Design Patterns for Parallel Vision Applications

Sanjay S. Kadam

A thesis submitted for the degree of
Doctor of Philosophy
in the University of London

University College London
Department of Computer Science

June 1998
Abstract

Computer vision is a challenging application for high performance computing. To meet its computational demands, a number of SIMD and MIMD based parallel machines have been proposed and developed. However, due to high costs and long term design times these machines have not been widely used. Recently, network based environments, such as a cluster of workstations, have provided effective and economical platforms for high performance computing. But developing parallel applications on such machines involves complex decisions about distribution of processes over the processors, scheduling of processor time between competing processes, communication patterns, etc. Writing explicit code to control these decisions increases program complexity and reduces program reliability and code re-usability.

We propose a design methodology based on design patterns which is intended to support parallelization of vision applications on a cluster of workstations. We identify common algorithmic forms occurring repeatedly in parallel vision algorithms and formulate these as design patterns. We specify various aspects of parallel behaviour of a design pattern, such as process placement or communication patterns, in its definition or separately as issues to be addressed explicitly during its implementation. Design patterns ensure program reliability and code re-usability since they capture the essence of working designs in a form that makes them usable in different situations and in future work.

The research work is concerned with presenting a catalogue of design patterns to implement various forms of parallelism in vision applications on a cluster of workstations. Using relevant design patterns, we implement representative vision algorithms in low, intermediate and high level vision tasks. Majority of these implementations show promising results. For example, given a 512x512 image, the image restoration algorithm based on Markov random field model can be completed in less than 45 seconds on a network of 16 workstations (Sun SPARCstation 5). The same task takes more than 10 minutes on a single such workstation.
Acknowledgements

I thank my supervisors Dr. Graham Roberts and Prof. Bernard Buxton for providing invaluable suggestions, moral support and cordial atmosphere for conducting this research work at Department of Computer Science, University College London.

I am obliged to the Association of Commonwealth Universities for financing my research work through the British Council in the form of Commonwealth Scholarship. I also thank Dr. Vijay P. Bhatkar, Executive Director, C-DAC (Center for Development of Advanced Computing), Pune, India, for approving the required study leave (from my current employment) for completion of my Ph.D work.

I am grateful to many colleagues and friends with whom I had both academic and non-academic interactions. My special thanks go to Jonathan Poole for introducing me to the concepts in parallel and distributed computing using UC++ (a concurrent extension of C++). I also thank Dr. Julia Schnabel for providing me with an application in medical imaging which served as an excellent example for parallelization as discussed in chapter 6. I am also thankful to Dr. Niladri Chatterjee for reviewing the early drafts of this thesis and providing numerous suggestions towards enhancing the technical quality of the material presented. Many thanks to Kamalendu Pal, Arif Iqbal, Adil Qureshi, Ihsan Khan, Giannis Koufakis, Dr. Bill Langdon and other researchers in the department for several discussions on both technical and social aspects of student life in London.

I express utmost gratitude to my parents and relatives for all their good wishes and blessings. I am also indebted to my wife Dr. Pratima for her continuous support, encouragement and concern for the completion of this research work. Her adjustments to the seclusion and frequent disruptions to the family life during the course of this study were highly appreciable. And finally, thanks to our little sons KirtiRaj and Pranav who with their ever cheerful appearance and innocent but invigorating smiles helped me to relieve myself from the tensions and hardships of the student life.
Contents

1 Introduction ................................................................. 16

1.1 Overview ................................................................................................................................. 16

1.2 Aims of this Research Work ................................................................. 20

1.3 Contributions of the Dissertation ................................................................. 21

1.4 Organization of the thesis ................................................................. 22

2 Parallelism in Computer Vision ...................................................... 24

2.1 Parallel Computing ............................................................................................................... 25

2.1.1 Parallel computing systems ..................................................................................... 25

2.1.2 Algorithmic classes ................................................................................................... 27

2.1.3 Performance of parallel programs ...................................................................... 28

2.2 An Overview of Computer Vision ................................................................. 29

2.2.1 Object recognition in 2D scenes ........................................................................ 30

2.2.2 Feature Detection ................................................................................................. 31
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.5.3 Use of clusters</td>
<td>47</td>
</tr>
<tr>
<td>2.5.4 Parallel computing using Clusters</td>
<td>48</td>
</tr>
<tr>
<td>2.6 Parallelization using Design Patterns</td>
<td>50</td>
</tr>
<tr>
<td>2.6.1 Design patterns</td>
<td>50</td>
</tr>
<tr>
<td>2.6.2 Forms of Parallelism in Vision</td>
<td>52</td>
</tr>
<tr>
<td>2.6.3 Design patterns for parallel vision</td>
<td>55</td>
</tr>
<tr>
<td>2.7 Related work</td>
<td>56</td>
</tr>
<tr>
<td>2.8 Summary</td>
<td>58</td>
</tr>
</tbody>
</table>

### 3 Design patterns for parallelizing vision applications

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Organization of patterns</td>
<td>61</td>
</tr>
<tr>
<td>3.2 Description of design patterns - a template</td>
<td>63</td>
</tr>
<tr>
<td>3.3 Farmer-Worker Pattern</td>
<td>65</td>
</tr>
<tr>
<td>3.4 Master-Worker Pattern</td>
<td>70</td>
</tr>
<tr>
<td>3.5 Controller-Worker Pattern</td>
<td>76</td>
</tr>
<tr>
<td>3.6 Divide-and-Conquer Pattern</td>
<td>82</td>
</tr>
<tr>
<td>3.7 Temporal Multiplexing Pattern</td>
<td>87</td>
</tr>
<tr>
<td>3.8 Pipeline Pattern</td>
<td>91</td>
</tr>
<tr>
<td>3.9 Composite Pipeline Pattern</td>
<td>98</td>
</tr>
<tr>
<td>3.10 Summary</td>
<td>104</td>
</tr>
</tbody>
</table>
Contents

4 Low level algorithms 106

4.1 Parallelization of low level algorithms ........................................... 109

4.2 Partitioning the image data ......................................................... 110

4.3 Grey scale transformations .......................................................... 112

4.4 Image filtering ............................................................................. 113
  4.4.1 Convolution ............................................................................. 114
  4.4.2 Rank filtering ........................................................................ 119
  4.4.3 Spatial filters ......................................................................... 120

4.5 Fast Fourier transforms ............................................................... 122

4.6 Image restoration .......................................................................... 124
  4.6.1 Markov random field models for image recovery .................... 124

4.7 Summary ..................................................................................... 128

5 Intermediate level processing 130

5.1 Region based segmentation ........................................................ 132

5.2 Parallel Region-based segmentation ............................................ 134

5.3 Segmentation using Perceptual Organization ............................... 138
  5.3.1 Sequential Line grouping algorithm .................................... 139
  5.3.2 Parallel Line grouping algorithm ......................................... 143

5.4 Summary ..................................................................................... 145
## Contents

6 High level processing

6.1 Sequential geometric hashing algorithm

6.1.1 Preprocessing Phase

6.1.2 Recognition phase

6.2 Parallel geometric hashing algorithm

6.3 Multi-scale active shape description - an application

6.3.1 An overview of the shape description process

6.4 Parallelization of the shape description process

6.4.1 Parallelization using Temporal Multiplexing pattern

6.4.2 Parallelization using Pipeline pattern

6.4.3 Parallelization using Composite Pipeline pattern

6.5 Summary

7 Conclusion

7.1 Aims and Motivation

7.2 Research Review

7.3 Contributions of the Research work

7.4 Comparison with related work

7.5 Future work

A Notation
List of Figures

2.1 An overview of a typical vision based application ......................... 31
2.2 Processing levels in a typical vision based application ................. 37
2.3 A 4-connected mesh, pyramid and a 3-dimensional hypercube of processing elements .......................................................... 40
2.4 Shared memory machines (interconnected by a bus and switching network) and systolic/pipeline systems ................................. 43
2.5 Common cluster configurations: bus, star and a ring ...................... 45
3.1 Farmer-Worker Pattern ................................................................. 66
3.2 Object Interaction in the Farmer-Worker Pattern ............................ 67
3.3 Master-Worker Pattern ................................................................. 72
3.4 Object Interaction in the Master-Worker Pattern ............................ 73
3.5 Controller-Worker Pattern ........................................................... 77
3.6 Object Interaction in the Controller-Worker Pattern ...................... 78
3.7 Convolution masks for finding a) horizontal edges and b) vertical edges 82
3.8 DC Pattern .................................................................................. 83
List of Figures

3.9 Object Interaction in the DC Pattern .................................................. 84
3.10 TM Pattern .......................................................................................... 88
3.11 Object Interaction in the TM Pattern .................................................. 89
3.12 Vehicle identification system .............................................................. 92
3.13 Pipeline Pattern .................................................................................. 93
3.14 Object Interaction in the Pipeline Pattern .......................................... 94
3.15 Vehicle identification system .............................................................. 99
3.16 Composite Pipeline Pattern ............................................................... 100
3.17 Object Interaction in the Composite Pipeline Pattern ....................... 101

4.1 Partitioning of an image. a) Row partitioning b) Row partitioning with
data that is to be overlapped and/or communicated .................................. 111
4.2 Performance of histogram equalization ............................................... 113
4.3 Performance of the convolution operation using a 3x3 window .......... 115
4.4 Performance of the convolution operation using a 15x15 window ....... 116
4.5 Performance of the convolution operation on a 1Kx1K image .......... 117
4.6 Performance of the Farmer-Worker pattern in convolution operation on
varying the processor load and number of subtasks (window size 15x15) ... 118
4.7 Performance of the sharpening operation using spatial filters (window size
11x11) ........................................................................................................ 121
4.8 The data blocks needed to transpose the intermediate results .......... 123
List of Figures

4.9 Performance of the image restoration algorithm using MRF model (window size 3x3) ................................................................. 126

4.10 Performance of the Master-Worker pattern (in image recovery operation using the MRF model on a 512x512 image) subject to the external load and load distribution .............................................. 128

5.1 a) Partitioned image b) Corresponding quadtree .................................................. 133

5.2 a) Distribution of subimages b) Merging of subimages ............................................. 135

5.3 Performance of the parallel split and merge segmentation algorithm ...................... 136

5.4 Line Grouping ................................................................................................................. 139

5.5 Relational constraints in the line grouping algorithm a) proximity b) collinear-
ity and c) continuation ...................................................................................................... 141

5.6 Indexing technique used in the line grouping process. a) search area for the base line b) the index array .................................................. 142

6.1 Preprocessing phase in the geometric hashing algorithm a) Orthogonal co-
ordinate system defined by the basis set b) Adding (model, basis) pairs in
the hash table ....................................................................................................................... 150

6.2 Recognition phase in the geometric hashing algorithm a) Orthogonal coordinate system defined by the basis set b) Accessing and collecting (model, basis) pairs from the hash bins in hash table .............................................. 151

6.3 Hash table data structure a) symmetric indexing in hash table b) hash entries in normal hash table c) reduction in hash entries using symmetries 154

6.4 Performance of the geometric hashing algorithm for object recognition .................. 156
6.5 Multi-scale shape description process a) propagation step applied on a set of five image slices b) multi-scale shape stack of an image slice computed in the shape focusing step (Figure (b) adapted from (Schnabel, 1997))... 160

6.6 Shape focusing performed at different scales in the image scale-space of an image slice using active contour models: (a) $\sigma = 8$ (b) $\sigma = 4$ (c) $\sigma = 2$ (d) $\sigma = 1$. Image (a) also contains the initial contour superimposed in black. All images are taken from (Schnabel, 1997)... 160

6.7 Visualization of the stack contours (those displayed in Figure 6.6) stacked using triangulation. Image taken from (Schnabel, 1997)... 161

6.8 Parallelization of the shape description process using a Pipeline pattern. The integer values denote sequential time (in seconds) required for executing corresponding components of the Pipeline pattern... 164

6.9 Parallelization of the multi-scale shape description process using a Composite Pipeline pattern... 167
List of Tables

4.1 Execution time in \( \text{min:sec} \) for histogram equalization ........................................... 113

4.2 Execution time in \( \text{min:sec} \) for the convolution operation ................................. 114

4.3 Performance of the Farmer-Worker pattern on varying the external load and number of subtasks. The execution time \( \text{min:sec} \) displayed are for the convolution operation \( \text{window size 15x15} \) ......................................................... 118

4.4 Execution time in \( \text{min:sec} \) for the rank filtering operation .......................... 120

4.5 Execution time in \( \text{min:sec} \) for the sharpening operation ............................... 121

4.6 Execution time in \( \text{min:sec} \) for FFT operation ......................................................... 123

4.7 Execution time in \( \text{min:sec} \) for image restoration using MRF model ................. 125

4.8 Performance of the Master-Worker pattern when subjected to the external load. The execution times \( \text{min:sec} \) displayed are for the image restoration operation using the MRF model on a 512x512 image ......................................................... 127

5.1 Execution time in \( \text{min:sec} \) for the parallel split and merge segmentation algorithm .......................................................................................................................... 136

5.2 Execution time in \( \text{min:sec} \) for various operations in the parallel split and merge segmentation algorithm applied on a 512X512 image ......................................................... 137
List of Tables

5.3 Execution time in (min:sec) for the line grouping process ............... 144

6.1 Execution time in (min:sec) for the geometric hashing algorithm ....... 156

6.2 Execution times in (seconds) for different implementations and individual steps of the shape description process ......................... 168
Chapter 1

Introduction

1.1 Overview

Computer Vision deals with the principles and techniques to extract and interpret useful information in a scene by capturing and analyzing images of that scene. It has applications in several areas such as remote sensing, autonomous vehicle guidance, industrial inspection, and medical imaging. Some of these applications, such as autonomous vehicle guidance, are real-time and involve algorithms which must complete their computations within a fraction of a second. Some applications requiring human interaction are interactive and must complete within few seconds or less, depending on the type of interaction required. Other applications are batch applications which can tolerate maximum latency of few hours or even days. The nature of the algorithms involved in these applications is thus varied. But most of these algorithms are computationally intensive and require enormous computing power for their practical implementation.

Computer vision uses a broad spectrum of algorithms covering different areas such as image and signal processing, graph theory, mathematics, and artificial intelligence. From a computational perspective, vision processing is conveniently classified into three levels: low, intermediate, and high. Low level processing involves pixel-based transformations where uniform computations are applied at each pixel or a neighborhood around each
pixel in the image data. These computations are mainly numeric and well structured. Intermediate level processing involves both numeric and symbolic computations. It comprises algorithms to form regions of interest in the image data, such as grouping of low level features (e.g. edges) into lines, arcs, or rectangular borders of an object. High level processing involves symbolic computations where data provided by the low and intermediate level algorithms is used for testing and generating hypotheses for object recognition. A typical vision application comprises low, intermediate and high level vision tasks/algorithms and thus involves both numeric and symbolic computations. Therefore, although vision has been identified as a grand challenge application for high performance computing, computational characteristics of vision applications are different from the structured number crunching computations arising in most other grand challenge applications (Wang et al., 1996).

To meet the computational demands of vision tasks, several efforts have been directed towards providing a high performance computing support for their practical implementation. A brief survey of the research efforts in high performance parallel computing for vision can be found in (Webb, 1994). These efforts can broadly be grouped into following categories, based on the type of computing platforms they utilize: special-purpose hardware chips, SIMD based machines, specialized vision systems and general purpose parallel machines. Special-purpose hardware chips serve as accelerators to specific vision algorithms since they implement the computations in hardware. However, they are suitable only for specific well-structured low level vision algorithms, such as image convolution.

SIMD based multiprocessor machines such as meshes, array processors, hypercubes, and pyramids, consist of simple processing elements connected by a communication network. These machines perform well for implementing most of the low level vision algorithms. But they are not well suited for high level vision algorithms since these algorithms involve nonuniform processing and complex data structures. Specialized vision systems are special purpose parallel machines designed to suit the requirements of vision tasks. They are capable of being partitioned into one or more independent SIMD and MIMD subsystems to match the computational characteristics of vision algorithms at various levels. For example, the image understanding architecture (IUA) (Weems et al., 1989) has three hierarchical levels of computing platforms to support processing of low, intermediate and
high level vision tasks. Specialized vision systems, however, have complex architectures which involve significant design and development effort. The need to develop new system software for such machines results in huge system development costs.

General purpose parallel machines such as IBM SP-2, Meiko CS-2, Intel Paragon, Cray T3D, and SGI Power Challenge, have been used successfully for a variety of high performance computing applications. These commercial machines have not been developed for any specific applications, but are meant to be general purpose systems. Most of them have a similar architecture consisting of processors interconnected by a high speed network. These processors are those that are used in large uniprocessor workstations. These machines are typically organized as a single box that contains all the processor and memory modules interconnected by a special purpose interconnection network. Although there have been some attempts to use these machines for parallel vision applications (Wang, 1995), they are still not very popular with many organizational setups.

Recently, network-based computing environments, such as a cluster of workstations, have provided effective and economical platforms for high performance computing. A cluster of workstations offers several advantages for parallelizing and executing large applications on a relatively low-priced and readily available pool of machines. It provides multiple CPUs for parallel computing and dramatically improves virtual memory and file system performance. It can approach or exceed supercomputer performance for some applications and can easily be tuned to advances in processor and network technology (Anderson et al., 1995), (Turcotte, 1996). A cluster of workstations can incorporate heterogeneous architectures, so applications can select the most suitable computing resources for each computation.

But developing parallel applications on such machines involves complex decisions about distribution of processes over the processors, process synchronization, scheduling of processor time between competing processes, communication patterns, etc. Writing explicit code to control these decisions increases program complexity and, reduces program reliability and code reusability. Also, the available machines and their capabilities can vary from one execution to another, and high communication costs can degrade the performance in many applications. Moreover, developers do not wish to spend time in low level parallel
programming in order to gain the advantages of potential parallelism in an application. Most of them use or modify existing parallel code to implement parallelism for their applications. In fact, some recent surveys of experienced parallel programmers have shown that about 69% modify existing programs or compose programs from existing blocks of code. The remaining 31% who start from scratch are typically computer scientists and applied mathematicians (Pancake, 1996).

The main goal of this thesis is to present a design methodology based on design patterns intended to support parallelization of vision applications on a cluster of workstations. Most of the parallel algorithms used in implementing vision tasks repeatedly use only a finite set of algorithmic forms. We identify these common algorithmic forms and formulate these as design patterns. We specify various aspects of parallel behavior of a design pattern, such as process placement and communication patterns, in its definition or separately as issues to be addressed explicitly during its implementation. Design patterns ensure program reliability and code reusability since they capture the essence of working designs in a form that makes them usable in different situations and in future work (Coplien & Schmidt, 1995). The use of the design patterns would enable development of distributed software quickly, economically and reliably. Using a cluster of workstations, researchers can use the design patterns to implement many interactive and batch applications in computer vision.

A cluster of workstations is characterized by high communication costs and a variation in speed factors of individual machines in the network. We need to address these issues while formulating the design patterns. One factor that minimizes the effect of high communication costs on performance is granularity. Granularity of an algorithm describes the amount of work associated with each task relative to communication. An algorithm that exchanges data between its processes after a small number of computations is called fine-grained while an algorithm where the computations continue for a long time before the communication is required is termed as coarse-grained. Since a cluster of workstations is inherently coarse-grained, we need to formulate design patterns so that they implement coarse-grained parallelism. Also, the design patterns should distribute the work load according to the speed factors of individual machines in the network.

We begin our work by analyzing the computation and communication characteristics
of vision algorithms. We identify various forms of parallelism in vision algorithms and formulate design patterns to implement them. Each design pattern captures common designs used by developers to parallelize their applications. We present a catalogue of design patterns to implement various forms of parallelism in vision applications on a cluster of workstations. Using relevant design patterns, we implement representative vision algorithms in low, intermediate and high level vision tasks, and present the experimental results of the corresponding parallel implementations.

In low level, we implement algorithms such as histogram equalization, convolution, image filtering using spatial filters, and image restoration using Markov random field models. In intermediate level, we implement region-based split and merge segmentation algorithm and line grouping algorithm based on principles of perceptual grouping. In high level, we implement geometric hashing algorithm for object recognition. We also discuss parallelization of an application in medical imaging, namely, multi-scale active shape description of MR (magnetic resonance) brain images using active contour models.

1.2 Aims of this Research Work

The focus of the work in this thesis is to develop methodologies to support parallelization of vision applications on a cluster of workstations. The main goals of this thesis work are:

- To analyze computational characteristics of vision tasks and identify common algorithmic structures in their parallel implementations.
- To capture and articulate these algorithmic structures as design patterns in a form that makes them usable in different situations and in future work.
- To use these design patterns for implementing some representative vision algorithms in low, intermediate and high level vision processing.
- To evaluate the viability of using a cluster of workstations to parallelize vision applications.
1.3 Contributions of the Dissertation

The contributions of this dissertation are three fold. Firstly, we propose a design methodology based on design patterns intended to support parallelization of vision applications on a cluster of workstations. Secondly, we present coarse-grained parallel algorithms for some representative vision algorithms in low, intermediate and high level vision processing. Thirdly, we use relevant design patterns to implement these parallel algorithms on workstation clusters. These contributions are summarized as follows:

- Design patterns: We identify common algorithmic structures occurring repeatedly in parallel vision tasks/applications and formulate these as design patterns. We describe each design pattern using a *template* which outlines intent, motivation, structure, interaction amongst the components and applicability of the design pattern. This description enables selection and use of a design pattern in different situations and in future work.

- Coarse-grained parallel algorithms: We present coarse-grained parallel algorithms and implementations for several vision tasks such as convolution, image filtering, image restoration, region-based segmentation, line grouping, and geometric hashing algorithm for object recognition. We also present different parallel implementations of the multi-scale active shape description process (an application in medical imaging) using different design patterns.

- Implementation on a cluster of workstations: Using relevant design patterns, we perform parallel implementations of the selected representative vision tasks stated above. The results of these implementations enable critical assessment of the design patterns for achieving improvements in application performance. It also enables evaluating the viability of using workstation clusters for implementing parallel vision applications.
Chapter 1. Introduction

1.4 Organization of the thesis

The remainder of the thesis is organized as follows:

- Chapter 2 reviews concepts and methods in several different areas related to the parallel vision systems. We begin with a brief introduction to parallel computing systems and parallel algorithms. We then describe general principles and methods used in the field of computer vision, giving specific emphasis on applications involving analysis of 2D scenes. We also describe the computational characteristics of vision algorithms and outline SIMD and MIMD based parallel machines used for parallelizing these algorithms. We then describe parallel computing on workstation clusters and discuss their advantages over the conventional parallel machines. We present various forms of parallelism in vision algorithms and introduce the concept of design patterns intended to support parallelization of vision applications on a cluster of workstations. Finally, we outline some of the leading research efforts related to the work presented in this thesis.

