent-Program Design. *ACM TOPLAS*, 16(3):577–604, May 1994.

[202] J. M. Anderson and M. S. Lam. Global Optimisations for Parallelism and Locality on Scalable Parallel Machines. *SIGPLAN Notices*, 28(6):112–125, Jun 1993.

[203] M. W. Hall, S. P. Amarasinge, B. R. Murphy, S.-W. Liao, and M. S. Lam. Detecting Coarse-Grain Parallelism Using an Interprocedural Parallelizing Compiler. In *Proceedings of Supercomputing '95*, Dec 1995.

[204] S. S. Pinter and R. Y. Pinter. Program Optimization and Parallelization Using Idioms. *ACM TOPLAS*, 16(3):305–327, May 1994.

[205] A. L. Fisher and A. M. Ghuloum. Parallelizing Complex Scans and Reductions. *SIGPLAN*, 29(6):135–146, Jun 1994.

[206] A.V. Aho and J.D. Ullman. *Principles of compiler design*. Addison-Wesley, 1977.

[207] G. J. Michaelson. *Interpreter Prototypes from Formal Language Definitions*. PhD thesis, Dept. Computing and Electrical Engineering, Heriot-Watt University, Edinburgh, Apr 1993.

[208] S. Peyton Jones. *The Implementation of Functional Languages*. Prentice-Hall, 1987.

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

[210] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, 1981.

[211] L. Cardelli. A Polymorphic λ-calculus with type: Type. Technical Report Rep. No. 10, DEC, Palo Alto, California, Jan 1986.

[212] R. Milner. The Polyadic π-Calculus: A Tutorial. LFCS Report Series ECS-LFCS-91-180, Dept. Computer Science, University of Edinburgh, 1991.

[213] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Semantics*. MIT Press, 1977.

[214] R. M. Burstall, D. B. MacQueen, and D. T. Sanella. HOPE: An Experimental Applicative Language. Technical Report CSR-62-80, Department of Computer Science, University of Edinburgh, 1980.

[215] D. A. Turner. *Miranda: A Non-strict Functional Language with Polymorphic Types*, volume 201 of *LNCS*, pages 1–16. Springer-Verlag, 1985.

[216] P. Hudak, S. Peyton Jones, and P. Wadler (eds.). Report on the Programming Language Haskell, A Non-strict, Purely Functional Language Version 1.2. Technical report, Yale University, University of Glasgow and others., Glasgow, 1992.

[217] L. Cardelli. Basic Polymorphic Type Checking. *Science of Computer Programming*, 8:147–172, 1987.

[218] J. Mitchell. Coercion and Type Inference. In *Proc. 11th Ann. ACP Symposium on Principles of Programming Languages*, pages 175–185, Salt Lake City, Ut., Jan 1984. ACM.

[219] J. Mitchell. Type Inference and Type Containment. *LNCS*, 173:51–67, 1984.

[220] D. Tarditi and A. Diwan. The Full Cost of a Generational Copying Garbage Collection Implementation. Technical report, School of Computer Science, Carnegie Mellon University, Aug 1993.

[221] J. Goguen and K. Parsaye-Ghomi. *Algebraic Denotational Semantics using Parameterised Abstract Modules*, volume 107 of *LNCS*, pages 292–309. Springer Verlag, NY, 1981.

[222] D. Sannella and A. Tarlecki. Algebraic Specification and Formal Methods for Program Development: What are the Real Problems ? *EATCS Bulletin*, 41:134–137, 1990.

[223] H. Søndergaard and P. Sestoft. Referential transparency, definiteness and unfoldability. *Acta Informatica*, 27:505–517, 1990.

[224] W. V. O. Quine. *Word and Object*. MIT Press, 1960.

[225] P. J. Landin. The Mechanical Evaluation of Expressions. *Computer J.*, 6(4):308–320, 1964.

[226] H. Curry and R. Feys. *Combinatory Logic*. North-Holland, Amsterdam, 1958.

[227] G. Boudol. *Computational Semantics of Term Rewriting Systems*, chapter 5, pages 169–236. CUP, 1985.

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

[229] J. R. McGraw. The VAL Language: Description and Analysis. *ACM TOPLAS*, 4(1):44–82, Sep 1982.

[230] K. Ekanadham. *A Perspective on Id* In [26]. ACM Press, 1991.

[231] S. K. Skedzielewski. *Sisal,* In [26]. ACM Press, 1991.