- Chapter 3 presents a detailed description of each design pattern. We use a template to specify various aspects of parallel behavior (such as process placement and communication patterns) of each design pattern. The templates outline intent, motivation, structure, interaction amongst the components and applicability of the design patterns.

- Chapter 4 discusses parallelization of some low level vision algorithms such as histogram equalization, convolution, image sharpening using spatial filters, fast fourier transforms, and image restoration using Markov random field models. Each algorithm is parallelized by using either Farmer-Worker, Master-Worker or Controller-Worker pattern.

- Chapter 5 presents results of parallelization of some intermediate level algorithms such as region-based segmentation, and line grouping algorithm based on the principles of perceptual organization. We use Divide-and-Conquer pattern for implementing the parallel region-based segmentation algorithm. The line grouping algorithm is parallelized by using the Controller-Worker pattern.
Chapter 6 presents results of parallelization of a high level vision algorithm, namely, geometric hashing for object recognition. We use a Farmer-Worker pattern to perform multiple matching operations (probes) for identifying each object in an image. In the last section of this chapter, we discuss parallelization of an application in medical imaging, namely, multi-scale active shape description of MR (magnetic resonance) brain images using active contour models. We discuss three different approaches of parallelizing the shape description process. Each approach uses a different design pattern, namely, Temporal Multiplexing, Pipeline or Composite Pipeline.

- Finally, chapter 7 presents concluding remarks and directions for future research.
Chapter 2

Parallelism in Computer Vision

Computer vision is a challenging application for high performance computing. Many vision applications are computationally intensive and involve complex processing. For a practical and real-time implementation of vision applications, high-performance computing support is essential. Over the past several years, parallel processing has been perceived to be an attractive and economical way to achieve the required level of performance in vision applications. Computational demands and real-time constraints associated with the vision applications have induced several research efforts to explore the use of parallel computing resources for parallelizing vision applications (Webb, 1994). Most vision applications consist of image preprocessing followed by object identification. Although both these tasks involve large number of computations, they embody different computational paradigms. As a result, several special and general purpose parallel machines have been proposed, developed and used in implementing parallel solutions to many vision algorithms.

This chapter gives an overview of the algorithms in computer vision and presents parallel systems and methodologies used in parallelizing vision applications. The chapter is organized as follows. Section 2.1 introduces some concepts in parallel computing. Section 2.2 gives an overview of the principles and methods involved in the field of computer vision. Section 2.3 discusses computational characteristics of vision applications and their classification into three levels, low, intermediate and high. Section 2.4 outlines different parallel systems used for parallelizing vision applications. Section 2.5 describes parallel
computing on a cluster of workstations. Section 2.6 proposes a methodology, based on design patterns, which can be used to parallelize a majority of the vision applications on network-based machines, such as a cluster of workstations. We also describe various forms of parallelism that can be applied to parallelize vision applications. Finally, section 2.7 outlines some of the leading research efforts which have been inspirational to the work presented in this thesis.

2.1 Parallel Computing

Parallel computing is concerned with applying multiple processors to solve a single computational problem for achieving better performance. This section begins with an introduction to parallel computing systems. It is followed by a description of abstract algorithmic classes characterizing different parallel algorithms. These classes are useful when discussing algorithms at a higher level.

2.1.1 Parallel computing systems

A parallel computer is a collection of processors and memory connected by some type of communication network. Parallel computing systems include a full spectrum of sizes and prices, from a collection of workstations attached to a local-area network, to an expensive high-performance machine with thousands of processors connected by high-speed switches (Duncan, 1992).

The architectures of the computing systems are commonly organized in terms of instruction streams and data streams ( Flynn, 1972). The three cases that have become familiar terms to the parallel programmer are SISD (single instruction, single data), SIMD (single instruction, multiple data) and MIMD (multiple instruction, multiple data). SISD computers are the traditional von Neumann computers that have a single instruction stream and a single data stream. All operations on these computers are logically sequential. In a SIMD parallel computer a single instruction stream is applied to multiple data streams. SIMD-based machines usually consist of a large number of simple processors
connected by an interconnection network. The MIMD data model is the most general model of a parallel computer. A MIMD computer has multiple processing elements each of which is a complete computer in its own right.

Although SIMD systems are easy to program, optimizing SIMD programs to yield acceptable performance is very difficult. As a result, SIMD computers have not been very popular for scientific computing. This makes MIMD systems the overwhelming majority of parallel systems, especially when a cluster of workstations is viewed as a single MIMD computer. A MIMD computer consists of processors and memory. The memory can be either shared or distributed among the processors. We can therefore consider two distinct programming models: shared memory MIMD and distributed memory MIMD. However, since the same issues of data locality and concurrency arise in both the cases, we can view MIMD computer in terms of a common programming model. One such model is the coordination model (Mattson, 1996), (Foster, 1995), where a parallel computation is viewed as a collection of distinct processes which interact at discrete points through a coordination operation. The term coordination refers to the basic operations to control a parallel computer. It includes coordination operations for information exchange, process synchronization and process management. These coordination operations may vary in speed and structure, however, the overall model is essentially the same.

But describing parallel and distributed computers in terms of a coordination model is not universally accepted like the von Neumann model (Mattson, 1996). However, such a model can be stated and used for programming parallel computers within a universal programming model. Although the computer systems differ, the difference is granularity (ratio of computation to communication), and not the fundamental programming model (Mattson, 1996).

The programming model, in order to be useful, must be implemented as a programming environment. There are several programming environments supporting various incarnations of the coordination model which run well on parallel computers as well as a cluster of workstations (Turcotte, 1993), (Cheng, 1993). One can develop a parallel code using some high level language designed specifically to support parallel and distributed computing. Alternatively, one can use a sequential language combined with a coordination library
(often called as message-passing library), such as PVM (Sunderam, 1990).

Programs written for parallel MIMD systems fall into two categories: SPMD (single program multiple data) and MPMD (multiple program multiple data). For SPMD programs, each processor executes the same object code. SPMD style of programming is easy to code since the programmer needs to maintain a single source code. In contrast, MPMD programs allow each processor to have a distinct executable code. A programmer can split the program into different modules which can be developed and debugged independently or reused as components of other programs. A MPMD program requires less memory compared to its equivalent SPMD version (Mattson, 1996).

2.1.2 Algorithmic classes

Most of the parallel algorithms can be classified in terms of the regularity of the underlying data structures (space) and the synchronization required as these data elements are updated (time) (Angus et al., 1989), (Mattson, 1996). Based on this classification scheme there are four general classes of parallel algorithms:

1. Synchronous

Synchronous algorithms are those in which regular data elements are updated at regular intervals of time. They are regular in space and regular in time. They involve tightly coupled manipulation of identical data elements. Synchronous algorithms can be expressed in terms of a single instruction stream, and are therefore easily mapped onto SIMD computers. The parallelism is usually expressed in terms of the decomposition of the data. In fact, the data drives the parallelism, hence the name data parallelism. However, data parallelism is more general than SIMD parallelism, since data parallelism does not insist on a single instruction stream.

2. Loosely synchronous

A loosely synchronous algorithm synchronously updates data elements which differ from one processor to another. Loosely synchronous algorithms are regular in time but irregular in space. They have tight coupling between the tasks as in the syn-
chronous case. However, due to variation in the data elements across the processors, the work loads can vary from processor to processor. Hence, loosely synchronous algorithms need some mechanism to balance the computational load among the processors of the parallel computer.

3. Asynchronous

Asynchronous algorithms do not have regular data updates, so the system proceeds with nonuniform and sometimes random synchronization. These algorithms are irregular in time and usually irregular in space with unpredictable or nonexistent coupling between the tasks. This class of problems, other than the embarrassingly parallel subset described next, is most rare. This is because programs for implementing asynchronous algorithms are difficult to construct. While synchronous and loosely synchronous algorithms are usually parallelized by focusing on data decomposition, asynchronous algorithms are usually parallelized by decomposition of the control, which is referred to as functional or control parallelism.

4. Embarrassingly parallel

Embarrassingly parallel algorithms are those asynchronous algorithms for which the tasks are completely independent and uncoupled. The parallelism in this case is trivial and the programs are among the simplest parallel programs to construct. Problems in this class are very common in parallel computing since their computations easily map into this model. In fact, any program consisting of a loop with compute-intensive and independent iterations can be parallelized using this model. Embarrassingly parallel programs usually utilize an SPMD style of programming combined with some mechanism for load balancing. Load balancing schemes can either be static or dynamic.

2.1.3 Performance of parallel programs

The main goal of parallelism is to reduce the execution time of the whole program in order to produce the results faster. The performance estimates of a parallel program are based on the timings of its complete sequential code. The sequential program typically comprises of two distinct sections of code, inherently sequential code and potentially parallel code.
Chapter 2. Parallelism in Computer Vision

The parallel content $p$ of the program is defined as the ratio of the time taken to execute the potentially parallel code upon the time taken to execute the whole code. The maximum theoretical speedup that can be achieved for a given program is a function of the parallel content $p$ and the number of processors that will be used ($N$). It is given by Amdahl's law (Amdahl, 1988) which is stated as follows

$$
\text{Theoretical speedup} = \frac{1}{(1-p) + (p/N)}
$$

(2.1)

The theoretical speedup is lower than the ideal speedup, which reflects the ideal that applying $N$ processors to a program should cause it to complete $N$ times faster. The size of the gap between the ideal and theoretical speedup is a function of the serial content of the program. This suggests that the amount of speedup that can be achieved for every program is limited beyond a certain number of processors. The gap between the theoretical and ideal speedup may change due to the increase in problem size (e.g. when number of iterations are increased in a simulation). The gap narrows down when the parallel content of the program increases due to increase in problem size, while the gap may actually widen if the length of the serial bottlenecks also increase upon increase in problem size. However, the theoretical speedup is rarely achievable by a parallel application. There will actually be an observed speedup which is much lower than the theoretical speedup, reflecting the effect of external overhead on the total execution. This overhead comes from two sources a) the additional processor cycles expended in simply managing the parallelism b) wasted time spent waiting for I/O, communication among processors, and, competition from the operating system and other users (Pancake, 1996). Theoretical speedup does not take these factors into account.

2.2 An Overview of Computer Vision

The basic input in computer vision is a set of one or more image(s) of some scene, while the output is a description of the objects in that scene. An image, captured by a sensor, is an array of numbers called pixels that represent average brightness (gray level) or color values at discrete grid points in the scene. A gray level is usually represented as an 8-bit
integer having 256 distinct values, while each color value is represented by a $n$-valued tuple measuring brightness in a set of $n$-spectral bands (e.g., red, blue and green).

We can view image processing as a prelude to computer vision. Image processing algorithms operate on images to extract and represent scene information. Higher level vision algorithms use scene information for object recognition and scene interpretation. Computer vision therefore encompasses processing from sensing to scene interpretation. The main areas of image processing include *image enhancement and restoration* (to improve appearance of an image or to undo effects of image degradations such as blurring or noise), *image compression* (to reduce an image to smaller sets of data which can be used for reconstruction of an acceptable approximation to the original image), and *image reconstruction from projections* (to construct images of cross-section of an object by analyzing a set of projections taken from different directions, as in tomography).

Since majority of the applications in computer vision involve two dimensional (2D) scenes and the general goal is to recognize objects of interest in the images of these scenes, we will restrict our discussion to analysis of 2D scenes. The following subsections outline general techniques involved in 2D object recognition. A detailed discussion dealing primarily with 2D vision can be found in (Ballard & Brown, 1982), (Rosenfeld & Kak, 1982), (Sonka et al., 1993), while an outline of both 2D and 3D vision is given in (Rosenfeld, 1988).

### 2.2.1 Object recognition in 2D scenes

Some examples of applications involving 2D scenes are: recognition of alphanumeric characters from an image of a document, recognition of blood cells from an image of a specimen seen through a microscope, and identification of houses and roads from high altitude aerial photographs. A general framework describing major techniques used in object recognition is shown in Figure 2.1. Feature detection techniques are used for detecting local features such as edges (at which the gray level changes abruptly), lines, curves, spots, and corners. Segmentation partitions the image pixels into homogeneous regions. Both segmentation and feature detection assign labels to the image pixels which indicate the classes to which
Chapter 2. Parallelism in Computer Vision

the pixels belong.

Recognition/Generic Description
   Model Matching/Object Recognition

Relational structure
   Segmentation/Resegmentation
   Property Measurement

Scene Features
   Feature Detection
   Image enhancement/restoration

Digitized Image of the Scene
   Imaging device

Real-World scene

Illumination

Figure 2.1: An overview of a typical vision based application

Resegmentation techniques group the segmented regions in the image into groups or parts that satisfy certain geometric constraints. Property measurement algorithms compute various properties such as area, perimeter, and average gray level, for such parts. Model matching or object recognition is then regarded as identification of image parts that correspond to the object parts and satisfy the appropriate constraints.

2.2.2 Feature Detection

We describe basic feature detection techniques used for detecting various local features in the image.

1. Templating

A subimage of a local feature that is to be detected is regarded as a template and matched at every possible position in the image for best fit. The degree of match or mismatch identifies the feature at the corresponding pixels. Thus if \( f(i,j) \) and \( t(i,j) \) represent pixel intensities in the image and the template, respectively, a measure of the mismatch between them can be expressed by (Rosenfeld, 1988)\[
D_{(a,b)}(f,t) = \sum_i \sum_j (f(i+a,j+b) - t(i,j))^2,
\]
where \((a,b)\) is the displacement of the origin of \( t \).
relative to that of $f$. The value of $(a, b)$ that minimizes $D$ represents the most likely position of the template in the image. This method is computationally intensive and does not give correct results if the image intensity varies significantly over areas of the size of the template.

2. Edge detection

Edge detection techniques attempt to find pixels that lie on the borders between different objects in the image. Some standard approaches used are (Rosenfeld, 1988):

- **Mask matching**: A template representing ideal edges in various orientations is matched in the neighborhood of each pixel in the image. A pixel is classified as an edge pixel if the degree of such a match is sufficiently high. Sharp matches are obtained by using masks which are second differences of ideal step (or ramp) edges. This technique is also used for detecting lines, curves, spots and corners.

- **Gradient magnitude**: If $\Delta_x$ and $\Delta_y$ denote the first differences of the image gray level in the $x$ and $y$ directions, then the direction of maximum rate of change of gray level is $\tan^{-1}(\Delta_y/\Delta_x)$, and the gradient magnitude of this maximum rate of change is $\sqrt{\Delta_x^2 + \Delta_y^2}$. A pixel lies on an edge if the gradient magnitude at that pixel is sufficiently high. The differences $\Delta_x$ and $\Delta_y$ can be regarded as good approximations to the partial derivatives and the digital image as a good approximation to a smoothly varying brightness function. The gradient magnitude approach has several refinements (e.g. local maximum selection, differences of averages, etc.) to overcome the effect of noise in the image. A detailed description of these can be found in (Rosenfeld, 1988).

- **Laplacian Zero-crossing**

  In this approach, a Laplacian of the image gray level i.e. the sum of the second differences in the $x$ and $y$ directions in the neighborhood of a given pixel is computed. This sum is positive on one side of an edge and negative on the other, hence its zero-crossings define the location of the edges.

- **Hough transforms**

  The Hough transform attempts to detect features such as lines, circles, or curves, that have equations of a particular type by working in a suitable parameter space. For example, to detect arbitrary straight lines, a local curve
detection process is applied to the image to get the edge pixels. A straight line is characterized by a slope $\theta$ and distance $r$ from the origin in the $(r, \theta)$ parametric space (Duda & Hart, 1972). If $P$ is an edge pixel and if it lies on a straight line we can compute $(r, \theta)$ for $P$ and mark the position $(r, \theta)$ in a discrete $(r, \theta)$ array. When this process is done for all edge pixels and if the image contains many collinear $P$'s then there will be a pixel in the $(r, \theta)$ array that has a high count of marks.

### 2.2.3 Segmentation

Segmentation techniques are used for identifying pixels that form homogeneous regions in the image. Feature detection is a special form of segmentation since it identifies special types of pixels which have specific local properties. The common techniques used in segmentation (Rosenfeld, 1988) are described below.

1. Gray level thresholding

   Regions in this segmentation are assumed to have approximately constant gray level across the pixels constituting them. A plot of frequency of each gray level (called the image histogram) in the image gives various peaks (surrounded by valleys) which represent ideal gray levels of the corresponding regions. The image can be segmented into regions by dividing the gray scale into intervals each containing a single peak. This method of segmentation is known as (multi)-thresholding; the points separating the intervals on the gray scale are called thresholds. Thresholding produces good results only if the peaks are well separated. Various refinements to this basic technique can be applied when the peaks overlap or are widely separated.

2. Relaxation techniques

   These are iterative techniques which are used for getting a stable solution from an initial approximation. In the context of segmentation, each pixel is initially classified independently (with certain probabilities). These pixels are then reclassified iteratively to make the classification more consistent. The consistency criteria in segmentation of the image into regions means that if majority of the neighbors of a
pixel \( P \) belong to a given class, so should \( P \). If the goal is to detect edges or curves, the consistency criteria means that if \( P \) lies on an edge or a curve having a given slope at \( P \), so should its neighbors in that direction having a similar slope.

3. Global Homogeneity

In this approach, entire region or curve is required to be a good fit (e.g., in the least squares sense) to some standard function. For example, an edge or curve may be required to be a good fit to a straight line or to a polynomial of higher degree. A split and merge approach can then be used for segmenting an image or a curve into globally homogeneous parts. In this approach, an entire image or a curve is split (e.g., into quadrants or arcs) if the measure of the fit is not good enough. The splitting process is repeated for each part until the entire image or curve is partitioned into parts each of which has a good fit and no two adjacent parts can be merged to yield a good fit.

4. Region Growing; Edge or Curve Tracking

In region growing, a region is built by starting with a set of one or more ‘similar’ pixels (e.g. ‘similar’ by pixel difference) and gradually extending this set by repeatedly adding new pixels or connected sets which resemble pixels already in the set. The resemblance is usually governed by some homogeneity criterion (based on either gray tone or texture) that must be satisfied by the new pixels for inclusion in the region. The procedure for edges or curves is analogous. One starts with strong edge/curve pixels and extends them by adding neighboring edge pixels that continue the edge smoothly or preserve the good global fit. The main disadvantage of this approach is that the results of segmentation are order-dependent. They depend on choice of the starting point and the order in which the pixels are examined for possible incorporation into the region, edge or curve.

5. Hierarchical Techniques

Here one applies a local feature detection technique to a reduced-resolution image to detect ‘coarse features’ of various sizes (edges between large regions, thick curves, large spots, etc.). The finer image features can then be located by examining successively higher-resolution versions of the image in the vicinity of the detected features.
This process requires only a succession of local searches and thereby reduces the cost of global search.

### 2.2.4 Resegmentation

Resegmentation methods are used for forming meaningful entities or parts by segmenting or grouping regions, edges or curves using certain geometric criteria. Examples of such entities are (Rosenfeld, 1988):

1. Connected components and holes: Segmentation of an image often results in many disconnected fragments. Resegmentation methods applied to such fragments result in maximal connected sets of pixels called connected components. Holes are regions surrounded by pixels of a connected component.

2. Borders, Arcs and Curves: Edges obtained in segmentation may be grouped together to form borders of objects or to form arcs and curves in the image. An arc may be further segmented into smoothly curved subarcs which may meet at corners.

3. Thining, Shrinking and Expanding: These techniques are used for forming a skeleton of given objects or to dilate a given object in the image.

### 2.2.5 Properties and Relations

After the resegmentation process, many useful properties of the image parts can be measured by applying various techniques. Examples of such properties are: number of connected components or holes, area (number of pixels in the image part), perimeter, compactness \((\text{area}/\text{perimeter}^2)\) and elongatedness \((\text{area}/\text{thickness}^2)\). Many types of relations between image parts are important for object recognition especially when these are between parts of objects. Most of these relations are defined in terms of relative property values such as lightness/darkness, size, positional reference (e.g. near, far, above below, etc.), and orientation (parallel, etc.) (Rosenfeld, 1988).
2.2.6 Object Recognition

Object recognition may be achieved in several ways. In the graph-based approach, the objects are assumed to consist of parts having certain properties and relationships. They are represented as labeled graphs with nodes representing parts, labeled with property values, and the arcs representing relations, labeled with relation values. Two such graphs are created, one for the expected class of objects (called object graph) and the other for the actual observed object classes in the image (scene graph). Recognition is then achieved by finding subgraphs of the scene graph that are close matches to the object graph. The main limitation in this approach is that the observed image parts may not correspond to the expected object parts. This may be due to segmentation errors, where a single node may split into several nodes or several nodes may merge into a single node. Also, it is sometimes difficult to characterize objects as labeled graphs.

In another approach, although applicable only in some special cases, the objects are characterized by a set of ideal (global) property values or constraints on these values. Recognition then consists of matching an observed list with the ideal list. In certain cases an entire object is treated as a template and matched for optimal fit in the image. The graph-based approach, however, appears to be more general and is applicable in the majority of the cases (Rosenfeld, 1988).

2.3 Computational Characteristics

Investigation of parallel processing solutions to vision applications necessitates understanding the nature of the computations involved. A typical vision application involves several stages of processing with a varying mix of symbolic and numeric processing. Vision applications are conveniently classified into three levels (Weems et al., 1989): low level, intermediate level, and high level as shown in Figure 2.2. The low level processing involves well-structured local computations on the image data while the other levels involve symbolic computations with irregular communication patterns.
2.3.1 Low level processing

Low level processing involves image processing techniques such as image enhancement and restoration, and computer vision techniques of feature extraction and edge detection. Low level processing consists of pixel-to-pixel transformations, where uniform computations are applied at each pixel or at a neighborhood around each pixel in the image. The computations are numeric, regular and well suited to spatial parallelism. The communication pattern is local and processing across the image is identical. Although the computations required at low level are quite straightforward, the sheer volume of data to be processed demands enormous computing power.

2.3.2 Intermediate level processing

At the intermediate level, the basic unit of information is a description of low level image features such as edges, curves, and intensity regions. The algorithms in this category consist of both symbolic and numeric computations. The symbolic computations involve grouping of the low level features into meaningful entities such as sets of parallel lines, rectangular borders of an object, or planes. The algorithms at this level attempt to output descriptions of possible objects in the image data. The grouping operations (e.g. merging and splitting of regions, or linking and reorganizing of lines) involve a large amount of
non-local communications. The fragments of lines require matching and merging across large fraction of the image. Similarly, regions need to be merged and compared with others from possibly non-contiguous areas during the segmentation process. The communication pattern is thus data dependent and irregular.

2.3.3 High level processing

High level applications generate and test hypotheses for object recognition based on data provided by the low and intermediate levels of processing. The applications at this level attempt to recognize objects in the image using either graph-based or rule-based techniques on the object descriptions generated at the intermediate level. Processing at this level is very irregular and may involve dynamic scheduling of the computations.

The volume of data analyzed as the processing progresses from low levels to high levels is substantially reduced. However, the information content of the data is much higher. For example, where pixel values in low level represent brightness values in the image data, relevant data in high level may represent relative size or shapes of the objects. The data types shift from primarily numeric to primarily symbolic (Yalamanchilli & Aggarwal, 1994). Hence, the computations involving these data structures are complex (e.g. object recognition, automatic vehicle guidance). The source of computational burden shifts from large volumes of data to complex numerical and inferencing operations as the processing progresses from low to high level.

Low level algorithms are usually highly structured, repetitive and composed of fixed sets of operations with relatively few data-dependent branches. It is therefore possible to obtain relatively accurate estimates of the operation counts. But high level algorithms are highly data-dependent and processing requirements can vary widely based on the application domain. For example, it is very difficult to estimate the numbers of feature or objects to be processed, and even more difficult to estimate the amount of computations involved. It is therefore very difficult to establish the processing requirements and source of parallelism (e.g. data/functional parallelism) in high level vision algorithms. Hence, the nature of the algorithmic characteristics change as processing evolves from low level
to high levels. These characteristics have influenced the design of several different parallel architectures discussed in the next section.

2.4 Parallel systems for vision

Many applications in computer vision have enormous data throughput and processing requirements which have far exceeded the capabilities of existing uniprocessor architectures. Parallel processing has been perceived as a necessary solution that has led to the conception, design, and subsequent analysis of a number of parallel systems for computer vision which are described below (Weems et al., 1989), (Choudhary & Patel, 1990). The literature on parallel systems for computer vision is vast, however, most of the material can be found in (Duff & Levialdi, 1982), (Kendall & Uhr, 1982), (Uhr, 1987), (Page, 1988), (Prasanna Kumar, 1991), (Narayan et al., 1992), (Siegel et al., 1992).

2.4.1 Mesh connected systems

Mesh connected machines consist of a large number of simple processing elements arranged in a two-dimensional array, with each processing element connected to its four, six, or eight neighbors (Figure 2.3). The processing elements execute instructions broadcast by a central controller in a SIMD mode. The organization of these machines matches the structure of the image data which makes them suitable for low level image processing operations involving computations on individual pixels or small neighborhoods of pixels. They are, however, not suitable for intermediate and high level processing due to simplicity and SIMD nature of the processing elements. Also, communication of information across long distances in the communication network is very time consuming. Some examples of mesh connected machines are (Choudhary & Patel, 1990), (Yalamanchilli & Aggarwal, 1994) the Massively Parallel Processor, the Binary Array Processor, the Distributed Array Processor (DAP) and the Cellular Logic Image Processor (CLIP) series of machines at University College in London, the state of the art in the series being CLIP7 processor array.
2.4.2 Pyramids

Pyramid machines consist of a large number of simple processing elements arranged in layers of mesh-connected arrays. With exception of the array at the lowest layer, each array in the pyramid is one fourth as large as the array below it and each processing element is connected to four processors in the array below it (Figure 2.3). Pyramid machines attempt to minimize the communication delays over large distances present in the mesh connected systems. However, due to SIMD nature of the processing elements these machines can be used in improving speed of mostly low level algorithms, especially those which depend upon communication between pixels that are spatially distant in an image. Some examples of Pyramid machines are (Choudhary & Patel, 1990) Non-Von, the Ultracomputer, PAPIA, and MPP Pyramid.

Figure 2.3: A 4-connected mesh, pyramid and a 3-dimensional hypercube of processing elements

2.4.3 Hypercubes

Hypercube machines consist of \(2^n\) processors connected by a communication network that resembles an \(n\)-dimensional cube. Each processor is connected to \(n\) other processors and can communicate with any other processor using at most \(n\) communication links (Figure 2.3). Hypercube machines can be built to operate in both SIMD and MIMD mode.
These machines provide efficient communication between all the processors because the network has small diameter. They can be used for most low level algorithms and some intermediate and high level applications. However, the algorithms need to be tuned to the underlying topology. Also, larger hypercubes are costly to build since they require many links to be added to each processor. An example of a SIMD hypercube is Connection machine CM-2 while Intel Hypercube, NCube and Cosmic Cube are examples of some MIMD hypercube machines (Choudhary & Patel, 1990).

### 2.4.4 Shared memory machines

Shared memory systems are usually MIMD machines consisting of several general purpose processors which have access to a large global memory through an interconnection network. In some cases the processors may also have a small amount of local memory. The interconnection network may be bus-based or may involve use of a multistage switching network. The former involves a high-speed bus that connects the processors and the memory while the latter provides links between processors and memory on a demand basis (Figure 2.4). The bus-based machines have limited scalability since the common bus used in communication limits the number of processors that can be added in the system. Scalability is much better in the multistage switching network machines but the interconnection networks are complex to build.

Shared memory machines are suitable for high level vision applications due to ease of programming and uniform view of the system. The control of information and synchronization is much easier compared to that in the distributed memory machines. However, due to slow access to global memory and time penalty in process synchronization, such systems are efficient only for coarse-grained parallelism. Examples of shared memory machines (Choudhary & Patel, 1990) that use bus architecture are Sequent Balance and Encore Multimax and those which use multistage networks are BBN Butterfly, IBM RP3 and Cedar.
2.4.5 Pipelined Systems and Systolic arrays

The machines in this category consist of a pipeline of processing elements where data is fed from one end of the pipeline. This data then passes through the processing elements in a serial fashion, and the results are obtained at the other end of the pipeline (Figure 2.4). These systems are used for performing a sequence of operations on a stream of input data. Such systems are useful in morphological operations where long sequences of local operations are performed on a given image data. The examples of machines in this category are cyctocomputers and the systolic arrays (e.g. SLAP or Scan Line Array Processor) (Yalamanchilli & Aggarwal, 1994).

Several solutions have been developed for low level image processing algorithms using systolic arrays (Uhr et al., 1986). These solutions, called systolic solutions, are realized by organizing the flow of data streams through such arrays. Systolic solutions have been obtained for a variety of problems such as edge detection, connected component labeling, and fast Fourier transforms. However, the difficult problem in using these machines is to determine whether a systolic solution exists for a certain problem and, if so, to derive this solution. A representative of the state of the effort is the CMU Warp project. The CMU Warp, a linear systolic array of 10 Warp cells or processing elements, was designed to provide high-speed operations for a number of low level image processing applications. But its flexibility makes it possible to program a variety of other applications as well. The array can operate as a purely systolic array or as a set of processors on a bus in the SIMD or MIMD mode (Yalamanchilli & Aggarwal, 1994).

2.4.6 Partitionable Systems

Due to varied nature of vision applications there were many efforts to design and develop architectures that supported both SIMD and MIMD types of processing. Such hybrid systems addressed the issue of flexibility, partitionability and reconfigurability needed in low, intermediate and high level vision applications. Some examples of such systems include PM4, PASM, REPLICA, Disputer, WISARD, VisTA, the Image Understanding Architecture (IUA) and NETRA. A brief description of all these systems can be found in
Chapter 2. Parallelism in Computer Vision

Figure 2.4: Shared memory machines (interconnected by a bus and switching network) and systolic/pipeline systems

(Yalamanchilli & Aggarwal, 1994), (Choudhary & Patel, 1990), and (Prasanna Kumar, 1991). The common characteristics of these machines is that they consist of large number of processing units which can be partitioned into groups that can operate in SIMD and MIMD mode. The architecture of IUA (Weems et al., 1989), for example, has three different layers of processing units suitable for low, intermediate and high level vision algorithms. However, such systems involve considerable design and development costs due to their specialized and complex architecture.

2.4.7 General purpose parallel systems

General purpose parallel systems are the current high-performance parallel machines such as IBM SP-2, Meiko CS-2, Intel Paragon, Cray T3D, and PARAM 10000\(^1\). Since they are based on workstation microprocessor technology, these systems are versatile and cost-effective compared to the specialized vision systems described earlier. These systems mainly consist of processing units, each with a local memory, and a high speed interconnection network. They are mostly tightly-coupled, i.e. the interconnects are system-specific with point-to-point links between the processors. Their major disadvantage is that it is difficult for a parallel application to use the resources efficiently. Also, the system-specific interconnects do not provide a flexibility of adding existing machines as hosts. They

\(^1\)Developed by C-DAC (Center for Development of Advanced Computing), Pune, India
cannot incorporate heterogeneous architectures, hence applications cannot select most suitable computing resources for each computation. Therefore, although tightly-coupled systems always support faster communication, their advantage is likely to shrink over time (Steenkiste, 1996).

2.5 Computing on workstation clusters

During the past several years, network-based computing environments, such as a cluster of workstations, have proved to be an attractive alternative for high-performance computing over the conventional parallel machines. This is due to rapid advances in microprocessor technology and emergence of high-speed networks having a network bandwidth of the order of gigabit per second (Boden et al., 1995), (Steenkiste, 1996). A cluster of workstations offers several advantages for implementing high-performance computing solutions. It provides multiple CPUs, large memory, stable software, and heterogeneous computing environments for developing high-performance computing solutions to many computation-intensive problems. It is believed that the future computing environments will slowly migrate towards the concept that 'the network is the computer' (Turcotte, 1996).

2.5.1 Cluster Configuration

A workstation cluster is basically a collection of workstations connected by a commodity network, such as Ethernet or ATM. The three common network topologies employed with workstation clusters are shown in Figure 2.5. The Ethernet or bus is the most commonly implemented network for clusters. Switch based interconnects are typically configured in a star arrangement, and are used exclusively with dedicated clusters. There are also hierarchical designs in which multiple types of interconnects are utilized.

The workstations in a cluster communicate with each other by exchanging messages or data packets transmitted using either transmission control protocol (TCP) or user datagram protocol (UDP). The former processes streams of data such that the reliability of message delivery is assured. The latter sends data packets that are attempted to be
delivered (i.e. reliability of message delivery is not assured) (Turcotte, 1996). Two software methods are used for communicating the messages: message passing and distributed shared memory. Message passing involves explicit transmitting of messages between the systems. Distributed shared memory (DSM), which is usually implemented using message passing, involves accessing of data without the concern for physical location.

Workstation clusters have one obvious limitation due to the use of relatively slow network interconnection. The interconnects have a low bandwidth and a high latency, where, bandwidth refers to the speed at which message data is transmitted and latency is the time spent in initiating the transmission of a message. Ethernet, the most commonly implemented network for clusters, transmits information at 10Mb/s and has a message latency of 3$\mu$s. There have been several efforts to design expensive high-speed interconnects to overcome the limitations induced by the speed of Ethernet. Typical examples include, FDDI (100 Mb/s), HiPPI (800 Mb/s), VME Bit3 (20 Mb/s) and ATM OC-12 (622 Mb/s) (Turcotte, 1996).

The need to maximize the network performance (high bandwidth and low latency), particularly for parallel applications, has yielded unique solutions. A recent example of one such network is Myrinet (Boden et al., 1995). It consists of a collection of workstations, the network comprising links and switches to route the data, and the network interface between the workstations and the network links. The network interface consisting of a special processor can transfer blocks of data to allow for the overlap of computation and communication. One way message latencies of 100 $\mu$s and bandwidths of 255 Mb/s have been observed in a Myrinet-based system interconnecting several Sparc workstations. (Boden et al., 1995).
Chapter 2. Parallelism in Computer Vision

Workstation clusters are simple to configure. However, it is important to identify categories of applications which can be implemented on these systems more effectively. The applications which require computational capabilities of high-performance computing systems can be categorized as follows (Turcotte, 1996):

- **Capability demand** which includes megaproblems that require all the computational capabilities of any available system including memory and CPU. Grand Challenge applications which require massive parallel processing fall into this category.

- **Capacity demand** which includes applications requiring substantial, but far from ultimate, performance and making moderate demands on memory. These jobs are ideal candidates for workstation clusters.

Workstation clusters provide practical and cost-effective computing solution for the capacity demand problems. They provide complementary rather than practical replacements for the general-purpose parallel computing machines.

2.5.2 Advantages of workstation clusters

Workstation clusters offer several advantages over the traditional parallel computing environments (Turcotte, 1996) as described below:

- **Workstation clusters** provide simple, inexpensive and readily accessible computing platform to design, develop and implement parallel solutions to a wide range of applications. They offer excellent price/performance benefits in comparison with the traditional parallel computing solutions.

- **Workstations** provide large, cost-effective memory which is not available in most traditional parallel computers, and as problems continue to grow in complexity and detail, availability of a large memory is as important as the processor speed.

- **Workstation clusters** offer stable software environments compared to dedicated parallel machines. Software environments such as operating systems, compilers, libraries,
and software tools, are yet to develop to a point of general acceptability for dedicated parallel machines.

- Clusters provide a cost-effective environment to study topics related to heterogeneous computing. It is generally believed that future, high-performance computing systems will achieve maximum performance capabilities only by exploiting the benefits of heterogeneous computing environments.

- Clusters have a graceful degradability. The entire cluster is not lost due to failure of a single system in the cluster. Also, since the clusters are created using commodity components, maintenance costs are usually much less than for an equivalent investment in a dedicated parallel computer.

### 2.5.3 Use of clusters

Clusters can be used as enterprise clusters or dedicated clusters (Turcotte, 1996). Enterprise clusters are configured with workstations that are owned by different individuals or groups. The machines in this cluster are normally heterogeneous (multivendor), and are almost exclusively connected via Ethernet. This type of clustering relies on individual owners contributing their unused computing cycles to a shared pool. The individual owners expect to receive more resources than they contribute. Enterprise clusters are controlled and managed by a management software. This software enables effective use of collective idle time available on most workstations. This idle time can be used to process jobs of several different users in the group. The management software ensures that the systems of individual owners are not saturated when they try to use their own systems. The individual owners can specify how their system will participate in the resource pool.

Several papers have proposed different schemes for sharing resources in enterprise clusters where the main idea is to identify idle machines in the network and schedule background jobs on them with minimum disruption to the individual owners of the machines. When the owner resumes activity at a workstation, the job is either suspended, terminated, or moved to another machine in the cluster. These efforts have resulted in either speeding individual jobs or programs by locating idle resources (Alonso & Cova,
Dedicated clusters are installed as substitutes or replacements to the traditional parallel computing systems. They consist of individual workstations managed by a single group which administers the clusters like a central mainframe. They are usually interconnected by high-speed networks such as FDDI, SOCC, and HiPPI (Turcotte, 1996). Dedicated clusters usually have a control workstation which manages the job queue and acts as an interface to the remaining clusters. The control system can be used to dynamically partition the clusters to execute interactive jobs (e.g. code development, graphics, etc.), serial batch jobs and jobs that have been parallelized.

2.5.4 Parallel computing using Clusters

Workstation clusters, both enterprise and dedicated, can be used as parallel computing environments for implementing parallel solutions to a wide range of applications. There have been several papers which have addressed the issues involved in solving a single problem on a collection of workstations. Silverman and Stuart (Silverman & Stuart, 1989) have used the cluster as a loosely coupled message passing parallel computer to solve some asynchronous algorithms in numerical analysis. Magee and Cheung (Magee & Cheung, 1991) have proposed a supervisor-worker programming model to distribute computations over a set of workstations.

Atallah et al. (Atallah et al., 1992) have proposed a resource management technique called coscheduling or gang scheduling. It involves dividing a large task into subtasks which are then scheduled to execute concurrently on a set of workstations. The subtasks need to coordinate their execution by starting at the same time and computing at the same pace. Wang and Blum (Wang & Blum, 1996) have developed a small message-passing library to implement iterative numerical algorithms which require synchronization at the end of each iteration (synchronous algorithms). Finally, there have been attempts to demonstrate the capability of workstation clusters to solve some grand challenge problems (Beguelin et al.,
Two commonly used approaches to parallelize applications using clusters are:

- Extension of existing sequential languages (e.g. C++, FORTRAN) to handle necessary communications and synchronization (see (Wilson & Lu, 1996) for several concurrent C++ extensions).
- Defining new programming languages or an environment based on object-oriented, functional or logical paradigms.

There are several software systems such as Express, Linda, p4, PVM, and MPI, which are used for creating parallel applications on workstation clusters. A comprehensive review of these systems is contained in (Turcotte, 1993). This section briefly describes characteristics of a Parallel Virtual Machine system which is used as a programming environment in this thesis.

Parallel Virtual Machine (PVM) (Beguelin et al., 1992) was developed at Oak Ridge National Laboratory, Tennessee, and is the most popular system for developing parallel applications on workstation clusters (Turcotte, 1993). PVM is a software library which allows utilization of a heterogeneous network of parallel and serial computers as a single computing resource. It is based on the message passing model (coordination model discussed in Section 2.1.1). An application in PVM consists of multiple components, each of which implements a particular functional process. There are four categories of components in PVM: process management, interprocess communication, synchronization and service (status checking, buffer manipulation, etc). The PVM model is based on asynchronous processes which are typically executed as individual programs (e.g. heavyweight Unix Processes) on each system in the cluster. The communication between the processes occurs via explicit message passing.
Chapter 2. Parallelism in Computer Vision

2.6 Parallelization using Design Patterns

Most of the parallel programs are usually coded in terms of high level constructs where the functions for communication, synchronization, and sometimes even computation are rolled into a single routine. This style of parallel code development increases program complexity and reduces program reliability and code reusability. Writing explicit parallel code for parallelizing various applications on a cluster of workstations has some additional problems too. The available machines and their capabilities can vary from one execution to another, which, sometimes can lead to a significant reduction in parallel performance. Also, about 69% of parallel programmers (Pancake, 1996) modify or use existing blocks of code to compose new programs. Since most of the parallel programs, especially those in vision, utilize a rather small set of recurring algorithmic structures, it is meaningful to identify and formulate these algorithmic structures as design patterns. Such decoupling would reduce program complexity and increase code reusability in different situations and in future software development.

2.6.1 Design patterns

The concept of a design pattern was introduced by architect Christopher Alexander who described the recurring themes in architecture as the design patterns (Alexander, 1979). A pattern represents a replicated similarity in a design, and in particular a similarity that can be customized and tuned to human needs and comforts. Thus, an arch on every door and window of a room is a pattern, yet it does not specifically imply the size of the archs, their height from the floor nor their framing. The idea introduced by Christopher Alexander has inspired software designers over the past decade to discover (and rediscover) software architectural patterns in the software people develop. In software, design patterns are software abstractions that occur repeatedly while developing software solutions for problems in a particular domain such as business data processing, telecommunications, distributed communication software, and parallel vision processing (Gamma et al., 1994).

Design patterns capture the static and dynamic structures of the solutions that occur repeatedly when developing applications in a particular domain (Coplien & Schmidt,
They articulate proven design techniques for developing software solutions in a particular context. Capturing and articulating key design patterns helps to enhance software quality by addressing basic challenges in software development. These challenges include communication of designs among the developers; accommodating new design paradigms or styles; resolving reusability and portability issues; and avoiding development traps and pitfalls that are usually learned only by costly trial and error (Coplien & Schmidt, 1995).

Design patterns serve as a good communication medium. When several software developers are discussing various potential solutions to a problem, they can use the pattern names as a precise and concise way to communicate complex concepts effectively. Design patterns are extracted from working designs. They capture the essential parts of a design in a compact form, including specifics about the context that makes the patterns applicable or not. This compact representation helps developers and maintainers understand the architecture of a system, which allows more effective software development (Beck et al., 1996).

Patterns promote design reuse where routine solutions with well-understood properties can be reapplied to new problems with confidence (Monroe et al., 1997). Encouraging and recording the reuse of best practices can lead to a significant code reuse. A collection of design patterns would help developers produce good designs faster and would provide alternatives when applied to particular situations.

The design patterns in the parallel vision systems, implemented on network-based machines (such as a cluster of workstations) are the software components which distribute and execute computations of various vision applications on these machines. Developing a parallel implementation for an application in such an environment usually involves a sequence of steps. These steps include a) partitioning the application into different tasks, b) using a suitable parallel programming language or tool, to concurrently implement (map) these tasks on a given number of workstations, and c) managing the low level programming details such as marshalling data, sending and receiving messages, and process (task) synchronization. The partitioning, mapping and communication structure of the parallelization process of an application is a parallel programming paradigm that can be used to parallelize any other application with a similar computational structure. The design patterns essentially capture these parallel programming paradigms and relieve the
user from tedious parallelization details.

The main advantages in using design patterns for parallelizing vision applications on a cluster of workstations are:

1. The design patterns can be developed to utilize a readily available pool of workstations which, for some applications, can approach or exceed performance over non-available fastest machines.

2. A design pattern decouples the details of parallel implementations from the user.

3. A design pattern can be reutilized to parallelize any application with a similar computational structure as implemented by that pattern.

2.6.2 Forms of Parallelism in Vision

Many vision applications can be parallelized by using various forms of parallelism. Each form of parallelism is a simple organizational technique that can be used for designing and developing parallel algorithms for a certain class of problems. Identifying various forms of parallelism in vision applications would help in capturing and articulating key design patterns in parallel vision systems. Many of these forms are variants of the class of algorithms described in section 2.1.2.

Data Partitioning

In this form of parallelism, the image array is partitioned into adjacent regions or subimages and each subimage is processed in parallel by a different processor. Such type of parallelism is suitable for low level processing operations, such as image filtering and convolution. The regions may overlap at the boundaries of the subdivisions to enable processing of the pixels at the region boundaries.
Chapter 2. Parallelism in Computer Vision

Synchronous Iteration

In case of Synchronous Iteration, each processor performs the same iterative computation on a different region of image data. The processors, however, must be synchronized at the end of each iteration and hence no processor can start the next iteration until all the processors have finished the previous iteration. The need for synchronization is due to the fact that data produced by a given processor during $i$-th iteration is used by other processors during $(i+1)$th iteration. This form of parallelism is suitable for iterative smoothing and sharpening operations on the image data.