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

[233] M. D. Cripps, J. Darlington, A. J. Field, P. G. Harrison, and M. J. Reeve. *The Design and Implementation of ALICE: a Parallel Graph Reduction Machine*, pages 300–326. 1987.

[234] J. A. Keane. An Overview of the Flagship System. *J. Functional Programming*, 4(1):19–45, Jan 1994.

[235] T. Johnsson. *Parallel Evaluation of Functional Programmes: The <ν-G> Machine Approach*, pages 1–5. Springer-Verlag, 1991.

[236] Simon L Peyton Jones, Chris Clack, Jon Salkild, and Mark Hardie. GRIP - A High-Performance Architecture for Parallel Graph Reduction. In G Kahn, editor, *Functional Programming Languages and Computer Architecture*, pages 98–112. Springer-Verlag, Berlin, September 1987.

[237] S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, Florida, Jan 1996.

[238] P. Bailey and M. Newy. Implementing ML on Distributed Memory Multiprocessors. *SIGPLAN*, 28(1):56–59, Jan 1993.

[239] P. R. Bailey and M. C. Newey. An Extension of ML for Distributed Memory Multicomputers. Technical report, Department of Compuuter Science, Australian National University, 1994.

[240] G. Koutsakis. Parallel Low Level Vision from Functional Prototypes. Master's thesis, Dept. of Computing and Electrical Engineering, Heriot-Watt University, Riccarton, Edinburgh, Oct 1993.

[241] M. I. Clarkson. Parallel Implementation of a Range Image Segmentation Algorithm. Master's thesis, Department of Computing and Electrical Engineering, Heriot-Watt University, Riccarton, Edinburgh, EH14 4AS, 1994.

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

[243] F.A. Rabhi. Experimenting divide-and-conquer algorithms with a parallel graph reduction machine. In *Technical Report*, University of Sheffield. Dept. Computer Science.

[244] E. Horowitz and A. Zorat. Divide and Conquer for Parallel Processing. *IEEE Transactions on Computers*, C-32(6):582–585, 1983.

[245] R. J. Schalkoff. *Digital Image Processing and Computer Vision*. Wiley, 1989.

[246] J. Jeuring. The Derivation of a Hierarchy of Algorithms for Pattern Matching on Arrays. In *Proceedings of the ATABLE-92 Workshop*, 1992.

[247] P.J. Besl and R.C. Jain. Segmentation through variable-order surface fitting. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 10:167–192, Mar 1988.

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

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

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

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

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

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

[254] H. K. Yuen, J. Illingworth, and J. Kittler. Ellipse detection using the Hough transform. *Proc. 4th Alvey Vision Conf.*, pages 265–271, 1988.

[255] H. K. Muammar and M. Nixon. Tristage Hough transform for multiple ellipse extraction. *IEE Proc. Comput. Digital Tech.*, 138(1):27–35, 1991.

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

[257] E. R. Davies. Application of the generalised Hough transform to corner detection. *IEE Proc. E*, 135:49–54, 1988.

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

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

[260] M. G. Albanesi and M. Ferretti. A Space Saving Approach to the Hough Transform. *Proceedings 10th Int. Conf. on Pattern Recognition*, 1990.

[261] A. W. Appel and D. B. MacQueen. *Standard ML of New-Jersey*, volume 528 of *LNCS*, pages 1–13. Springer-Verlag, 1991.

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

[263] S. F. Hummel and R. Kelly. A Rationale for Massively Parallel Programming with Sets. *Journal of Programming Languages*, 1(3):187–207, 1993.

[264] D. Le Métayer. ACE: An Automatic Complexity Evaluator. *ACM TOPLAS*, 10(2):248–266, Apr 1988.

[265] Y. Cohen, S. Landy, M. Pavel, and G. Sperling. *HIPS: Image Processing Under Unix Software and Applications.* Human Information Processing Laboratory, Department of Psychology, New York University, 1982.

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

[267] G. Stockman. Object Recognition and Localisation via Pose Clustering. *Computer Vision, Graphics and Image Processing*, 40:361–387, 1987.

[268] W. J. Austin and A. M. Wallace. Object Location by Parallel Pose Clustering. *Computer Vision and Image Understanding*, 72(3):304–327, 1998.

[269] P.J. Besl and R.C. Jain. Invariant surface characteristics for 3D object recognition. *Computer Vision, Graphics and Image Processing*, 33:33–80, 1986.