Algorithmic Parallelism

In algorithmic parallelism, the algorithm is partitioned into several independent parts and each part is processed by a separate processor concurrently. Each processor works independently and requires no explicit synchronization or communication with the other processors. For example, the two convolutions in Sobel edge detection can be executed concurrently on separate processors (Downton et al., 1996).

Temporal Multiplexing

In this form of parallelism, instead of splitting individual image data sets, complete image data sets are processed in parallel by different processors. This form of parallelism is also identified as processor farming (Downton et al., 1996). However, temporal multiplexing form of parallelism is sometimes also associated with operator parallelism in low level image processing. Low level operators, such as erosion and dilation in image morphology, can be cascaded into several stages (Pitas, 1993). Each stage, implemented on separate processor, processes a complete image data set. The output of any stage is input of the subsequent stage. For example, if $F$ is a operator operating on image $I$, then $F$ can be cascaded into several stages as follows:

$$ O = F(I) = F_n(F_{n-1}(... (F_2(F_1(I))...)...)) \quad (2.2) $$
But the cascaded implementation of an operator/algorithm also represents the pipeline form of parallelism (described later). In this thesis we do not associate this form of parallelism with temporal multiplexing. We identify temporal multiplexing with a type of processing that involves implementation of an algorithm/operator as a single program unit, operating on complete image data sets.

**Workpool**

In the workpool mode of parallelism, a central pool of similar computational tasks is maintained. A large number of workers repeatedly retrieve tasks from the pool, perform required computations, and possibly add new tasks to the pool. The computation terminates when the task pool is empty. This technique is used for implementing solutions to combinatorial problems in high level vision such as tree or graph searches. A large number of tasks are generated dynamically which can be picked up by any worker process.

**Pipeline**

In pipelining, the application algorithm is sequentially subdivided into various components arranged in a pipeline. Each component is processed by different processor and performs a certain phase of the overall computation. The data flows through entire pipeline structure through the neighboring component processors.

**Pipeline Processor Farm**

Pipeline Processor Farm (Downton et al., 1996) is a generalized form of pipeline parallelism where each component in the pipeline may be parallelized by various parallel programming techniques described earlier.
2.6.3 Design patterns for parallel vision

Based on various forms of parallelism discussed in section 2.6.2, we present following design patterns to parallelize vision applications on a cluster of workstations. A detailed description of each pattern is presented in chapter 3.

- Farmer-Worker pattern: This pattern consists of a farmer process (or component) which is continuously polled for a computational work by a set of independent worker components. It is mainly used for implementing data parallelism, where the image data is divided into different subimages which are processed independently by different workers. There is no communication between the worker components.

- Master-Worker pattern: This pattern consists of a master component which distributes the work to various worker components. Each worker component communicates with neighboring worker components to exchange the intermediate results. This pattern is used for parallelizing synchronous data parallel algorithms.

- Controller-Worker pattern: This pattern is similar to the Master-Worker pattern described above, except that each worker may communicate with every other worker in the pattern. It is used for parallelizing a class of problems in which each object or subtask of the problem needs to interact with every other object or a subtask.

- Divide-and-Conquer pattern: This pattern is used for structuring applications in which either the data or the application algorithm is divided into several subtasks. Each subtask may be executed on single processor or may be further divided (recursively) into smaller subtasks.

- Temporal Multiplexing pattern: This pattern is used for processing several data sets or a sequence of image frames on multiple processors. Each processor processes a complete data set and executes the same program code.

- Pipeline pattern: This pattern consists of a pipeline of components executed concurrently in a specified order. It is used in situations where a vision application can be divided into components which are by themselves independent, and interact with
Chapter 2. Parallelism in Computer Vision

each other only by using output data stream of one component as an input data stream to another.

- Composite Pipeline pattern: Structurally, this pattern is similar to the pipeline pattern. The only difference is that each component of the pipeline can itself be parallelized using any of the design patterns stated above.

2.7 Related work

In this section, we outline some of the leading research efforts which have been inspirational to the work presented in this thesis. Although the concept of design patterns is new, the idea of identifying and capturing common forms as software abstractions in parallel software systems is a decade old.

Zimran et al. (Zimran et al., 1990) have proposed a set of implementation machines used for parallel implementation of various applications on a shared and distributed memory parallel machines. A layer of implementation machines (IM) is introduced between the application and the physical machine. The implementation machines consist of common parallel programming paradigms such as master/slave, pipeline, and pyramids. Each implementation machine is associated with a mathematical representation that can predict the performance bounds for distributed computations. An application is developed in terms of one or more implementation machines which are then implemented efficiently on the underlying hardware. The IMs are made available in the form of modifiable templates which implement the relevant communication and synchronization functions. However, the set of implementation machines presented, do not address issues related to domain-specific problems. They represent only the general forms of parallel programming paradigms.

Magee and Cheung (Magee & Cheung, 1991) have described the supervisor-worker paradigm to distribute the computations of an application on a network of workstations. They have discussed the robustness and load balancing properties of this paradigm and have applied simple formulae to predict the performance of an algorithm, implemented using this paradigm. The supervisor-worker paradigm consists of a supervisor process
that distributes the computational work to a number of worker processes, each working independently of the other. However, only embarrassingly parallel class of applications can be parallelized using this paradigm.

Singh et al. (Singh et al., 1991) developed a system called FrameWorks which uses templates to generate distributed applications on a network of workstations. Programs are written as sequential procedures enclosed in templates. The templates hide the low level parallelization details, such as communication and synchronization. A user selects appropriate templates (e.g. pipeline, contractor, input/output) to describe the behavior of a parallel program. The system then generates the code for implementing the communication and synchronization between the processes. The concepts of the FrameWorks system were later used to create another such system called Enterprise.

The Enterprise system, like FrameWorks, has a graphical interface by which the users can create parallel applications using assets such as pipeline, master/slave, divide-and-conquer (Schaeffer et al., 1993). This system automatically inserts necessary code for communication and synchronization relieving the users from low level programming details which include marshalling data, sending/receiving messages and synchronization. However, both FrameWorks and Enterprise systems do not support data parallelism and complex synchronization, communication, and scheduling structures. Most of the parallelism that can be achieved in an application is performed by pipelining and temporal multiplexing. In these forms of parallelism the processors operate only on complete images.

Darlington et al. (Darlington et al., 1993) have proposed a set of higher-order parallel forms called skeletons as the basic building blocks of a parallel program. They have also provided program transformations which convert between skeletons, giving portability across several different machines. A skeleton captures an algorithmic form common to a range of programming applications. Each skeleton is associated with a set of architectures on which efficient realizations of the skeleton are known to exist. The skeletons are also associated with performance models which can be used to predict the performance of a parallel program implemented using these skeletons. A set of transformations is used for transforming one skeleton to another in order to suit the architectural requirements of different machines. However, the skeletons represent a general class of parallel programming.
paradigms. They are not domain-specific and therefore need to be tuned and extended in order to reflect the characteristics and control structures associated with the problems in a given domain.

Downton et al. (Downton et al., 1996) have proposed a design methodology based on pipeline of processor farm (PPF) for parallelizing vision applications on MIMD machines. Their design method enables parallelization of complete vision systems (with continuous input/output) in a top-down fashion, where parallel implementations of individual algorithms are treated as components in the design model. However, this design methodology is implicit, i.e. it does not present detailed description of the methods or designs used in parallelization of individual algorithms. For example, their paper identifies 'data parallelism' as one of several methods for parallelizing vision algorithms. But 'data parallelism' can be applied to both synchronous and embarrassingly parallel algorithms. Our work in this thesis aims to make the design information in designs/methods for parallel vision systems, explicit. We abstract and document the design information in their design methodology in the form of Composite-Pipeline pattern in this thesis.

2.8 Summary

In this chapter we have reviewed concepts and methods in several different areas related to the parallel vision systems. We began with a brief introduction to parallel computing systems and their classification as SISD, SIMD and MIMD machines, based on the instruction streams and data streams. This was followed by a discussion on parallel algorithms and their classification in terms of different algorithmic classes such as synchronous, loosely synchronous, asynchronous, and embarrassingly parallel. These classes are useful when discussing about computations at higher level. We have also given a brief introduction on measuring performance in parallel programs.

We then described general principles and methods used in the field of computer vision. Our primary concern has been vision applications involving analysis of 2D scenes. We presented different techniques and algorithms for feature detection, segmentation, resegmentation and object recognition used in 2D vision. We also described the computational
Chapter 2. Parallelism in Computer Vision

Characteristics of these algorithms and their classification into three levels: low, intermediate and high. Low level algorithms are usually highly structured, repetitive and composed of fixed sets of operations. Higher level algorithms on the other hand, are very irregular and may involve dynamic scheduling of the computations. The distinctive nature of their characteristics has influenced the design and development of several different parallel architectures in computer vision. Several such architectures comprising either SIMD, MIMD or both SIMD and MIMD (partitionable) dedicated parallel machines have been described.

We described parallel computing using workstation clusters and discussed their advantages over the conventional parallel machines. This was followed by an introduction to the concept of design patterns. Design patterns are software abstractions that occur repeatedly while developing software solutions for problems in a particular domain. Various forms of parallelism in vision applications were identified in order to capture and articulate key design patterns in parallel vision systems. Finally, we have outlined some of the leading research efforts that have been inspirational to the work presented in this thesis.
Chapter 3

Design patterns for parallelizing vision applications

Design patterns for parallel vision applications (introduced in section 2.6.3) represent designs or methods used for implementing these applications on various parallel architectures. Some of these patterns, such as Farmer-Worker and Master-Worker, represent common methods which can be used for parallelizing algorithms not only in vision but also in other computing disciplines. But other patterns, such as Temporal Multiplexing and Composite Pipeline, are suitable only for parallelizing applications in vision (for an example, see (Downton et al., 1996)).

There have been several efforts in the past to present different design methods for parallelizing vision algorithms/applications on various parallel architectures (Downton et al., 1996), (Stout, 1987). However, there have been no attempts to abstract and document the design information in these design methods. This chapter attempts to fill this gap by capturing and documenting this design information in the form of design patterns. These design patterns have been formulated to represent common algorithmic structures in various parallel vision algorithms/applications described in (Kendall & Uhr, 1982), (Uhr, 1987), (Stout, 1987), (Page, 1988), (Prasanna Kumar, 1991), (Hussain, 1991), (Pitas, 1993), (Wang et al., 1996), (Downton et al., 1996). A documentation or catalogue of key design patterns for parallel vision applications would give standard names and
Chapter 3. Design patterns for parallelizing vision applications

definitions to the techniques used in parallelization of these applications. By making
design knowledge explicit in the form of design patterns, experienced and novice designers
would be able to reuse the designs in different situations (Coplien & Schmidt, 1995).
Design patterns are useful in turning an analysis model into an implementation model
(Beck et al., 1996).

This chapter describes a system of design patterns used for parallelizing majority
of vision applications on coarse-grained machines, such as a cluster of workstations. A
system of patterns for parallel vision applications consists of many different patterns used
in different situations. In order to facilitate their effective use and to help developers
in selecting and implementing the right patterns for a given situation, it is necessary to
describe the patterns in a uniform way. Such a description must address all the aspects
relevant to a pattern's characterization, detailed description, implementation, selection
and comparison with other patterns. A system of patterns should address issues concern­
ing the construction of patterns into more complex and heterogeneous structures. A
comprehensive and well-defined system of patterns forms a uniquely powerful and flexible
vehicle for expressing software systems (Buschmann & Meunier, 1995).

This chapter is organized as follows. Section 3.1 describes different classification
schemes used in classifying the patterns at various levels of abstraction. Section 3.2
outlines the template used for describing the design patterns. The remaining sections
describe different design patterns used in parallelizing various vision applications on a
cluster of workstations. The patterns in these sections have also been published in (Kadam
et al., 1997) (Kadam et al., 1996).

3.1 Organization of patterns

Design patterns vary in their level of abstraction and are usually organized into different
categories based on some classification scheme. Such a classification scheme is believed to
provide a guide when selecting a pattern for a particular design situation. Gamma et al.
(Gamma et al., 1994) classify design patterns according to their functionality. The design
patterns can either have creationals, structural, or behavioral purpose.
Chapter 3. Design patterns for parallelizing vision applications

Creational patterns concern the process of object creation. The Singleton pattern (Gamma et al., 1994) is a creational pattern used in ensuring that a class or a component of some design pattern has only one instance. Structural patterns deal with the composition of classes or objects. The Proxy pattern (Gamma et al., 1994) is a structural pattern which makes the clients or the users of a component communicate with a representative rather than with the component itself. Behavioral patterns characterize the ways in which classes or objects interact and distribute the responsibility. The Iterator pattern (Gamma et al., 1994) is a behavioral pattern which provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

The classification scheme proposed by Gamma et al. has certain limitations. The classes of functionality in this classification scheme are of general nature rather than being specific for any application domain. Hence, it is difficult to select appropriate patterns for solving or structuring problems in a given application domain. Buschmann and Meunier (Buschmann & Meunier, 1995) therefore proposed a classification scheme which classifies the patterns into different classes based on different levels of abstractions in software systems. They identified three different classes of patterns, namely, architectural frameworks, design patterns and idioms. This classification scheme was later used to formally describe a system of patterns for software architecture (Buschmann et al., 1996).

An architectural framework expresses a fundamental paradigm for structuring software systems. It provides a set of predefined subsystems and includes rules and guidelines for organizing the relationships between them. For example, a Pipeline pattern described in section 3.8 can be considered as an architectural pattern when it is used for structuring a vision application that can be divided into a sequence of independent subsystems, executed in a specified order. Each subsystem interacts with its neighboring subsystems only by exchanging streams of data. An application structured using a Pipeline pattern may be parallelized by executing the application subsystems concurrently. The execution and the interactions of the application subsystems are implemented by the corresponding components of the Pipeline pattern.

An architectural framework consists of several smaller units called design patterns. Design patterns describe the basic scheme for structuring subsystems and components
of a software system, as well as the relationships between them. Design patterns are medium-level patterns, smaller in scale than the architectural patterns. A Master-Worker pattern described in section 3.4 is an example of a design pattern which can be used for distributing computations of an application to identical worker components. Idioms, on the other hand, are low-level patterns which are specific to some programming language. An idiom describes the aspects of both design and implementation of the specific components in a pattern by using the features of a given language. A singleton pattern described earlier is an example of an idiom.

The classification scheme based on different levels of abstractions in software systems (also termed as system granularity) can sometimes be ambiguous. A pattern can be used to structure either a complete software system or just a single component or subsystem. A Pipeline pattern, for example, can be part of a larger system. Its classification as an architectural pattern or design pattern therefore depends on the context. Similarly, the boundary between the design patterns and idioms is imprecise. In fact, Buschmann and Meunier (Buschmann & Meunier, 1995) stated this ambiguity when they proposed their classification scheme. However, this classification scheme provides a reasonable hierarchy for describing most of the patterns in software systems.

We do not follow any strict classification scheme, but rather use it as a general guide to specify the type of patterns we propose and describe in this thesis. Using the classification scheme formally used by Buschmann et al. to classify the patterns in their book (Buschmann et al., 1996), we describe a system of patterns for parallel vision applications at the level of architectural frameworks and design patterns. If not stated otherwise, we use the term design patterns to represent all the patterns at various levels of abstraction. Also, we use the terms pattern and design pattern as synonyms.

3.2 Description of design patterns - a template

We use a template to describe all the design patterns presented in this thesis. The template provides description of how each pattern works, where it should be applied and what are the tradeoffs in its use. This description scheme for the patterns is closely related
to the ones proposed by Gamma et al. (Gamma et al., 1994) and Buschmann et al. (Buschmann et al., 1996). Its intention is to support the understanding, comparison, selection, and implementation of patterns within a given design situation. The template used for describing each pattern is given below:

1. Pattern name
   The name of the pattern which conveys the essence of that pattern.

2. Intent
   A short statement about the main functionality of the pattern and the problems that it addresses.

3. Motivation
   An example illustrating a concrete instance of the pattern. The motivational example relates the pattern to its practical usage.

4. Structure
   The structure of the pattern in terms of objects or components described in both textual and graphic representation. We use a variant of the object model (described in Appendix A) to display the structure of the pattern.

5. Interaction
   The interactions between the components of the pattern and between the outside world are depicted. We adapt the object message sequence chart notation (described in Appendix A) to describe the interactions between the components of a pattern.

6. Implementation
   The general guidelines for implementing the pattern. These are, however, only suggestions which should be suitably modified depending upon the needs of a given problem.

7. Consequences
   The consequences and trade-offs of using a pattern. The parameters that can be varied independently by using the design pattern. We describe the benefits and potential liabilities of a pattern.
8. Applicability

The set of conditions and requirements that indicate when the pattern may be applicable.

9. Known Uses

We provide examples of the use of the pattern in different situations. We also provide some related efforts in using the pattern or its variants.

### 3.3 Farmer-Worker Pattern

**Intent**

The Farmer-Worker pattern, which provides dynamic load balancing, is used for implementing embarrassingly parallel algorithms. The farmer component divides the problem task into a collection of independent subtasks. The worker components grab individual subtasks and perform identical operations on the data, before returning the transformed values to the farmer for collating.

**Motivation**

Averaging is a simple image enhancement technique which is used for removing noise from an image corrupted by random noise. It uses linear local window operations to change the pixel intensities in the corrupted image using the equation

\[
 f'(a,b) = \left( \sum_{(i,j) \in N} f(i,j) \right)/N \tag{3.1}
\]

where, \( f \) is noisy image, \( f' \) is filtered image, and \( N \) is a set of \( N \) neighboring pixel points around a point \((a,b)\) in the image (Sonka et al., 1993). The averaging operation, using the Farmer-Worker pattern, can be parallelized by dividing the image into subimages and averaging these subimages concurrently on different processors.
Chapter 3. Design patterns for parallelizing vision applications

Structure

The Farmer-Worker pattern consists of a farmer component and several independent but identical worker components or processes as shown in Figure 3.1. The client interacts with the farmer component to parallelize a certain application. The farmer component is responsible for partitioning the application into several independent subtasks, starting the worker components to process these subtasks, collecting the partial results from the worker components, and finally returning the collected results to the client. The worker components are responsible for processing the individual subtasks created and assigned by the farmer. The Farmer-Worker pattern consists of one farmer and at least two workers.

Interaction

The interactions between the components of the Farmer-Worker pattern are shown in Figure 3.2.

- The client requests the farmer to parallelize a given application.
- The farmer component divides the application into different subtasks and starts several worker components to process these subtasks.
Chapter 3. Design patterns for parallelizing vision applications

Figure 3.2: Object Interaction in the Farmer-Worker Pattern

- Each worker repeatedly requests a subtask, performs specified computation on the data in the subtask, and returns the results back to the farmer. This continues until a termination condition is encountered.

- The termination condition occurs when there are no more tasks to be processed. The farmer detects this condition and signals the worker components to terminate.

- The farmer collates the results returned by the workers for a given application. The farmer returns the collated result to the client.

Implementation

The Farmer-Worker pattern can be implemented by following the steps described below:

1. Partition the work. Specify how the problem task can be divided into a collection of independent subtasks. For the averaging operation, we could either partition the image into horizontal or vertical blocks of subimages. Each subimage represents a subtask to be
processed. The subimages must also include the required pixel values at the boundaries of the partition.

2. **Combine the results.** Specify how the final results should be collated from the partial results obtained from the worker components. In the averaging example, the farmer component simply collates the averaged subimages onto the output image without any change.

3. **Specify the interaction between the farmer and the workers.** This interaction can be implemented in at least three different ways: a) Each worker receives a subtask from the farmer at the beginning. When a worker returns the partial results to the farmer, the farmer collates these results and sends another subtask to the worker. b) A separate component called *gatherer* is created. While the farmer distributes the subtasks to the workers, the gatherer collects the partial results from each worker. The gatherer then returns the final collected result to the farmer. c) If the operation of collecting the partial results is trivial or easily delayed to the end of computation, the farmer can turn into a worker after setting up the collection of subtasks in a common repository, such as a subtask queue. The workers now fetch the subtasks from the subtask queue. However, this implementation needs a shared counter to manage the subtask queue. In all the cases, when there are no more subtasks to be processed, the farmer sends a termination message to each worker (and gatherer in (b)). In the averaging example, as the farmer simply collects the results returned by the workers, we use the first method to implement the interaction between the farmer and the workers.

4. **Implement the farmer and the worker components** according to the specifications outlined in previous steps.

**Consequences**

The Farmer-Worker pattern provides several benefits:

*Dynamic load balancing:* The Farmer-Worker pattern provides an even distribution of the load when the computational requirements of the individual subtasks and the speed of different processors in the parallel system, vary significantly and unpredictably. The
worker components in a Farmer-Worker pattern grab the subtasks and process them at their own pace. A faster processor or node of the parallel system would grab and process more subtasks than the slower nodes. Hence, the number of subtasks processed by each worker is proportional to the speed of their corresponding nodes or processors. The Farmer-Worker pattern therefore provides dynamic load balancing of the subtasks during its execution.

*Scalability and flexibility:* It is possible to add new workers or change existing algorithms in the workers without major changes to the farmer. The client is not affected by these changes. Similarly, it is possible to change the algorithms for partitioning the work or co-ordinating the workers in the farmer component without affecting the client.

The Farmer-Worker pattern suffers from the following liabilities:

*Feasibility:* The Farmer-Worker pattern may not always be feasible. The activities of partitioning of the work, starting and controlling the workers, delegating the work amongst the workers, and collecting the final results, consume processing time. The pattern would be effective only when the time spent in these activities is significantly lower than the time required to perform the computations in a given application.

*Effectiveness:* The Farmer-Worker pattern is effective only when there are more subtasks than the number of processors. The parallelism in this pattern is expressed in terms of the number of subtasks. When all the subtasks are processed, no further parallelism is available in the application. On the other hand, too many subtasks with relatively lower compute to communication ratio, may lead to poor performance. A proper balance between the granularity and the number of subtasks created is therefore critical for the effectiveness of this pattern.

**Applicability**

The Farmer-Worker pattern represents a parallel programming paradigm for implementing embarrassingly parallel algorithms. It can be used to parallelize any vision application in which
Chapter 3. Design patterns for parallelizing vision applications

- the data can be partitioned into several independent data sets
- each data set can be processed concurrently by different workers
- the processing of each data set does not require interaction between the worker components to exchange the intermediate results

Known Uses

The Farmer-Worker pattern has applications in various levels of vision processing. In low level processing, it can be used for parallelizing local window-based operations such as convolution, edge detection, linear and non-linear (e.g. median) filtering, and image thinning. In intermediate level it can be used to extract features of individual objects concurrently. In high level processing it can be used for processing several features or objects for object recognition, concurrently. The algorithmic structure/motif represented by the Farmer-Worker pattern is described in (Mattson, 1996).

3.4 Master-Worker Pattern

Intent

The Master-Worker pattern is used for parallelizing a class of problems which exhibit a synchronous form of parallelism. The master component divides the problem into several subtasks and distributes them to identical worker components. Each worker component performs computations on its assigned subtask iteratively, and communicates the intermediate results to its neighboring workers at the end of each iteration. The master component collates the final results returned by the worker components after a fixed number of such iterations.

Motivation

An extremum filter is a window-based non-linear operator which sharpens the blurred edges in an image to the original step edges (Kramer & Bruckner, 1975). The extremum
filter replaces the central pixel value within a filter window by the \textit{nearest} extreme pixel value occurring within the window. It can be expressed using the following equation

\begin{equation}
    f'(a, b) = \begin{cases} 
        \max\{f(i,j)\} & \text{if } \max\{f(i,j)\} - f(a,b) \leq f(a,b) - \min\{f(i,j)\} \\
        \min\{f(i,j)\} & \text{otherwise}
    \end{cases}
\end{equation}

where, $f'(a,b)$ represents the new pixel value, and $\max\{f(i,j)\}$ and $\min\{f(i,j)\}$ represent the maximum and minimum values (extreme values) occurring within a window centered at point $(a,b)$. Extremum filter is applied \textit{iteratively} so that the blurred edges converge to the original step edges. Kramer et al. (Kramer & Bruckner, 1975) have reported that at least 20-50 iterations were required to observe a complete convergence in a $27 \times 33$ image. The execution time required for operating on larger images can therefore be quite significant. In fact, it can be seen that the computational complexity of this operator with $M$ iterations, operating on a $m \times n$ image and using a $m \times m$ window, is $O(2Mm^2n^2)$. The extremum filter operator can be parallelized by dividing the image into several subimages, and filtering these subimages concurrently using different worker components. Each worker component communicates the required boundary information to its neighboring workers after every iteration. By using a set of $P$ processors, the computational complexity of the extremum filter operator can be reduced to $O(2Mm^2n^2/P)$, subject to the communication overheads.

\textbf{Structure}

The Master-Worker pattern consists of a master component and several identical worker components or processes as shown in Figure 3.3. The worker components are spatially arranged in a pipeline to reflect the communication structure of the partitioned problem which the pattern implements. The client interacts with the master component to parallelize certain application. The master component is responsible for partitioning the application into several subtasks, starting the worker components to process these subtasks, collecting the results returned by the workers, and finally returning the collected results to the client. The worker components are responsible for \textit{repeatedly} performing the computations on their assigned subtasks, and communicating the intermediate results to
their neighboring workers after every iteration. The Master-Worker pattern consists of one master and at least two workers.

![Figure 3.3: Master-Worker Pattern](image)

### Interaction

The interactions between the components of the Master-Worker pattern are shown in Figure 3.4.

- The client requests the master to parallelize a given application.
- The master component divides the application into several subtasks and starts the worker components to process these subtasks. The number of subtasks created is equal to the number of processors available.
- Each worker performs a fixed number of *compute-communicate* cycles. A compute-communicate cycle denotes an operation in which the workers compute on the data in their assigned subtasks and then communicate the intermediate results to their neighboring worker components. The workers return the computed results back to the master after performing a fixed number of these *compute-communicate* cycles.
- The master collates the results returned by the workers for the given application. The master returns the collated result to the client.
Chapter 3. Design patterns for parallelizing vision applications

Implementation

The Master-Worker pattern can be implemented by following the steps described below:

1. **Partition the work.** Specify how the problem task can be divided into a collection of subtasks. The number of subtasks created should be equal to the number of processors or machines available in the parallel system. Also, the amount of computational work in each subtask should be proportional to the speed factors of individual machines used in parallelization. For the filtering operation, one can partition the image into either horizontal or vertical blocks of subimages. Each subimage represents a subtask to be processed.

2. **Combine the results.** Specify how the final results should be collated from the results returned by the worker components. In the filtering example, the master component simply collates the filtered subimages onto the output image without any change.
Chapter 3. Design patterns for parallelizing vision applications

3. Specify the interaction between the master and the workers. This interaction can be specified as follows. The master starts the worker components and distributes a single subtask to each worker component. The master then waits for the workers to return the computed results. When all the workers communicate their computed results, the master terminates all the worker components. The master collects and returns the final result to the client.

4. Specify the interaction between the worker components. This interaction can be specified as follows. When a worker completes its computation in any compute-communicate cycle, it communicates the required intermediate results to its neighboring workers asynchronously. It then suspends its activities and waits to receive the intermediate results from its neighboring workers. Note that when a process sends a message asynchronously, it does not wait for the destination process to receive it. This implementation therefore does not lead to a deadlock condition.

5. Implement the master and the worker components according to the specifications outlined in previous steps.

Consequences

The Master-Worker pattern provides several benefits:

*Scalability and flexibility:* The Master-Worker pattern is scalable with respect to the addition of new workers. It is also flexible with respect to changing of the existing algorithms in the workers without involving major changes to the master. The client is not affected by such changes. Similarly, it is possible to change the algorithms for partitioning the work or co-ordinating the workers in the master component without affecting the client.

*Separation of concerns and efficiency:* The Master-Worker pattern separates the client code from the code for splitting the work, delegating the work to different workers, managing interactions between the workers, collecting the results from the workers, and handling the worker failures. The Master-Worker pattern can speed up computation time in many applications. However, it may not always be feasible to parallelize any application due to overheads in parallelization (see below).
The Master-Worker pattern suffers from the following liabilities:

**Feasibility:** The Master-Worker pattern may not always be feasible. The activities of partitioning of the work, starting and controlling the workers, delegating the work to the workers, managing the worker-worker communication, and collecting the final results, are time consuming. This pattern would be effective only when the time spent in these activities is significantly lower than the computing time required to execute a given application.

**Load balancing:** The Master-Worker pattern can suffer from serious load imbalances during its execution. This can happen when it is implemented on non-dedicated parallel systems, such as enterprise clusters (see section 2.5.3). Each worker in the Master-Worker pattern depends on the other workers to perform computations on its assigned subtask. A machine in an enterprise cluster can lead to reduction in performance of this pattern, when it is time-shared by other users while executing some worker component of the pattern. A static load distribution based on the speed factors of individual machines used in parallelization is effective only on dedicated parallel systems.

**Error Recovery:** It is hard to devise mechanisms to handle a failure in some worker component during the execution of this pattern. Since each worker is dependent on the other workers for performing its computations, such a failure can lead to a deadlock condition. It is also difficult to deal with the failure of communication between the master and the workers or between different workers.

**Applicability**

The Master-Worker pattern represents a parallel programming model for implementing synchronous parallel algorithms. It can be used to parallelize any vision application in which

- the data can be partitioned into several data sets
- each data set can be processed concurrently by different workers
- the processing of each data set requires an interaction between the worker components to exchange the intermediate results
Known Uses

The Master-Worker pattern has applications mostly at low level vision processing. The higher levels do not exhibit regularity in data structures and computation. In low level processing, it can be used for parallelizing iterative window-based operations such as spatial non-linear filters, and iterative relaxation algorithms used for image restoration and segmentation. The algorithmic structure/motif represented by the Master-Worker pattern is described in (Mattson, 1996).

3.5 Controller-Worker Pattern

Intent

The Controller-Worker pattern is used for parallelizing a class of problems in which each object or subtask of the problem can potentially interact with any other object or a subtask. The controller component divides the problem into several subtasks and distributes them to identical worker components. Each worker performs calculations on its assigned subtask, and communicates the intermediate results to some or all other worker components. The controller component collates the final results returned by the worker components.

Motivation

Histogram equalization is a popular grey scale transformation which is used for enhancing the contrast in an image. It aims to transform the image to have equally distributed brightness levels over whole of the brightness scale. A histogram $H$ of an image is a probability density function of the grey values in the image. If $n_k$ represents the number of pixels at a grey level $k$ and if $N$ denotes the total number of pixels in an image, then the histogram $H$ is defined as $H(i) = n_i/N$. Histogram equalization maps the original pixel values from a scale $[a, b]$ to the new values from a scale $[c, d]$ such that the desired output histogram is uniform over the whole new brightness scale $[c, d]$. The transformation function is monotonically increasing and is given by (Sonka et al., 1993)
Chapter 3. Design patterns for parallelizing vision applications

\[
f'(i, j) = \left(\frac{(d - c)}{N}\right) \sum_{k=a}^{b} H(k) + c
\]

(3.3)

where \( f \) and \( f' \) represent the original and transformed image functions, respectively.

Histogram equalization algorithm can be parallelized using the Controller-Worker pattern. The Controller divides the image into several subimages and sends each subimage to different worker. Each worker computes the partial histogram of its subimage and communicates it to all other workers. Each worker then combines these partial histograms to form a complete histogram of an entire image. The workers perform histogram equalization on their subimages (using equation 3.3) and return the transformed subimages to the Controller.

Structure

The Controller-Worker pattern consists of a controller component and several identical worker components or processes as shown in Figure 3.5. The client interacts with the controller to parallelize certain application. The controller component is responsible for partitioning the application into several subtasks, starting the worker components to
Chapter 3. Design patterns for parallelizing vision applications

process these subtasks, collecting the results returned by the workers, and finally returning
the collected results to the client. The worker components are responsible for performing
the computations on their assigned subtasks. Each worker may exchange intermediate
results with some or all other worker components, during the computation. The Controller-
Worker pattern consists of one controller and at least two workers.

Interaction

The interactions between the components of the Controller-Worker pattern are shown in
Figure 3.6.

![Figure 3.6: Object Interaction in the Controller-Worker Pattern](image)

- The client requests the controller to parallelize a given application.
- The controller divides the application into several subtasks and starts the worker
  components to process these subtasks. The number of subtasks created is equal to
  the number of processors available.
- Each worker performs computations on its assigned subtask and communicates the intermediate results to one or more worker components. The workers return the computed results back to the controller.
- The controller collates the results returned by the workers, and returns the collated result to the client.

Implementation

The Controller-Worker pattern can be implemented by following the steps described below:

1. **Partition the work.** Specify how the problem task can be divided into a collection of subtasks. The number of subtasks created should be equal to the number of processors or machines available in the parallel system. Also, the amount of computation in each subtask should be proportional to the speed factors of the individual machines used in parallelization. For the histogram equalization operation, we could either partition the image into horizontal or vertical blocks of subimages. Each subimage represents a subtask to be processed.

2. **Combine the results.** Specify how the final results should be collated from the results returned by the worker components. In the histogram equalization example, the controller component simply collates the transformed subimages onto the output image, without any change.

3. **Specify the interaction between the controller and the workers.** This interaction can be specified as follows. The controller starts the worker components and distributes a single subtask to each worker component. The controller then waits for the workers to return the computed results. When all the workers communicate their computed results, the controller signals the workers to terminate their processing. The controller collects and returns the final result to the client.

4. **Specify the interaction between the worker components.** Each worker may communicate (asynchronously) the intermediate results to some or all other worker components, and may wait to receive the same from some or every other worker component. Thus, this
interaction may sometimes involve global broadcasting of messages from each worker to all other workers.

5. Implement the controller and the worker components according to the specifications outlined in previous steps.

Consequences

The Controller-Worker pattern provides several benefits:

*Scalability and flexibility:* The Controller-Worker pattern is scalable with respect to the addition of new worker components. Increasing the number of worker components does not result in major changes to the controller or to the client program. Also, it is easy to change the program code in all worker components to realize different implementations.

*Separation of concerns and efficiency:* The Controller-Worker pattern separates the client code from the code for splitting the work, delegating the work to different workers, managing interactions between the workers, and collecting the results from the workers. The Controller-Worker pattern can speed up execution time of many computationally intensive applications. However, it may not always be feasible to parallelize a given application due to overheads in parallelization (see below).

The Controller-Worker pattern suffers from the following liabilities:

*Feasibility:* The Controller-Worker pattern may not always be feasible. The activities of partitioning of the work, starting and controlling the workers, delegating the work to the workers, managing the worker-worker communication, and collecting the final results, are time consuming. In fact, significant delays can occur in the worker-worker interactions especially when they involve global broadcasting of messages from each worker to all other workers.

*Load balancing:* The Controller-Worker pattern can suffer from serious load imbalances during its execution. This can happen when it is implemented on non-dedicated parallel systems, such as enterprise clusters (see section 2.5.3). Each worker in the Controller-Worker pattern may depend on the other workers to perform the computations on its
assigned subtask. A machine in an enterprise cluster can reduce the performance in this pattern, when it is time-shared by other users during the execution of some worker component within the pattern. A static load distribution based on the speed factors of individual machines used in parallelization is effective only on dedicated parallel systems.

**Error Recovery:** It is hard to devise mechanisms to handle a failure in some worker component during the implementation of this pattern. If each worker depends on the other workers for performing its computations, such a failure can lead to a deadlock condition. It is also difficult to deal with the failure of communication between the controller and the workers or between different workers.

**Applicability**

The Controller-Worker pattern can be used to parallelize any vision application in which

- the data can be partitioned into several data sets
- each data set can be processed concurrently by different workers
- the processing of each data set requires an interaction between some or all the worker components, to exchange intermediate results.

**Known Uses**

The Controller-Worker pattern has applications mostly at low and intermediate level processing. In low level processing, it can be used for parallelizing two-dimensional Fast Fourier Transforms. In the intermediate level, it can be used for parallelizing Hough transforms and connected component labeling algorithms.

An *iterative* variant of the Controller-Worker pattern can be realized by performing the compute-communicate cycles iteratively. Each worker component performs computations on its assigned subtask iteratively. Each worker communicates the intermediate results to some or all other worker components at the end of every iteration. However, a parallel implementation using an iterative variant of the Controller-Worker pattern involves huge
communication costs, and therefore may not result in any significant performance gains in many applications.

### 3.6 Divide-and-Conquer Pattern

**Intent**

The Divide-and-Conquer (DC) pattern is used for structuring applications in which either the data or the application algorithm is divided into several subtasks. Each subtask may be executed on single processor or may be further divided (recursively) into smaller subtasks. The subtasks are executed independently and concurrently producing several partial results. A set of combining functions are then applied on these partial results to produce the main result.

**Motivation**

An edge, a local boundary of some object in an image, represents a sharp discontinuity in the image function $f(x, y)$. It is described by a gradient that points in the direction of the largest growth of the image function. An edge has both magnitude and direction which is calculated using the gradient. The gradient is approximated by first-order differences and expressed as a gradient operator $\nabla f(x, y) = (\Delta_x f(x, y), \Delta_y f(x, y))$. A popular gradient operator is the Sobel edge detector which is represented by two convolution masks for finding edges in the horizontal ($\Delta_x$) and the vertical directions ($\Delta_y$) as shown below.

\[
\begin{align*}
-1 & & 0 & & 1 \\
-2 & & 0 & & 2 \\
-1 & & 0 & & 1 \\
\end{align*}
\]

(a)           

\[
\begin{align*}
1 & & 2 & & 1 \\
0 & & 0 & & 0 \\
-1 & & -2 & & -1 \\
\end{align*}
\]

(b)           

Figure 3.7: Convolution masks for finding a) horizontal edges and b) vertical edges

The direction of the edge at a point $(x, y)$ in the image is given by $\tan^{-1}(\Delta_y/\Delta_x)$,
while the edge magnitude is expressed as $\sqrt{\Delta_x^2 + \Delta_y^2}$. The Sobel edge detector can be parallelized using the DC pattern by computing the horizontal and vertical gradients concurrently. The horizontal and the vertical gradients can then be combined to compute the edge direction and the edge magnitude, using the expressions given above.

**Structure**

The DC pattern consists of a manager component and several distinct worker components or processes as shown in Figure 3.8. The manager component creates a set of worker components to process each subtask. Each worker may perform computations on its assigned subtask or may recursively divide it further into smaller subtasks for executing them on a different set of processor nodes.

![Figure 3.8: DC Pattern](image)

**Interaction**

The interactions between the components of the DC pattern are shown in Figure 3.9.

- The client requests the manager to parallelize a given application.
- The manager starts the worker components and distributes the subtasks to different worker components.
Chapter 3. Design patterns for parallelizing vision applications

Figure 3.9: Object Interaction in the DC Pattern

- Each worker component performs computation on its assigned subtask and returns the partial results to the manager. Alternatively, a worker may recursively divide its assigned subtask into smaller subtasks and execute them concurrently on a different set of processor nodes. A worker, in this case, acts as a manager for parallelizing its assigned subtask.

- The manager computes the main result from the results returned by the worker components.

- The manager returns the main result to the client.

Implementation

The DC pattern can be implemented by following the steps described below:

1. Design the manager component. The manager controls the worker components. It creates and schedules the worker components during the processing of the subtasks. If the DC pattern is used for implementing data parallelism, specify the dividing function which partitions the data into subtasks. However, if the DC pattern is used for implementing...
algorithmic parallelism, divide the application algorithm manually into distinct program units. The manager should create worker components to execute these program units. In both the cases, specify the combining function which combines the partial results returned by the worker components. In the Sobel edge detection example, a combining function in the manager combines the edge data returned by the worker components, in order to compute the edge direction and edge magnitude.

2. **Design the worker component.** Each worker may simply apply a computing function on its assigned subtask. Alternatively, a worker may serve as a manager for parallelizing its assigned subtask using a different set of processor nodes. Each worker should return the partial results (of the assigned subtask) to its corresponding manager. In the Sobel edge detection example, each worker component computes the edge data in the horizontal ($\Delta_x$) and the vertical ($\Delta_y$) directions, concurrently.

3. **Specify the interaction between the manager and the workers.** This interaction can be specified as follows. The manager starts the worker components and distributes a single subtask to each worker component. The manager then waits for the workers to return the computed results. When a worker communicates its result, the manager signals the worker to terminate its processing. In the Sobel edge detection example, the manager communicates complete image data to each worker and waits for receiving the edge data from all the workers.

4. **Implement the manager and the worker components** according to the specifications outlined in previous steps.

**Consequences**

The DC pattern provides several benefits:

*Separation of concerns:* The manager component separates the client code from the code in worker components used for performing the actual computations in the subtasks. Also, the code for creating and controlling the worker components is encapsulated in a manager component, separate from the client.
Chapter 3. Design patterns for parallelizing vision applications

Efficiency: The DC pattern provides a simple strategy of parallelizing an application. It can be used to achieve improved performance in many applications which can be divided (recursively) into smaller but independent computational units.

Error Recovery: It is relatively easy to devise mechanisms to handle a failure in some worker component during the execution of this pattern. This is due to the fact that all the worker components process their subtasks independently.

The DC pattern suffers from the following liabilities:

Scalability: The scalability of the DC pattern when used for implementing algorithmic parallelism is constrained by the amount of parallelism that can be achieved in the algorithm. In fact, the algorithm dictates the parallelism.

Load imbalances: The DC pattern may lead to load imbalances when used for implementing data parallelism. For example, equal distribution of the image data in the connected component labeling algorithm may lead to unequal load distribution when the connected components span only a small region in the image.

Applicability

The Divide-and-Conquer pattern can be used for parallelizing any vision application in which

- the data or the algorithm can be divided into several subtasks
- each subtask can be executed on a single processor or may recursively be parallelized using the divide-and-conquer principle
- all subtasks created can be processed concurrently on different processors without explicit communication between the processors

Known Uses

The divide-and-conquer parallel programming model has been used for parallelizing a number of vision algorithms. Stout (Stout, 1987) has proposed several divide-and-
conquer algorithms for image processing. Sunwoo et al. (Sunwoo et al., 1987) have used divide-and-conquer techniques to segment an image into different regions. Choudhary and Thakur have parallelized connected component labeling algorithms on coarse grained machines using the divide-and-conquer principle (Choudhary & Thakur, 1994). Hameed et al. (Hameed et al., 1997) have employed different divide-and-conquer approaches to parallelize a contour ranking algorithm on coarse grained machines.

3.7 Temporal Multiplexing Pattern

Intent

The Temporal Multiplexing (TM) pattern is used for processing several data sets or a sequence of image frames on multiple processors. Each processor processes a complete data set and executes the same program code.

Motivation

A computer-assisted sperm motility system enables studying the motion of the sperms in living organisms (Irvine, 1995). In human beings it is used for estimating the degree of male fertility. In a sperm motility system, a sequence of image frames of the sperm movement are captured over a given time frame. These image frames are then analyzed for finding the sperm and motion characteristics such as the sperm density, size and shape of the sperm heads, velocity of the sperms, and the shape of the motion trajectory. A sperm motility system involves a set of common preprocessing and feature extraction operations on the individual image frames. The module to compute the velocity of individual sperms, for example, involves simple operations such as image thresholding, noise suppression, removal of thin lines (sperm tails) or contaminating particles, segmentation, and finally region merging for extracting the sperm heads/cells. The processed image frames are then combined (superimposed) for tracking the motion trajectories of individual sperms and to compute the sperm velocities.

Since the preprocessing and feature extraction operations on individual image frames
are independent of each other, the TM pattern can be used to process each image frame concurrently. Performing data parallelism on individual image frames in such cases may not improve performance due to communication overheads and simplicity of the operations.

**Figure 3.10: TM Pattern**

**Structure**

The TM pattern consists of a manager component and several identical worker components or processes as shown in Figure 3.10. The manager creates, controls and schedules the worker components to process the data sets. It receives the data sets from an external component called the *data source*. The worker components are responsible for performing computation on individual data sets, and to return the processed values to an external component called the *data sink*. The TM pattern consists of one manager and at least two workers.

**Interaction**

The interactions between the components of the TM pattern are shown in Figure 3.11.

- The external data source supplies a sequence of data sets to the manager
- The manager assigns individual data sets to available workers. If all the workers are
busy, the manager suspends its activities until some worker is free to process a data set.

- Each worker processes its assigned data set, sends the processed values to a data sink component, and interacts with the manager for a new data set.

- The above two steps are repeated until there are no more data sets to be processed.