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

[271] D. S. Hirschberg, A. K. Chandra, and D. V. Sarwate. Computing Connected Components on Parallel Computers. *CACM*, 22(8):461–464, Aug 1979.

[272] G. H. Golub and C. F. van Loan. *Matrix Computations.* Johns Hopkins University Press, 1989.

[273] L. Clarke and G. Wilson. Tiny - An Efficient Routing Harness for the Inmos Transputer. *Concurrency - Practice and Experience*, 3(3):221–245, 1991.

[274] A. Ferreira and S. Ubeda. Ultra-fast Parallel Contour Tracking, with Applications to Thinning. *Pattern Recognition*, 27(7):867–878, 1994.

[275] M. Mirmehdi and T. J. Ellis. Parallel Approach to Tracking Edge Segments in Dynamic Scenes. *Image and Vision Computing*, 11(1):35–48, 1993.

[276] J. P. Gambotto and O. Monga. A Parallel Hierarchical Algorithm for Region Growing. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 1985.

[277] G. P. Otto and T. K. W. Chau. A "Region-Growing" Algorithm for Matching of Terrain Images. In *Proc. Alvey Vision Conf.*, pages 153–158, 1988.

[278] T.-J. Fan, G. Medioni, and R. Nevatia. Segmented Descriptions of 3-D Surfaces. *IEEE Journal of Robotics and Automation*, 3(6):527–538, 1989.

[279] E. El-Hujazi and A. Sood. Range Image Segmentation with Applications to Robust Bin-Picking using Vacuum Gripper. *IEEE Transactions on Systems, Man and Cybernetics*, 20(6):1313–1324, 1990.

[280] R. Hoffman and A.K. Jain. Segmentation and Classification of Range Images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 9(5):608–620, Sep 1987.

[281] S. M. Bhandarkar and A. Siebert. Integrating Edge and Surface Information for Range Image Segmentation. *Pattern Recognition*, 25(9):947–962, 1992.

[282] B. Sabata, F. Arman, and J. K. Aggarwal. Segmentation of 3D Range Images Using Pyramidal Data Structures. *CVGIP: Image Understanding*, 57(3):373–387, May 1993.

[283] T. S. Newman, P. J. Flynn, and A. K. Jain. Model-Based Classification of Quadric Surfaces. *CVGIP: Image Understanding*, 58(2):235–249, Sep 1993.

[284] F. Quek, R. Jain, and T. E. Weymouth. An Abstraction-Based Approach to 3-D Pose Determination from Range Images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(7):722–736, 1993.

[285] G. Roth and M. D. Levine. Extracting Geometric Primitives. *CVGIP: Image Understanding*, 58(1):1–22, 1993.

[286] A. Gupta and R. Bajcsy. Volumetric Segmentation of Range Images Using Superquadric Models. *CVGIP: Image Understanding*, 58(3):302–326, 1993.

[287] Y. K. Ham and R. H. Park. 3D object recognition in range images using hidden Markov models and neural networks. *Pattern Recognition*, 32(5):729–742, 1999.

[288] Z. Shao and J. Kittler. Shape representation and recognition based on invariant unary and binary relations. *Image and Vision Computing*, 17(5-6):429–444, 1999.

[289] I. Shimshoni and J. Ponce. Probabilistic 3D object recognition. *International Journal of Computer Vision*, 36(1):51–70, 2000.

[290] N. Yokoya and M. D. Levine. Range Image Segmentation Based on Differential Geometry: A Hybrid Approach. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(6):643–649, Jun 1989.

[291] M. H. Sunwoo, A. S. Baroody, and J. K. Aggarwal. A Parallel Algorithm for Region Labelling. In *Proc. of Workshop on Comput. Arch. for Pattern Anal. Mach. Intell.*, 1987.

[292] R. Cypher, J. L. Sanz, and L. Snyder. Hypercube and Shuffle-Exchange Algorithms for Image Component Labelling. In *Proc. of Workshop on Comput. Arch. for Pattern Anal. Mach. Intell.*, pages 5–10, 1987.

[293] M. Manohar and H. K. Rampriyan. Connected Component Labeling of Binary Images on a Mesh Connected Massively Parallel Processor. *CVGIP*, 45:133–149, Feb 1989.

[294] Y. Shiloach and U. Vishkin. An O(log n) Parallel Connectivity Algorithm. *J. Algorithms*, 3:57–67, 1982.

[295] W. E. Snyder and A. Cowart. An Iterative Approach to Region Growing Using Associative Memories. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 5(3), May 1983.