![Diagram](image)

**Figure 3.11: Object Interaction in the TM Pattern**

**Implementation**

The TM pattern can be implemented by following the steps described below:

1. **Design the manager component.** The manager controls the worker components. It creates and schedules the worker components for processing the data sets. The manager component maintains a queue of available worker components. When a worker requests a new data set, the manager adds it to the end of this queue. If the queue of available workers is not empty, the manager reads a data set from the data source and assigns it to the first available worker in this queue. However, when the queue is empty (all the
workers are busy), the manager suspends its activities until at least one worker is ready to process a data set. In the sperm motility system, the manager assigns each image frame to a separate worker. The manager, in this case, can also serve as a data source. It therefore maintains a repository of all the image frames to be processed by the worker components.

2. **Design the worker component.** Each worker should be designed to process the assigned data set, send the processed values to the data sink, and request a for new data set from the manager. In the sperm motility example, each worker performs a complete set of preprocessing and feature extraction operations on their assigned image frames. Each worker sends the processed image frames to the data sink component.

3. **Implement the manager and the worker components** according to the specifications outlined in previous steps.

**Consequences**

The TM pattern provides several benefits:

*Scalability and flexibility:* New worker components can be easily added without performing major changes to the manager component. Also, it is easy to change the program code in all worker components to realize different implementations.

*Efficiency:* The use of TM pattern enables scaling of the throughput to process the individual data sets in direct proportion to the number of processors used.

*Dynamic load balancing:* The TM pattern, like the Farmer-Worker pattern, provides an even distribution of the load while processing the data sets. The number of data sets processed by each worker is proportional to the speed of their corresponding nodes or processors.

The TM pattern suffers from the following liabilities:

*Effectiveness:* The TM pattern is effective only when there are more data sets/image frames than the number of processors. The parallelism in this pattern is expressed in terms of the number of data sets/image frames processed. When all the data sets/image
frames are processed, no further parallelism is available in the application.

Latency: The use of TM pattern does not improve the latency to process individual data sets, it remains unchanged in this pattern.

Applicability

The TM pattern can be used to parallelize any vision application in which

- it is required to process a collection/sequence of image frames or image data sets
- the processing of each image uses the same program code
- the images can be processed concurrently on different processors without explicit communication between the processors

Known Uses

The TM pattern is used for parallelizing complete data sets. Downton et al. (Downton et al., 1996) have used temporal multiplexing techniques in the postcode recognition system. They have used it for verifying the validity of postulated postcodes by matching them with the entries in a database of valid postcodes.

3.8 Pipeline Pattern

Intent

The Pipeline pattern is used for parallelizing applications which process a stream of data, and which can be divided into a sequence (pipeline) of several independent subtasks that are executed in a determined order. The data stream in the pattern is provided by a data source component. The processed results are collected by the data sink component. Each subtask is implemented by a worker component which reads a stream of data, processes it, and passes the processed results to another worker (or data sink) in the pattern.
Motivation

A vehicle identification system involves analyzing the images of the vehicles, for identifying the owners of the vehicles. Such a system, for example, can be used for tracking the identification of the vehicles, which break a specified speed limit on a motor highway or city roads. A high speed camera captures the images of the high speed vehicles which are then analyzed at a certain time of the day. A typical vehicle identification system consists of at least four distinct modules (subtasks) as shown in Figure 3.12.

The preprocessing module extracts the region in the image that surrounds the number plate. It then applies thresholding, edge detection and thinning operations on the extracted region in order to recover and skeletonize the characters in the number plate. The output of this module serves as an input to the feature extraction module, which extracts a number of features concerning each character. The feature vectors of all the characters in the number plate are then presented to the classification module. The classification module compares the feature vector of each character with a set of pre-stored exemplar feature vectors. A set of possible characters for each character in the number plate is then presented to the database search module.

The database search module searches a database of valid vehicle registration numbers for each complete set of characters that may potentially represent a number plate. The ones that match the database entries with the highest probabilities are then considered as recognized number plates. The database search module then outputs the identification of the vehicle from the database entry. For a given number plate image, if the system outputs more than one potential number plate entry, some verification (either manually or automated) needs to be devised to resolve the system ambiguity.
The distinct modules of the vehicle identification system can be easily structured using the pipeline pattern. Each module can run concurrently on different processors and interact with its neighboring modules only by exchanging streams of data.

**Structure**

The Pipeline pattern consists of a data source, a data sink, and several worker components as shown in Figure 3.13. The data source provides a sequence of input values (having the same structure or data type) in the pipeline. The data sink collects the processed values from the end of the pipeline. Each worker component is responsible for receiving the data from its preceding worker (or data source), processing this data, and sending the processed results to the following worker (or data sink). The first and the last worker components communicate with the data source and the data sink components, respectively. The intermediate worker components communicate only with their immediate neighbors. Note that a Pipeline pattern does not provide for dividing the application into different subtasks. It provides only a structure to an application that is divided manually into different subtasks. The client is responsible for creating, starting and terminating the components in the Pipeline pattern.

![Diagram of Pipeline Pattern](#)
Interaction

The interactions between the components of the Pipeline pattern are shown in Figure 3.14.

- The client calls the data source component to read the data sets.
- The data source component reads and attempts to send a new data set to the first worker. If the first worker is busy with processing a previous data set, the data source component suspends itself until the worker is ready to receive the current data set.
- Each intermediate worker (not shown in the figure for brevity) retrieves (pulls) a data set from its preceding worker, processes it, and sends (pushes) the processed data to its successor. A worker may suspend its activities temporarily, if the data from the preceding worker is not available, or if the worker following immediately is not waiting for the data.
- The last worker sends the processed data set to the data sink and waits for a new data set from its predecessor.
Chapter 3. Design patterns for parallelizing vision applications

- The last three processing steps are repeated until there are no more data sets to be processed in the pipeline.
- The data sink sends the processed data sets to the client.

Implementation

The Pipeline pattern can be implemented by following the steps described below:

1. **Divide the application.** The application should be manually divided into a sequence of functional units or subtasks. The processing in each subtask must depend only on the output of its direct predecessor. The computational load in each subtask should be proportional to the speed factors of the individual processors available for parallelizing the application. In the vehicle identification system, the application can be divided into four distinct functional units, namely, preprocessing, feature extraction, classification and database search.

2. **Design the data source and data sink components.** These can be designed in two different ways: a) Both the data source and the data sink are designed as separate components which are executed concurrently with respect to the client. The client calls the data source component to read and output the data stream into the pipeline, and waits for the data sink to return the final results collected during the execution of the pipeline. b) Alternatively, the client functions as a data source (or data sink) and creates a separate component for data sink (or data source). A client cannot perform both these tasks by itself, since it does not result in any performance gain on using this pattern. In the vehicle identification example, the data source may be designed as a separate component which reads vehicle images from the specified files and presents them to preprocessing module. The data sink component may simply store the details of each number plate and its potential owner(s) in a specified file.

3. **Design the worker components.** Each worker component should repeatedly receive a data set from its predecessor, processes it, and output the processed data set to its successor. Each worker should be implemented as a separate program unit that performs the required computation on its data set. In the vehicle identification example, each worker
performs specified operations on its input data and passes its output to the neighboring worker or data sink.

4. Specify the interaction between different components in the pattern. This interaction can be specified by using inter-process communication calls supported by a message-passing library (section 2.1.1). Note that each worker should format the results in order to pass them to its successor in the pipeline.

5. Implement the components and start the pipeline. The components in the pattern can be implemented according to the specifications given in previous steps. The client starts each component as a separate thread or process. The processing in the pipeline starts when the data source outputs the data sets to the first worker in the pipeline. Each data set is transformed by different worker components in the pipeline and is finally collected by the data sink. When there are no more data sets to be processed, the client terminates all the components of the pattern, after collecting the processed results from the data sink.

Consequences

The Pipeline pattern provides several benefits:

Flexibility: Since the worker components in the Pipeline pattern are independent and interact only by exchanging streams of data, they can be easily replaced by more efficient components having the same functionality. The worker components can be reused in different situations. Also, new worker components can be easily added to refine the functionality of the existing pipeline.

Efficiency: The Pipeline pattern helps in increasing the system throughput and reduce the latency in applications which process long streams of data. However, the use of Pipeline pattern for improving the application performance is feasible only when the granularity of each worker is sufficiently high. The time required to transfer the data between the worker components should be relatively lower than the time required to perform the computations on each worker component.
The Pipeline pattern suffers from the following liabilities:

Sharing global information: Sharing of global information between different components in the Pipeline pattern is inefficient and does not provide full benefits of the pattern.

Load balancing: Like the Master-Worker pattern, Pipeline pattern can suffer from serious load imbalances during its execution on enterprise clusters (section 2.5.3). Throughput and latency are influenced by the speed of the slowest worker component in the pattern.

Error Recovery: It is difficult to handle failures in the worker components during the execution of this pattern. Each worker is dependent on other workers for performing its computations. Consequently, a failure in any worker component can lead to a significant loss in processing time. In many cases, the application may need to be started from the beginning.

Scalability: An application parallelized using a Pipeline pattern is usually not scalable with respect to addition of processors used for parallelization. This is because the number of worker components in a Pipeline pattern are defined by the number of subtasks comprising the application.

Applicability

The Pipeline pattern can be used to parallelize applications in which

- it is necessary to process a long stream of data values
- the application is composed of a sequence of independent functional units which process the data stream independently, but in a determined order
- the functional units communicate with each other only by exchanging streams of data

Known Uses

The Pipeline pattern has applications at all levels of vision processing. At the low level, it can be used for parallelizing Canny edge detector (Sonka et al., 1993), when applied on
a sequence of image frames. The Canny edge detector is composed of several independent functional units and is therefore easily implemented using a Pipeline pattern (Ruff, 1988).

Note that the scalability of a Pipeline pattern may be increased by employing two or more Pipeline patterns for parallelizing a single application. Each Pipeline pattern can concurrently process a part of the data stream (if feasible) in the application. Using two or more Pipeline patterns to parallelize a single application can be considered as a variant of the Pipeline pattern. We call this variant the Multiple Pipeline pattern. Another variant of the Pipeline pattern (used in (Downton et al., 1996)) can be realized by making the pipeline communications 'both ways'. This enables output of one or more Pipeline components to be used as an input (feedback) of relevant component(s) in the Pipeline.

3.9 Composite Pipeline Pattern

Intent

The Composite Pipeline pattern consists of a pipeline of design patterns and/or sequential components which together parallelize a complete vision application processing a continuous stream of data. It provides a structure to these applications which can be parallelized by dividing these into several independent functional units that communicate with each other only by exchanging streams of data. Each functional unit in turn may be parallelized by using relevant design patterns or may be implemented as a sequential component.

Motivation

Consider the vehicle identification system as outlined in section 3.8. Since the input to each module depends on the output of the previous module, the performance of the overall system depends on the speed of the slowest module. The use of a Composite Pipeline pattern in this situation can lead to improved system performance compared to a simple pipeline implementation. Each module in this system (see Figure 3.15) may be parallelized by dividing the data set within each module into subtasks and processing these
Chapter 3. Design patterns for parallelizing vision applications

subtasks concurrently (data parallelism). Alternatively, each data set may be processed on a different processor without data partitioning (temporal multiplexing).

![Vehicle identification system](image)

Figure 3.15: Vehicle identification system

For example, the preprocessing operations on each image may be performed concurrently on different processors. Similarly, the search for database entries for different number plates may be executed on different processors. Both these modules exhibit temporal multiplexing form of parallelism. In the feature extraction and classification modules, each character in a image frame may be processed on a separate processor (data parallelism). However, such parallelism may not always be feasible if the communication overheads are too high. In such cases, temporal multiplexing alone may be used to increase the system performance.

**Structure**

The structure of the Composite Pipeline pattern is as shown in Figure 3.16. It is similar to the Pipeline pattern. It has a *data source* which provides the inputs, a *data sink* which collects the outputs and a sequence of design pattern and/or sequential worker components that process the input stream of data. We shall refer to the design patterns and the sequential worker components as *functional components* of the pattern. Each functional component is responsible for receiving the data from its predecessor, processing this data, and sending the processed results to it successor. Note that the Composite pattern, like the Pipeline pattern, does not provide for dividing the application into different subtasks. It only provides a structure to an application that is divided manually into different functional components. The client is responsible for creating, starting and terminating the components in the Composite Pipeline pattern.
Chapter 3. Design patterns for parallelizing vision applications

Interaction

The interactions between the components of the Composite Pipeline pattern are shown in Figure 3.17.

- The client calls the data source component to read the data sets.
- The data source component reads and attempts to send a new data set to the first functional component. If the first functional component is busy with processing a previous data set, the data source component suspends itself until the component is ready to receive the current data set.
- Each intermediate functional component (not shown in the figure for brevity) retrieves (pulls) a data set from its predecessor, processes it, and sends (pushes) the
processed data to its successor. A functional component may suspend its activities temporarily, if the data from the preceding component is not available, or if the following component is not waiting for the data.

- The last functional component sends the processed data set to the data sink and waits for a new data set from its predecessor.

- The last three processing steps are repeated until there are no more data sets to be processed in the pipeline.

- The data sink sends the processed data sets to the client.

**Implementation**

The Composite Pipeline pattern can be implemented by following the steps described below:

1. **Divide the application.** The application should be manually divided into a sequence of functional units. The processing in each functional unit must depend only on the output of
its direct predecessor. For example, in the vehicle identification system, the application is divided into preprocessing, feature extraction, classification and database search modules.

2. Design the data source and data sink components. These components, like in the Pipeline pattern, can be designed as two separate components different from the client. Alternatively, the client can function as a data source (or data sink) and create a separate component for data sink (or data source).

3. Design the functional components. Design each functional component as an independent program unit which runs sequentially or which can be parallelized by using relevant design pattern. Each functional component must repeatedly retrieve a data set from its predecessor, processes it, and output the processed results to its successor. In the vehicle identification system, some or all the modules may be designed to implement either data parallelism or temporal multiplexing on their assigned data sets.

4. Specify the interaction between different components in the pattern. This interaction can be specified by using inter-process communication calls supported by a message-passing library (section 2.1.1).

5. Implement the components and start the pipeline. The components in the pattern are implemented according to the specifications given in previous steps. The processing in the pipeline starts when the data source outputs the data sets to the first functional component in the pipeline. Each data set is transformed by different functional components in the pipeline and is finally collected by the data sink. When there are no more data sets to be processed, the client terminates all the components of the pattern, after collecting the processed results from the data sink.

Consequences

The Composite Pipeline pattern provides several benefits:

Flexibility: Since the functional components in the Composite Pipeline pattern are independent and interact only by exchanging streams of data, they can be easily replaced by more efficient components having the same functionality. For example, a slow sequen-
tial worker component may be replaced by an equivalent parallel functional component. The functional components can be reused in different situations. Also, new functional components can be easily added to refine the functionality of the existing pipeline.

Efficiency: The Composite Pipeline pattern can achieve better performance than a plain Pipeline implementation. A slow worker component in the plain Pipeline implementation can be identified and could possibly be implemented as a parallel functional component. However, the use of Composite Pipeline pattern is effective only when the granularity of each functional component is sufficiently high.

The Composite Pipeline pattern suffers from the following liabilities:

Load balancing: The Composite Pipeline pattern, like the Master-Worker and the Pipeline pattern, can suffer from serious load imbalances during its execution on the enterprise clusters (section 2.5.3). However, these load imbalances can possibly be reduced by using the Farmer-Worker or the Temporal Multiplexing pattern, to parallelize relevant functional components. Both these patterns have dynamic load balancing property.

Error Recovery: It is difficult to handle failures in functional components during the execution of this pattern. Each functional component is dependent on the other components for performing its computations. Consequently, a failure in any functional component can lead to a significant loss in processing time.

Applicability

The Composite Pipeline pattern can be used to parallelize applications in which

- it is necessary to process a long stream of data values.
- the application is composed of a sequence of independent functional units which process the data stream independently, but in a determined order.
- the functional units communicate with each other only by exchanging streams of data.
Chapter 3. Design patterns for parallelizing vision applications

- each functional unit may be implemented as a sequential component or may in turn be parallelized using relevant design pattern.

**Known Uses**

The Composite Pipeline pattern is an architectural pattern which is used for parallelizing complete vision systems. Singh (Singh et al., 1991) and Schaeffer (Schaeffer et al., 1993) have used composite pipeline principle to parallelize an image rendering application. They used temporal multiplexing to speed up individual stages of the pipeline. Downton *et al.* (Downton et al., 1996) later proposed the principle of composite pipeline as a design methodology for parallelizing embedded image processing applications, and applied it to parallelize the image coding and postcode recognition applications. They proposed both data and algorithmic parallelism (in addition to temporal multiplexing) to speed up individual stages of the pipeline. A variant of the Composite Pipeline pattern (used in (Downton et al., 1996)) can be realized by making the pipeline communications ‘both ways’. This enables output of one or more Composite Pipeline components to be used as an input (feedback) of relevant component(s) preceding in the pattern.

### 3.10 Summary

Design patterns for parallel vision applications represent designs or methods used for parallelizing these applications on various parallel architectures. Although the literature on parallelization of vision algorithms is vast, there has been no previous efforts to abstract and document the design information in these parallel implementations. In this chapter we have attempted to capture and document this design information in the form of design patterns. These design patterns can be used for implementing parallel solutions to many vision algorithms/applications on coarse-grained parallel machines, such as a cluster of workstations. Each pattern has been described in a uniform way using a template. The template provides description of how each pattern works, where it should be applied and what are the trade-off in its use.

The design patterns presented in this chapter include Farmer-Worker, Master-Worker,
Chapter 3. Design patterns for parallelizing vision applications

Controller-Worker, Divide-and-Conquer (DC), Temporal Multiplexing, Pipeline, and Composite Pipeline. The Farmer-Worker pattern is used for parallelizing embarrassingly parallel algorithms, while Master-Worker pattern and Controller-Worker pattern are used for parallelizing problems exhibiting synchronous form of parallelism. Divide-and-Conquer pattern is used for parallelizing algorithms that use a recursive strategy to split a problem into smaller subproblems and merge the solution to these subproblems into final solution. Temporal Multiplexing pattern is used for processing several data sets or image frames on multiple processors. Finally, Pipeline and Composite Pipeline patterns are used for parallelizing applications which can be divided into a sequence (pipeline) of several independent subtasks that are executed in a determined order. In the Composite Pipeline pattern, each subtask may be further parallelized using other relevant design patterns.
Chapter 4

Low level algorithms

The design patterns described in previous chapter can be used for parallelizing a majority of vision algorithms on coarse-grained parallel machines, such as workstation clusters. In the remaining part of this thesis, we use and evaluate the applicability of these patterns for parallelizing some representative vision algorithms on a cluster of workstations. There are two different ways in which this can be done a) for a given design pattern, one can describe a set of vision algorithms which can be parallelized using this pattern, alternatively, b) for a given vision algorithm, one can describe a set of one or more design patterns which can be used to parallelize this algorithm.

We follow the second approach by grouping algorithms in some order (e.g. low level, intermediate level, and high level in computer vision), and describing various design patterns that can be used to parallelize these algorithms. This approach ensures logical consistency of describing algorithms or techniques used in a given domain, such as computer vision. This chapter therefore discusses parallelization of some representative low level vision algorithms using the appropriate design patterns. Chapter 5 discusses parallelization of some intermediate level algorithms, while chapter 6 discusses parallelization of some representative high level algorithms/applications. We begin this chapter by describing characteristics of low level algorithms.

Low level algorithms aim at improving the image data by suppressing noise or unwanted
distortions, and enhancing some image features important for further processing and/or for human interpretation. The input and output to these algorithms are pixel based intensity images. The computations involved in these algorithms are pixel based image transformations which use a large number of simple mathematical operations on the pixel values in an input image to compute a new set of pixel values in the output image. This chapter discusses parallelization of some representative low level vision algorithms using the design patterns described in Chapter 3.

Low level vision algorithms can be broadly classified into two categories depending on the size of the pixel neighborhood used for calculating the new pixel value.

- **Local algorithms:** In local algorithms, the value of a processed pixel depends only on the values of the pixels placed in its local neighborhood (window). The size of the neighborhood in the local algorithms may be fixed, as in Sobel edge detection and thresholding operations, or may vary, as in convolution and filtering operations. We also classify the point operations in this category where the value of the new pixel depends only on the original value of that pixel (e.g. brightness correction). The local algorithms can be further classified as iterative and non-iterative. An example of an iterative local algorithm is the extremum filter described in section 3.4, while, the edge detection algorithm using the Sobel edge operator is an example of a non-iterative local algorithm.

- **Global algorithms:** In global algorithms, the value of a processed pixel may depend on values of all pixels covering large neighborhoods or even entire image. The algorithms in this category are further classified as global fixed and global varying. In the global fixed algorithms, the value of a processed pixel depends on the values of all pixels in the input image. Some examples of the global fixed algorithms are: histogram equalization, and the two dimensional discrete Fourier transform. In the global varying algorithms, the value of a new pixel may depend on the pixels in entire input image, or on the pixels in small region of the input image. For example, in a connected component labeling algorithm, a connected component may span only a small region or it may be spread over the entire image. The amount of computation in global fixed algorithms therefore depends only on the size of the input image,
while the amount of computation in global varying algorithms depends on both, the size and the contents of the input image.

The classification scheme described above was used by Choudhary and Patel (Choudhary & Patel, 1990) to provide an insight into the performance of an algorithm based on its communication requirements. We have extended it further to introduce the iterative and non-iterative class of local algorithms. The extended classification scheme enables identification of relevant design patterns which can be used for parallelizing the low level algorithms.

The rest of the chapter is organized as follows. Section 4.1 outlines the methods which can be used to parallelize most of the low level algorithms. Section 4.2 describes the scheme that is used in partitioning the image data. The remaining sections present the experimental results of parallelizing various representative low level vision algorithms. Section 4.3 presents parallelization of a histogram equalization algorithm which is a global algorithm used for contrast enhancement. Section 4.4 discusses various filtering operations and their parallel implementations. Section 4.5 presents results of parallelization of a two-dimensional Fourier transform. Finally, section 4.6 discusses parallelization of an image restoration algorithm using Markov random field models.

The algorithms presented in this chapter (and those in two chapters following immediately) have been implemented on a network of up to sixteen workstations. Each workstation is a Sun SPARCstation 5 machine with 32 Mbytes of local memory and a clock speed of 170 MHz. All workstations thus have the same speed factors (a workstation with a speed factor of 2 is twice as fast as a workstation with a speed factor of 1). The program code for implementing various parallel algorithms using corresponding design patterns has been written in C++ and the PVM message-passing kernel (Sunderam, 1990). The performance of the corresponding parallel implementations have been measured in terms of execution times and program speedups. The speedup of a parallel program is defined as

\[
\text{speedup} = \frac{\text{execution time on one workstation}}{\text{execution time on } p \text{ workstations}} \quad (4.1)
\]
4.1 Parallelization of low level algorithms

Most of the low level vision algorithms are parallelized by partitioning the image into subimages, and processing these subimages concurrently using different processors. Using this strategy, Siegel et al. (Siegel et al., 1992) parallelized a local convolution algorithm using two distinct approaches, namely, complete sums and partial sums. In the ‘complete sums’ approach, all the data needed by a processor to process its subimage is transferred to it before the computation. The processors then work independently without interacting with each other during the computation. With the ‘partial sums’ approach, each processor performs computation on its subimage and interacts with other processors to exchange the intermediate results during the computation. We extend these two approaches to parallelize most of the low level algorithms.

The local non-iterative algorithms can be parallelized using the ‘complete sums’ approach. They can be implemented by using the Farmer-Worker pattern (section 3.3). The local iterative and the global low level algorithms can be parallelized using the ‘partial sums’ approach. However, the algorithms within these classes exhibit different communication patterns. In a local iterative algorithm, each processor communicates with its neighbors after every iteration. These communications are regular and can be determined before the start of the computation. Local iterative algorithms can therefore be parallelized using the Master-Worker pattern (section 3.4). The global algorithms usually involve all-to-all processor communications. In certain cases, these communications may be determined before the start of the computation, as in the computation of a two dimensional fast Fourier transform of an image. But in other cases, they are determined dynamically or only after the start of the computation, as in the connected component labeling algorithm. The global algorithms are therefore parallelized using the Controller-Worker pattern (section 3.5).

Another important consideration in parallelization of the low level algorithms is the number of image partitions or subimages created for concurrent execution. The number of subimages created in the local non-iterative algorithms should be about two to three times more than the number of processors (workers) used in parallelization. This maximizes the degree of parallelism achievable in an application and results in better performance as
Chapter 4. Low level algorithms

4.2 Partitioning the image data

The performance of a low level algorithm parallelized on a cluster of workstations depends on a partitioning of the image into subimages, and corresponding communication overheads. The communication overheads are directly related to the way the image is partitioned. The communication overheads arise due to the distribution of subimages to the worker processors, exchange of the intermediate results (if applicable), and the collection of final results from the worker processors. There are many different methods to partition a given image into subimages. We use a simple row partitioning method in which an image is horizontally divided into a given number of subimages as shown in Figure 4.1. The row partitioning method allows one to divide a given image into any number of subimages of appropriate sizes. Thus each processor can be assigned a proportional workload based on its speed factor (Angus et al., 1989).

Figure 4.1 (a) shows the row partitioning of an image into distinct subimages (non-overlapping) for the global algorithms. Such algorithms do not need pixel values from other subimages in order to perform computations on the boundary pixels of any subimage. Figure 4.1 (b) shows the row partitioning scheme for parallelizing the local low level

An effective speed factor of a machine at any instant of time is the fraction of its CPU time that is dedicated for processing the subimage. The effective speed factor of a machine can vary over time depending on the workload (of external processes) on that machine. Note that this strategy of using the workloads to divide the image into subimages ensures only static load distribution. It is effective only when the application is parallelized on a dedicated workstation cluster (section 2.5.3), where the speed factors are always constant.

The number of subimages created in the local iterative and global low level algorithms should however be equal to the number of processors available. This is because each worker is required to interact with other workers to exchange the intermediate results during the computation. The computational workloads in the subimages, if measurable, should be proportional to the effective speed factors of the corresponding processors used in parallelization.
Figure 4.1: Partitioning of an image. a) Row partitioning b) Row partitioning with data that is to be overlapped and/or communicated

algorithms. In a local low level algorithm (except point operations), the value of a boundary pixel in any given subimage may depend on the values of the pixels present in other subimage(s). Therefore, each subimage also has an additional number of overlapping rows belonging to its neighboring subimages as shown in Figure 4.1 (b). In local iterative algorithms, these overlapping rows are communicated between the neighboring workers after every iteration.

Other methods for partitioning an image are column, diagonal, cross and heuristic. Both row and column partition methods are similar, hence, either of them could be used for partitioning the image. The diagonal partitioning method involves dividing the image into diagonal strips. This method is however difficult to implement and becomes extremely complicated when parallelizing local iterative algorithms. The cross partition method involves dividing the image in both horizontal and vertical directions. The number of subimages created using this method is always a square number. This places a restriction on the number of processors that can be used in parallelization, especially, in the algorithms parallelized using the ‘partial sums’ approach.

The heuristic partition method was proposed and used by Lee and Hamdi (Lee &
Hamdi, 1995) to parallelize the local convolution operation on a network of workstations. Their algorithm can partition the image into any number of subimages using both horizontal and vertical partitioning directions. However, both heuristic and cross partition methods involve rectangular shaped subimages. In local iterative algorithms, many worker processes may be required to exchange their intermediate results with eight other worker processes. In row or column partitioning, each worker process is required to interact with at the most two other worker processes. Therefore, the row partitioning method has number of advantages compared to the other partitioning methods.

### 4.3 Grey scale transformations

Grey scale transformations modify the brightness of the pixels in an image based on the properties of the pixels itself. They are used for enhancing the contrast and improve the appearance of an image so that it could be easily interpreted by a human observer. The most common grey scale transform for contrast enhancement is histogram equalization which was described in section 3.5.

Histogram equalization is a global low level algorithm. In this section, we present the experimental results of parallelizing this algorithm (as outlined in section 3.5) using the Controller-Worker pattern. The execution times for the histogram equalization algorithm parallelized using different number of workstations are displayed in Table 4.1. A plot of these execution times and the speedups achieved for this algorithm are shown in Figure 4.2.

The execution time for the histogram equalization algorithm on a single workstation is of the order of few seconds. However, the time spent in *all-to-all* worker communications is relatively large compared to the time spent in the actual computation. Hence, the execution time of the parallel algorithm increases significantly with increase in the number of workstations, even for 512x512 and 1Kx1K images. For a 2Kx2K image there is slight improvement in execution time until about five to six workstations (Figure 4.2), due to increase in computation time. However, the execution time increases for seven or more workstations. Hence, global algorithms involving all-to-all worker communications, but relatively lower execution time, should preferably be executed on a single workstation.
Chapter 4. Low level algorithms

Table 4.1: Execution time in (min:sec) for histogram equalization

<table>
<thead>
<tr>
<th>Image Size</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
<th>10</th>
<th>12</th>
<th>14</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>512x512</td>
<td>0:01</td>
<td>0:01</td>
<td>0:02</td>
<td>0:02</td>
<td>0:03</td>
<td>0:04</td>
<td>0:04</td>
<td>0:04</td>
<td></td>
</tr>
<tr>
<td>1Kx1K</td>
<td>0:04</td>
<td>0:04</td>
<td>0:15</td>
<td>0:15</td>
<td>0:16</td>
<td>0:16</td>
<td>0:17</td>
<td>0:22</td>
<td>0:23</td>
</tr>
<tr>
<td>2Kx2K</td>
<td>0:14</td>
<td>0:13</td>
<td>0:15</td>
<td>0:16</td>
<td>0:17</td>
<td>0:22</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.2: Performance of histogram equalization

4.4 Image filtering

Image filtering algorithms are image transforms that use a local neighborhood of a pixel in the input image to produce a new pixel value in the output image. A filter may be classified as linear or nonlinear. Linear filters calculate the new pixel value $f'(i, j)$ as a linear combination of the pixel values in a local neighborhood $N$ of the pixel $f(i, j)$ in the input image. A common class of linear filters are the convolution-based filters which are described in the next section. Linear filters, when used for removing noise in an image, blur sharp edges in that image. Nagao (Nagao & Matsuyama, 1979) and Lee (Lee, 1983) therefore suggested edge preserving non-linear filters, which, not only remove noise but also preserve sharp edges in a given image. Non-linear filters are discussed in section 4.4.2 and section 4.4.3.
4.4.1 Convolution

Convolution is a fundamental operation in image processing. It is used in image smoothing, edge or line detection (Sonka et al., 1993), feature extraction, and template matching (Ranka & Sahni, 1990). If \( \mathcal{N} \) is a set of neighboring points around a point \((a, b)\) in the image, and if \( h \) is a \( mxm \) convolution mask of co-efficients, the convolution \( f'(a, b) \) at \((a, b)\) is given by

\[
f'(a, b) = \sum_{(i,j) \in \mathcal{N}} h(i - c, j - d) f(i, j)
\]

where, \((c, d)\) is the displacement of the origin of \( h \) relative to that of \( f \). On a sequential machine, the computational complexity to perform the convolution operation on an image of size \( nxn \) is \( O(n^2m^2) \). This operation can be very time consuming when the size of the image and/or the size of the convolution mask is large. The execution time of this operation can be reduced by dividing the image into subimages, and convolving these subimages concurrently using different processors. By using a set of \( P \) processors, the computational complexity of the convolution operation can be reduced up to \( O(n^2m^2/P) \).

Table 4.2: Execution time in (min:sec) for the convolution operation

<table>
<thead>
<tr>
<th>Image Size</th>
<th>Window Size</th>
<th>Number of Workstations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>512x512</td>
<td>3x3</td>
<td>0:05</td>
</tr>
<tr>
<td></td>
<td>7x7</td>
<td>0:19</td>
</tr>
<tr>
<td></td>
<td>11x11</td>
<td>0:44</td>
</tr>
<tr>
<td></td>
<td>15x15</td>
<td>1:22</td>
</tr>
<tr>
<td>1Kx1K</td>
<td>3x3</td>
<td>0:19</td>
</tr>
<tr>
<td></td>
<td>7x7</td>
<td>1:18</td>
</tr>
<tr>
<td></td>
<td>11x11</td>
<td>3:04</td>
</tr>
<tr>
<td></td>
<td>15x15</td>
<td>5:28</td>
</tr>
<tr>
<td>2Kx2K</td>
<td>3x3</td>
<td>1:29</td>
</tr>
<tr>
<td></td>
<td>7x7</td>
<td>5:25</td>
</tr>
<tr>
<td></td>
<td>11x11</td>
<td>12:28</td>
</tr>
</tbody>
</table>
Convolution operation can be parallelized by using the Farmer-Worker pattern. Table 4.2 shows the execution times of the parallel convolution operation obtained by varying different parameters such as the window size, image size, and the number of workstations used in parallelization. The entries in this table enable study the influence of these parameters on the execution time and the speedup of the parallel convolution operation. We can make two different observations from this table. Firstly, by keeping the window size fixed, we can observe the performance results by varying both, the image size and the number of workstations used in parallelization. Secondly, by keeping the image size fixed, we can observe the performance results by varying the window size and the number of workstations used in parallelization.

The execution times and the speedups achieved for the parallel convolution operation using a 3x3 and a 15x15 window (window size fixed), for example, are shown in Figure 4.3 and Figure 4.4, respectively. Figure 4.3 shows that for a small window, the execution time decreases upon increase in the number of workstations used in parallelization. However, the execution time gradually increases when the number of workstations are increased beyond seven or eight. Corresponding speedup curves show a similar behavior. The increase in the execution times or the decline in the corresponding speedups after using eight or more workstations, is due to increase in the percentage of the communication time with respect to the corresponding computation time. However, when the window
size is larger (Figure 4.4), more computations are needed at each pixel in the convolution operation. Since the communication time in a 15x15 convolution operation is nearly the same as that in a 3x3 operation, the ratio of the computation time to the communication time is dominated by the computation time. This results in relatively greater speedups with increase in the number of workstations used in parallelization.

We can observe similar results by keeping the image size fixed, but varying the window size and the number of workstations used in parallelization. Figure 4.5 shows the performance results of the convolution operation on a 1Kx1K image. The observed speedups increase as the window size is increased. As in the above case, a larger window size implies more computations in the convolution operation. Therefore, as the time spent in communicating the subimages and the results is almost the same across windows of different sizes, the ratio of the computation time to the communication time increases. Hence, higher speedups can be obtained with the increase in window size and, the number of workstations used in parallelization.

![Graph](image.jpg)

Figure 4.4: Performance of the convolution operation using a 15x15 window
Chapter 4. Low level algorithms

The convolution algorithm was parallelized by Lee and Hamdi (Lee & Hamdi, 1995) on a network of SUN Sparc IPC workstations. They used a heuristic partitioning method (section 4.2) to partition the image into several different subimages of the same size. The number of subimages created were equal to the number of workstations used in parallelization. However, this partitioning scheme can reduce the performance of the parallel convolution algorithm in some cases as explained below:

- Firstly, the machines in a cluster of workstations may have different processing speeds or speed factors. Assigning subimages of the same size to such machines will lead to load imbalances. The size of a subimage assigned to any workstation should therefore be proportional to its effective speed factor.

- Secondly, even if all the workstations used in parallelization have the same speed factors, it is difficult to distribute these subimages to all workstations at the same time. There is always some delay before the last workstation gets its subimage and starts processing. This can cause some reduction in the overall performance of the parallel implementation.

- Finally, the performance of a parallel convolution algorithm implemented on an
enterprise cluster (section 2.5.3), will degrade significantly if a participating machine is time-shared to run other processes. Each machine in an enterprise cluster is time-shared between different users.

Hence, the heuristic partitioning method can sometimes result in significant reduction in the overall performance of the parallel convolution algorithm.

Table 4.3: Performance of the Farmer-Worker pattern on varying the external load and number of subtasks. The execution time (min:sec) displayed are for the convolution operation (window size 15x15).

<table>
<thead>
<tr>
<th>Row No.</th>
<th>External Load (Y/N)</th>
<th>Number of Workstations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1  2  4  6  8  10  12 14 16</td>
</tr>
<tr>
<td>1 (o)</td>
<td>N</td>
<td>5:28 2:49 1:22 0:56 0:41 0:40 0:32 0:31 0:26</td>
</tr>
<tr>
<td>2 (•)</td>
<td>Y</td>
<td>5:28 2:52 1:25 1:01 0:42 0:42 0:34 0:35 0:27</td>
</tr>
<tr>
<td>3 (*)</td>
<td>Y</td>
<td>5:28 5:10 2:24 1:36 1:11 0:58 0:48 0:41 0:36</td>
</tr>
</tbody>
</table>

Figure 4.6: Performance of the Farmer-Worker pattern in convolution operation on varying the processor load and number of subtasks (window size 15x15)

To overcome these limitations, we use the row partitioning method (in the Farmer-Worker pattern) to partition the image into several different subimages of the same size. However, the number of subimages created is at least two times more than the number of workstations used in parallelization. Each machine would therefore process a proportional number of subimages according to its speed factor. Table 4.3 shows the performance results
of the convolution operation, parallelized using two different methods. The convolution operation was performed on an 1Kx1K image using a 15x15 window. The entries in the first row of the table display execution times of the parallel convolution algorithm using the Farmer-Worker pattern. The workstations used in parallelization were of the same speed factors.

We then reduced the speed of one workstation, by executing a computation-intensive non-terminating external process. We implemented the parallel convolution algorithm using a different number of workstations, but always included the workstation executing the external process. The effective speed factor of the workstation executing an external process was nearly halved, since it was time-shared to execute an external process and a worker component of the Farmer-Worker pattern. The entries in the second row of Table 4.3 show results of this parallelization. The performance results are similar to the previous results (i.e. entries in first row of the table) since, most of the subimages are now processed by other workstations. There is not much reduction in the overall performance as can be seen from Figure 4.6.

However, if we partition the image into several different subimages of the same size and, if the number of subimages created are equal to the number of workstations used in parallelization, the performance of the parallel convolution operation degrades significantly. This can be seen from the entries in the third row of Table 4.3. The execution time is dominated by the slow workstation executing an external process. Since the slow workstation has the same workload as the other workstations, it takes more time to process its subimage. This reduces the overall performance in the parallel convolution algorithm. Hence, the Farmer-Worker pattern which has an inherent dynamic load balancing property, can be used to achieve improved performance over the conventional methods used for parallelizing an application.

4.4.2 Rank filtering

Rank Filters are non-linear filters which are used for reducing the variance in an image. They eliminate salt-and-pepper noise but unlike the linear filters they preserve the sharp
edges. A rank filter transforms an image by changing each pixel value to a specified value in the neighborhood of that pixel point. If \( \mathcal{N} \) represents a set of pixel values in the neighborhood of some pixel point \((i,j)\) and if the elements in \( \mathcal{N} \) are sorted in ascending order, then a rank filter \( R_i \) of \( i \)th order assigns the \( i \)th element in \( \mathcal{N} \). Three special rank filters are the \( R_{\min} \), \( R_{\max} \) and \( R_{\text{median}} \), which respectively assign minimum, maximum and median pixel values to the pixel point \((i,j)\). A review of the rank filters and their properties is given in (Hodgson et al., 1985).

Rank filters can be parallelized using the Farmer-Worker pattern. The execution times for the rank filtering operation parallelized using different number of workstations are displayed in Table 4.4. The performance results of the rank filtering operation are similar to the performance results of the convolution operation.

Table 4.4: Execution time in (min:sec) for the rank filtering operation

<table>
<thead>
<tr>
<th>Image Size</th>
<th>Window Size</th>
<th>Number of Workstations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>128x128</td>
<td>3x3</td>
<td>0:01</td>
</tr>
<tr>
<td></td>
<td>11x11</td>
<td>0:09</td>
</tr>
<tr>
<td>256x256</td>
<td>3x3</td>
<td>0:02</td>
</tr>
<tr>
<td></td>
<td>11x11</td>
<td>0:33</td>
</tr>
<tr>
<td>512x512</td>
<td>3x3</td>
<td>0:08</td>
</tr>
<tr>
<td></td>
<td>11x11</td>
<td>2:14</td>
</tr>
<tr>
<td>1Kx1K</td>
<td>3x3</td>
<td>0:30</td>
</tr>
<tr>
<td></td>
<td>11x11</td>
<td>9:14</td>
</tr>
</tbody>
</table>

### 4.4.3 Spatial filters

A combination of \( R_{\min} \) and \( R_{\max} \) rank filters forms a family of spatial filters. Spatial filters can be used as approximations to the true low-pass and high-pass filters. A spatial low-pass filter, for example, can be defined as \( R_{\min}^n R_{\max}^n (\mathcal{F}) \), where \( R_{\min}^n (\mathcal{F}) \) (or \( R_{\max}^n (\mathcal{F}) \)) denotes applying \( R_{\min} \) (or \( R_{\max} \)) \( n \) times to the image \( \mathcal{F} \). The cut-off frequency is determined by \( n \), the larger the value of \( n \), the lower is the value of cut-off frequency. Other definitions of spatial low-pass filters can be found in (Hussain, 1991). A high-pass
filtered image is obtained by subtracting the original image $F$ from the low-pass filtered image $R_{\min}^n R_{\max}^n (F)$. A high-pass filter sharpens details in an image.

Table 4.5: Execution time in (min:sec) for the sharpening operation

<table>
<thead>
<tr>
<th>Image Size</th>
<th>Window Size</th>
<th>Number of Workstations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1 2 4 6 8 10 12 14 16</td>
</tr>
<tr>
<td>128x128</td>
<td>3x3</td>
<td>0:04 0:02 0:01 0:02 0:02 0:02 0:02 0:02 0:02</td>
</tr>
<tr>
<td></td>
<td>11x11</td>
<td>1:09 0:35 0:20 0:16 0:14 0:13 0:11 0:10 0:08</td>
</tr>
<tr>
<td>256x256</td>
<td>3x3</td>
<td>0:15 0:09 0:05 0:05 0:03 0:03 0:03 0:03 0:02</td>
</tr>
<tr>
<td></td>
<td>11x11</td>
<td>4:51 2:27 1:23 1:07 0:54 0:53 0:46 0:41 0:34</td>
</tr>
<tr>
<td>512x512</td>
<td>3x3</td>
<td>0:59 0:30 0:15 0:11 0:09 0:08 0:07 0:07 0:06</td>
</tr>
</tbody>
</table>

Figure 4.7: Performance of the sharpening operation using spatial filters (window size 11x11)

Spatial filters are iterative operations, hence, they can be parallelized using the Master-Worker pattern. The execution times for the sharpening operation (high pass filtering) parallelized using different number of workstations are displayed in Table 4.5. A plot of these execution times and the speedups achieved for this operation are shown in Figure 4.7. The low-pass filtering was performed with a value of $n$, equal to 5.
Since a low-pass filtering of the image is involved in this operation, there is a need for communicating the boundary information after each iteration. Each iteration involves a rank filtering operation on all the subimages. However, the time required for performing the rank filtering operation on each subimage is much higher than the time required to exchange the boundary information. Therefore, the time spent in worker-worker communications does not appear to make a significant degradation in the overall performance.

4.5 Fast Fourier transforms

A two-dimensional fast Fourier transform (2D-FFT) of an image is a global algorithm in which the value of each pixel depends on the values of all pixels in the image. The Fourier transform of an image enables image filtering in the frequency domain. A two-dimensional fast Fourier transform $F(u,v)$ of an image $f(x,y)$ is given by (Sonka et al., 1993)

$$F(u,v) = \frac{1}{MN} \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} f(m,n) e^{-2\pi i \left( \frac{mu}{M} + \frac{nv}{N} \right)}$$ (4.3)

and

$$= \frac{1}{M} \sum_{m=0}^{M-1} \left[ \frac{1}{N} \sum_{n=0}^{N-1} \exp\left( \frac{-2\pi inv}{N} \right) f(m,n) \right] \exp\left( \frac{-2\pi imu}{M} \right)$$ (4.4)

where, $u = 0,1,\ldots,M-1$, $v = 0,1,\ldots,N-1$, and $i = \sqrt{-1}$. A 2D-FFT is separable and therefore can be expressed as two one-dimensional fast Fourier transforms: a one-dimensional FFT along the rows followed by a one-dimensional FFT of the intermediate results along the columns, or vice versa. The term in square brackets in equation 4.4, for example, corresponds to the one-dimensional Fourier transform of the $m$th row.

A 2D-FFT can be parallelized by computing the one-dimensional FFT along the rows (or columns), transposing the intermediate results, and finally computing a one-dimensional FFT along the columns (or rows) (Choudhary & Patel, 1990). We use the Controller-Worker pattern to implement this form of parallelism. Each processor or workstation is assigned a set of contiguous rows of the input image. The number of rows assigned to each processor is proportional to its speed factors. Each processor computes
the 1D-FFT along its rows (using 1D-FFT algorithm given in (Press et al., 1992)). The processors then communicate with each other to transpose the intermediate results (the row FFTs) as shown in Figure 4.8.