[296] N. Scaife, P. Bristow, G. Michaelson, and P. King. Engineering a parallel compiler for SML. In C. Clack, T. Davie, and K. Hammond, editors, *Proceedings of 10th International Workshop on Implementation of Functional Languages*, pages 213–226, University College London, Sep 1998.

[297] E. Trucco and R.B. Fisher. Experiments in curvature-based segmentation of range data. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 17(2):177–182, Feb 1995.

[298] A. Scher, M. Shneier, and A. Rosenfeld. Clustering of Collinear Line Segments. Technical Report TR-888, Computer Science Center, University of Maryland, 1980.

[299] R. Y. Tsai. An Efficient and Accurate Camera Calibration Technique for 3-D Machine Vision. In *IEEE Computer Vision and Pattern Recognition*, pages 364–374, 1986.

[300] L.-D. Cai. An Estimate of the Relationship between Zero Thresholds of Gaussian and Mean Curvature. Research Paper RP450, Division of Informatics, University of Edinburgh, 1989.

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

[302] K. A. Gallivan. *Parallel algorithms for matrix computations.* SIAM, 1990.

[303] M. Brady, J. Ponce, A. Yullie, and H. Asada. Describing Surfaces. *CVGIP*, 32:1–28, 1985.

[304] A. Hoover, G. Jean-Baptiste, X.Y. Jiang, P.J. Flynn, H. Bunke, D.B. Goldgof, K. Bowyer, D.W. Eggert, A. Fitzgibbon, and R.B. Fisher. An experimental comparison of range image segmentation algorithms. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 18(7):673–689, Jul 1996.

[305] P. Saint-Marc and G. Medioni. Adaptive Smoothing for Feature Extraction. In *Proceedings of the Image Understanding Workshop*, pages 1100–1113, Cambridge, Mass., Apr 1988.

[306] P.J. Rousseeuw and A.M. Leroy. *Robust Regression and Outlier Detection.* John Wiley & Sons, New York, 1987.

[307] P. Perona and J. Malik. Scale-space and edge detection using anisotropic diffusion. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(7):629–639, Jul 1990.

[308] D. W. Marquardt. An algorithm for least-squares estimation of non-linear parameters. *Journal of the Society for Industrial and Applied Mathematics*, 11:431–441, 1963.

[309] S. Lakshmivarahan and S. K. Dhall. *Analysis and design of parallel algorithms: arithmetic and matrix problems.* McGraw-Hill, 1990.

[310] G. Golub and C. Reinsch. Singular Value Decomposition and Least Squares Solutions. *Numer. Math.*, 14:403–420, 1970.

[311] T. F. Chan. An Improved Algorithm for Computing the Singular Value Decomposition. *ACM Transactions on Mathematical Software*, 8(1):72–83, Mar 1982.

[312] R. Paige, J. Reif, and R. Wachter. *Parallel Algorithm Derivation and Program Transformation.* Kluwer Academic, 1993.

[313] P. Fritzson and L. Finmo. *Parallel Programming and Applications.* IOS Press, 1995.

[314] W. Gentzsch and U. Harms. *High Performance Computing and Networking*, volume 797 of *LNCS.* Springer-Verlag, Munich, 1994.

[315] V. P. Kumar, P. S. Gopalakrishnan, and L. N. Kanal. *Parallel Algorithms for Machine Intelligence and Vision.* Springer-Verlag, 1990.

[316] D. Bjorner. *Abstract Software Specifications*, volume 86 of *LNCS.* Springer-Verlag, 1980.

[317] E. R. Olderog. *Programming Concepts, Methods and Calculi.* North-Holland, 1994.

[318] A. P. Wim Bohm and J. T. Feo. *High Performance Functional Computing.* 1995.

[319] J. Bezivin, J.-M. Hullot, P. Cointe, and H. Lieberman. *ECOOP '87 European Conference on Object-Oriented Programming*, volume 276 of *LNCS.* Springer-Verlag, 1987.

[320] P. Nixon. *A Parallel Programming Methodology Based on Paradigms.* IOS Press, 1995.

[321] M. Kara et al. *Abstract Machine Models for Parallel and Distributed Computing.* IOS Press, Apr 1996.

[322] A. Veronis and Y. Parker. *Transputer Research and Applications*, volume 5. IOS Press, 1992.

[323] J. R. Davy and P. M. Dew. *Abstract Machine Models for Highly Parallel Computers.* Oxford University Press, 1995.