![Figure 4.8: The data blocks needed to transpose the intermediate results](image)

Each processor needs to communicate and exchange a block with every other processor. A pair of data blocks exchanged by any two processors are shown with the same shades/patterns (Figure 4.8). After exchanging the row FFTs as specified above, each processor computes the 1D-FFT along the columns. Finally, each processor sends the computed results to the Controller.

<table>
<thead>
<tr>
<th>Image Size</th>
<th>Number of Workstations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>256x256</td>
<td>0:01</td>
</tr>
<tr>
<td>512x512</td>
<td>0:04</td>
</tr>
<tr>
<td>1Kx1K</td>
<td>0:20</td>
</tr>
</tbody>
</table>

The execution times for the FFT operation parallelized using different number of workstations are displayed in Table 4.6. From Table 4.6 we can observe that the communication overheads dominate the performance of that the FFT operation. The computational time for this operation on a single workstation is of the order of few seconds. However, the time spent in all-to-all worker communications and in communicating the final results to the
controller, is relatively large compared to the time spent in the computation. Moreover, the worker-worker communications involve costly floating point exchanges. It is therefore difficult, to achieve any significant performance gains in parallelization of the 2D-FFT operation in a workstation environment.

4.6 Image restoration

4.6.1 Markov random field models for image recovery

Markov random field (MRF) models and Bayesian methods are stochastic techniques used in image restoration, image segmentation and image interpretation. In an MRF model, the problem is formulated as an optimization problem [maximum a posteriori (MAP) estimation rule] by representing the local characteristics of the image pixels by Markov random field and its associated Gibbs distribution. An iterative optimization method, such as simulated annealing, is applied to generate a sequence of images which converge in an appropriate sense to the optimal MAP estimate. The algorithms based on this stochastic technique are computationally intensive and highly parallel. The algorithm used for image restoration is presented in a nut-shell. A detailed discussion and various other algorithms based on this technique are presented in (Mardia & Kanji, 1993).

If $f$ is the observed image and if $\Omega$ denotes the set of all possible interpretations of $f$, then the MAP estimate of $f$ is the one which maximizes the probability of the interpretation $g$ given the observed image $f$, i.e. we seek

$$\max_{\omega \in \Omega} [P(g = \omega | f)] \quad (4.5)$$

After rigorous mathematical analysis and simplification this ultimately leads to minimizing of an energy function which is given by (Buxton et al., 1986)

$$\sum_{(a,b)} \left( \sum_{(i,j)} F[f(a, b), f(i, j)] \right) + \sum_{(a,b)} (f(a, b) - g(a, b))^2 / 2\sigma^2 \quad (4.6)$$
where, \((a, b)\) is any point in the image and \((i, j) \in \mathcal{N}\), which is a set of neighboring points around the point \((a, b)\). The parameter \(\sigma\) denotes the standard deviation of the additive Gaussian noise (with zero mean) in the degraded image. The real-valued function \(F[f(a, b), f(i, j)]\) adds a value to the energy function which is inversely proportional to the degree of similarity between the pixel intensities of the image points \((a, b)\) and \((i, j)\).

The energy function given by equation 4.6 is minimized using simulated annealing process which is described below (Buxton et al., 1986), (Kapoor et al., 1994).

1. Initialize starting temperature \(T\)

2. For each point \((a, b)\) in the image do
   - compute energy at point \((a, b)\)
   - generate trial pixel value and using this value, compute trial energy at \((a, b)\).
     Compute change in energy \(\Delta E = \text{trial energy} - \text{energy}\)
   - if \((\Delta < 0)\) then accept the state change i.e. assign the trial value to point \((a, b)\)
     otherwise assign this trial value to the point \((a, b)\) only when \(\exp(-\Delta E/T) > \text{random}[0, 1]\)

3. Repeat step 2 \(N_{\text{inner}}\) times

4. Lower the temperature to \(C/\log(k+1)\) where \(k\) is the total number of iteration cycles (complete raster scans of the image) and \(C\) is a constant, independent of \(k\)

5. Repeat step 2 to step 4 \(N_{\text{outer}}\) times

Table 4.7: Execution time in \((\text{min}:\text{sec})\) for image restoration using MRF model

<table>
<thead>
<tr>
<th>Image Size</th>
<th>Number of Workstations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>128x128</td>
<td>0:35</td>
</tr>
<tr>
<td>256x256</td>
<td>2:23</td>
</tr>
<tr>
<td>512x512</td>
<td>10:19</td>
</tr>
</tbody>
</table>

The MRF algorithm can be parallelized by using the Master-Worker pattern. The execution times for the MRF algorithm parallelized using different number of workstations
are displayed in Table 4.7. A plot of these execution times and the speedups achieved for this operation are shown in Figure 4.10. The algorithm was executed with the values of $N_{inner} = N_{outer} = 10$. The MRF algorithm is communication intensive. The workers communicate the boundary information after each raster scan of their assigned subimages. The total number of worker-worker communications in this example is therefore very high (each worker communicating 100 times with every other worker). But since the computing time between successive communications within each worker is relatively larger, the observed speedups are quite close to the ideal speedups.

Note that, unlike the Farmer-Worker pattern, an application parallelized using the Master-Worker pattern does not have inherent load balancing property. This can result in serious load imbalances when the pattern is implemented on an enterprise cluster (section 2.5.3). Each worker component in the Master-Worker pattern depends on other workers to perform the computations on its assigned subtask (subimage). A machine executing a worker component of the Master-Worker pattern, can delay the processing in other worker components when it is also time-shared to run external processes. This can lead to significant reduction in the overall performance of the corresponding application that is parallelized using this pattern.
We can observe the effect of executing an external process (external load) on the performance of a Master-Worker pattern, by conducting a simple experiment. As an example of an application, we parallelize the image restoration operation based on MRF model, using the Master-Worker pattern. The performance results of the parallel implementation, using a 512x512 image, are shown in Table 4.8. The entries in the first row of the table display execution times without any external load or processes on the machines executing the pattern. The amount of work distributed to all the worker components is proportional to the effective speed factors of their corresponding machines. The entries in the second row display execution times when one of the machines is time-shared to run an external process, during the execution of a worker component of the Master-Worker pattern. The effective speed factor of such a machine is therefore halved with respect to the rest of the machines. Hence, the corresponding worker component takes longer time to perform its computations. All workers in the Master-Worker pattern exchange intermediate results with their neighbors after every iteration. The presence of a slow worker component therefore results in increased waiting time for the remaining worker components for exchanging their intermediate results. This reduces the overall performance in the application as can be seen from the entries in the second row of Table 4.8.

A potential solution to overcome the load imbalancing problem in the Master-Worker pattern is to dynamically schedule the worker components after every fixed number of iterations. However, the time required to schedule the worker components on other idle machines should be significantly lower than the overall computation time of the application.

Table 4.8: Performance of the Master-Worker pattern when subjected to the external load. The execution times (min:sec) displayed are for the image restoration operation using the MRF model on a 512x512 image.

<table>
<thead>
<tr>
<th>Row No.</th>
<th>External Load (Y/N)</th>
<th>Number of Workstations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1 (c)</td>
<td>N</td>
<td>10:19</td>
</tr>
</tbody>
</table>
Chapter 4. Low level algorithms

11.1 -

- no external load

24 6 8 10 12 14 16
16-1
14-12
10-8
6-4
2

0 2 4 6 8 10 12 14 16

Figure 4.10: Performance of the Master-Worker pattern (in image recovery operation using the MRF model on a 512x512 image) subject to the external load and load distribution

4.7 Summary

In this chapter we have presented parallel implementations of some representative low level vision algorithms on a cluster of workstations. Each algorithm has been parallelized using appropriate design patterns such as Farmer-Worker, Master-Worker, and Controller-Worker. The algorithms which have been parallelized include histogram equalization, convolution, rank filtering, image sharpening using spatial filters, 2D-FFT (of an image), and image restoration using MRF models. Some of these algorithms parallelized using the Controller-Worker pattern (e.g. histogram equalization and 2D-FFT) do not result in any significant speedups. This is because the time spent in all-to-all worker communications in the Controller-Worker pattern is relatively high compared to the time spent in the actual computation. These algorithms therefore do not represent ideal candidates for parallel implementation on workstation clusters.

Parallel implementations of other low level algorithms have however shown promising results. The convolution and rank filtering operations parallelized using the Farmer-Worker pattern have resulted in significant performance gains. We have also illustrated the advantage of using a Farmer-Worker pattern to achieve improved performance over the conventional methods of parallelizing these algorithms. The image sharpening and image
restoration algorithms representing synchronous form of parallelism have been parallelized using the Master-Worker pattern. Although these algorithms are communication intensive, the computing time between successive communications at each worker is relatively high. The observed speedups in these algorithms are therefore reasonably close to the ideal speedups. Finally, we also illustrated the problem of load imbalances that can occur in the Master-Worker pattern when implemented on enterprise clusters (section 2.5.3). These load imbalances, caused due to some external processes, can lead to a significant reduction in the overall performance of the corresponding algorithm parallelized using this pattern.
Chapter 5

Intermediate level processing

In this chapter we discuss parallelization of some representative intermediate level algorithms in computer vision. Intermediate level processing forms a bridge between the low level and the high level processing operations in computer vision. It comprises algorithms which reduce the visual information produced by the low level operations to a form suitable for the recognition step in high level processing. The basic unit of information processed by these algorithms is a token which can either represent a line, an intensity, color or texture based region, or a surface. The processing step involves grouping these tokens into generic entities such as sets of parallel lines, rectangles or polygons, homogeneous and contiguous regions, or plane surfaces. Hence, the operations involved at the intermediate level processing are mainly partitioning and merging which transfer the tokens into more useful and meaningful structures for further processing.

However, unlike in low level processing, the operations or computations at the intermediate level processing are not very regular. The form of parallelism in the algorithms at this level is therefore not immediately evident. For example, even the most sophisticated low level algorithms for detecting edges and lines in an image can generate a significant number of line fragments across the image. A grouping algorithm used for linking and reorganizing the line fragments into meaningful structures may need to match and merge fragments of lines across large fractions of the image. In the parallel implementation, this may lead to a large amount of non-local and irregular communication patterns between
a significant number of processors. Hence, developing parallel solutions for intermediate level algorithms is relatively difficult. During past several years, many parallel algorithms for the intermediate operations have been suggested and are constantly being improved. However, most of these algorithms have been designed for specific class of parallel architectures (Chaudhary & Aggarwal, 1990). In this chapter, we discuss parallelization of some representative intermediate level algorithms on coarse-grained machines, such as a cluster of workstations.

Segmentation is one of the most important intermediate level operations in computer vision. It involves extraction of features or objects from an image which are used in subsequent processing, namely, object description and recognition. The main objective of segmentation is to partition the image into meaningful regions which constitute a part or whole of the objects in an image. There are two main approaches to segmentation, namely, *region-based* and *edge or pixel-based* (Gonzalez & Woods, 1993), (Awcock & Thomas, 1995). Region-based segmentation aims at creating homogeneous regions by grouping together pixels which share common features. Pixel-based segmentation aims to detect and enhance edges in an image, and then link them to create a boundary which encloses a region of uniformity. Region-based segmentation is identified as a *similarity* method since the image regions require some similarity criterion for creation. In contrast, pixel-based segmentation is termed as *discontinuity* method since the creation of regions involves detection of edges that are abrupt discontinuities in pixel grey-level values.

This chapter is organized as follows. In section 5.1 we discuss parallelization of a region-based segmentation algorithm. In section 5.3 we discuss parallel implementation of a perceptual grouping algorithm used for grouping line tokens into meaningful entities such as straight lines, junctions, and rectangles or polygons. Perceptual grouping algorithms constitute pixel-based approach to image segmentation. In each of these sections, we present a sequential algorithm followed by its corresponding parallel implementation. These implementations have been designed and developed for parallel execution on a cluster of workstations, using the relevant design patterns.
5.1 Region based segmentation

Region-based segmentation can be formally defined as follows (Gonzalez & Woods, 1993). A region $R$ of an image $I$ is defined as a connected homogeneous subset of the image with respect to some ‘similarity criterion’ such as gray tone, or texture. Let $P$ denote a logical predicate which assigns the value true(1) or false(0) to $R$, depending only on the properties related to the pixels in $R$. For example, $P(R) = \text{true}$, if the difference between the maximum and minimum pixel value in $R$ is less than some threshold. A region-based segmentation of an image is a partition of $I$ into several homogeneous regions $R_i, i = 1, 2, ...n$ such that

\[ \bigcup_{i=1}^{n} R_i = I \]  \tag{5.1}

\[ R_i \cap R_j = \emptyset \quad \text{for all } i \text{ and } j, \ i \neq j \]  \tag{5.2}

\[ P(R_i) = 1 \quad \text{for } i = 1, 2, ..., n \]  \tag{5.3}

\[ P(R_i \cup R_j) = 0 \quad \text{for } i \neq j \]  \tag{5.4}

Condition (5.1) indicates that every pixel must be in a region. Condition (5.2) indicates that the regions must be disjoint (their intersection must be \emptyset, an empty set). Condition (5.3) deals with the properties that must be satisfied by the pixels in the regions. Finally, condition (5.4) indicates that the adjacent regions $R_i$ and $R_j$ are different in the sense of predicate $P$.

The Region-based segmentation algorithms can be classified into three categories:

1. **Region growing**: In region growing an image is divided into an arbitrary number of elementary regions, often starting at the level of individual pixels. These elementary regions are then merged to form larger regions on the basis of certain homogeneity criterion. The region growing algorithm starts with an image partition that satisfies condition (5.3) and proceeds to fulfill condition (5.4). The merging process terminates when no two adjacent regions are similar.

2. **Region splitting**: In contrast, region splitting views the entire image as a single region.
Each region is then recursively subdivided into smaller subregions, if the region is not homogeneous enough. The processing starts in a condition satisfying (5.4) and proceeds to fulfill condition (5.3). The measure for homogeneity is similar to that used in region growing.

3. Region splitting and merging: This scheme combines both the split and merge operations in one algorithm (Horowitz & Pavlidis, 1974), in order to exhibit advantages of both the methods. The image is initially subdivided into an arbitrary set of disjoint regions which are then merged and/or split in an attempt to satisfy the conditions stated in equations 5.1-5.4. A split and merge algorithm begins with satisfying neither of the two conditions (5.3) and (5.4) and ends up with satisfying both (5.3) and (5.4).

A simple realization of the split and merge technique is to represent the entire image as one region, initially, and then recursively divide a region into smaller and smaller quadrant regions in a quadtree fashion (Figure 5.1) such that for any region $R_i$, $P(R_i) = false$ (Gonzalez & Woods, 1993). Also, merge the adjacent regions $R_i$ and $R_j$ for which $P(R_i \cup R_j) = true$. The algorithm stops when no further splitting or merging is possible. The root of the tree in Figure 5.1 corresponds to the entire image while the leaves of the tree correspond to individual pixels. Each intermediate node corresponds to a subdivision.

![Figure 5.1: a) Partitioned image b) Corresponding quadtree](image)
5.2 Parallel Region-based segmentation

Region-based segmentation can be computationally expensive in images of complex scenes. Hence, recent work in region-based segmentation has concentrated mainly on developing efficient parallel algorithms (Copty et al., 1989), (Choudhary & Thakur, 1994), (Hambrusch et al., 1994), (Alnuweiri & Prasanna, 1992), (Willebeek-LeMair & Reeves, 1990), (Haralick & Shapiro, 1985). The effectiveness of a particular algorithm depends on the application area, input image, and the type of parallel architecture. In this section, we focus on experimental evaluation of the parallel split and merge segmentation algorithm applied on gray-scale images and implemented on coarse-grained machines, such as a cluster of workstations.

The region-based split and merge segmentation algorithm is well suited for parallel implementation using the divide and conquer principle. Divide and conquer algorithms (Stout, 1987) use a recursive strategy to split a problem into smaller subproblems and merge the solutions to these subproblems into the final solution. Divide and conquer strategies appear to provide a natural and efficient parallel solution to many problems on coarse-grained machines. Several divide and conquer algorithms have been proposed for image processing (Chaudhary & Aggarwal, 1991), (Stout, 1987), (Sunwoo et al., 1987).

The first phase in the parallel split and merge segmentation algorithm involves splitting the image into several subimages such that each processor or workstation has its own subimage associated with it. We describe the splitting and distribution process later. In the next phase, each workstation applies a sequential region growing algorithm to segment its associated subimage. The region growing algorithm defines individual pixels as initial elementary regions. It then adds adjacent pixels to a region if the difference between their grey values and the average pixel value of the current pixels in the region is less than a threshold. After completing the segmentation process, the final phase involves merging of the segmented subimages at the boundaries of subdivision. The merging process occurs in phases, in a binary tree fashion as shown in Figure 5.2 (b), and takes \( \log P \) steps for a given \( P \) number of processors. The segmented regions of the entire image are in the root processor after the merging step.
Chapter 5. Intermediate level processing

Merging of the segmented subimages is performed at the boundary of subdivision. While merging along any boundary, the intensity values of the two neighboring pixels at the boundary are compared. If they satisfy the homogeneity criterion (the difference in their values is less than some threshold) the two regions across the boundary are merged. The value of each pixel in the merged region is set to the average of all the pixel values within this region. If the values of two neighboring pixels are same or do not satisfy the homogeneity criterion the regions are kept unchanged.

The splitting and distribution of the subimages is the inverse of merging process performed at different levels of the binary tree. The processor at the root of the binary tree divides the image into two subimages and sends them across two processors at the lower level. Each intermediate processor in the tree subdivides its assigned subimage into three parts (Figure 5.2 (a)). It retains one part to itself (for segmenting), and sends other two parts to its left and right siblings in the binary tree. If there is no right sibling (as in node 3, Figure 5.2), the subimage is subdivided only in two parts. The leaf processors do not perform any subdivision on their assigned subimage. At the end of the splitting and distribution process, each processor (workstation) has an associated subimage. The size of this subimage is proportional to the speed factor of the underlying workstation.

Figure 5.2: a) Distribution of subimages b) Merging of subimages
Chapter 5. Intermediate level processing

The split and merge segmentation algorithm can be parallelized using the Divide-and-Conquer (DC) pattern (section 3.6). The execution times for the parallel segmentation algorithm implemented on varying number of workstations are displayed in Table 5.1. A plot of these execution times and the speedups achieved for this operation are shown in Figure 5.3. The value of the threshold used for adding adjacent pixels to the regions in the corresponding subimages, was 15. From Figure 5.1 it can be seen that although the execution time of the parallel segmentation algorithm initially decreases, it does not show any significant improvement when the number of workstations used in parallelization are increased beyond six. The corresponding speedup curves show similar behavior. The drop in the scalability of the parallel segmentation algorithm is due to the time complexity of the merging processes.

Table 5.1: Execution time in (min:sec) for the parallel split and merge segmentation algorithm

<table>
<thead>
<tr>
<th>Image Size</th>
<th>No. of Regions</th>
<th>Number of Workstations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>256x256</td>
<td>1385</td>
<td>2:12</td>
</tr>
<tr>
<td>512x512</td>
<td>2023</td>
<td>3:10</td>
</tr>
<tr>
<td>1Kx1K</td>
<td>2423</td>
<td>4:09</td>
</tr>
</tbody>
</table>

Figure 5.3: Performance of the parallel split and merge segmentation algorithm
Table 5.2 displays execution times for performing various operations in the parallel segmentation algorithm applied on a 512X512 image. The percentage figures for each operation in a column are computed with respect to the total parallel execution time (displayed in last row of the column) required to segment an image on a given number of workstations. The experimental results presented in this table show that the time spent in the merging operation increases with the increase in number of workstations used in parallelization. In certain cases, it exceeds the total time required for segmenting the individual subimages. The influence of the communication time on the overall performance of the parallel segmentation algorithm is relatively insignificant as can be seen from the percentage figures of the corresponding execution times displayed in Table 5.2.

Table 5.2: Execution time in (min:sec) for various operations in the parallel split and merge segmentation algorithm applied on a 512X512 image

<table>
<thead>
<tr>
<th>Operation</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
<th>10</th>
<th>12</th>
<th>14</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Region Growing</td>
<td>1:46</td>
<td>1:07</td>
<td>0:42</td>
<td>0:36</td>
<td>0:30</td>
<td>0:25</td>
<td>0:25</td>
<td>0:20</td>
</tr>
<tr>
<td></td>
<td>(92.2%)</td>
<td>(77.0%)</td>
<td>(67.7%)</td>
<td>(59.0%)</td>
<td>(51.7%)</td>
<td>(51.0%)</td>
<td>(50.0%)</td>
<td>(40.0%)</td>
</tr>
<tr>
<td>Merging</td>
<td>0:08</td>
<td>0:19</td>
<td>0:19</td>
<td>0:23</td>
<td>0:26</td>
<td>0:22</td>
<td>0:22</td>
<td>0:27</td>
</tr>
<tr>
<td></td>
<td>(7.0%)</td>
<td>(21.8%)</td>
<td>(30.7%)</td>
<td>(37.7%)</td>
<td>(44.8%)</td>
<td>(44.9%)</td>
<td>(44.0%)</td>
<td>(54.0%)</td>
</tr>
<tr>
<td>Communication</td>
<td>0:01</td>
<td>0:01</td>
<td>0:01</td>
<td>0:02</td>
<td>0:02</td>
<td>0:02</td>
<td>0:03</td>
<td>0:03</td>
</tr>
<tr>
<td></td>
<td>(0.8%)</td>
<td>(1.2%)</td>
<td>(1.6%)</td>
<td>(3.3%)</td>
<td>(3.5%)</td>
<td>(4.1%)</td>
<td>(6.0%)</td>
<td>(6.0%)</td>
</tr>
<tr>
<td>Total time</td>
<td>1:55</td>
<td>1:27</td>
<td>1:02</td>
<td>1:01</td>
<td>0:58</td>
<td>0:49</td>
<td>0:50</td>
<td>0:50</td>
</tr>
</tbody>
</table>

Hence, if communication time is not a dominant factor, the performance of a parallel algorithm implemented using a Divide-and-Conquer pattern is mainly influenced by the time complexity of the merge operation. Note that the regions produced by a parallel segmentation algorithm may sometimes be different from those produced by an equivalent sequential algorithm due to different starting pixel points. This can happen when the contrast between the regions in the image is low. The majority of the previous implementations of the parallel segmentation algorithms have either used binary images or grey-level images containing artificial regions which have a high degree of contrast between each other.
5.3 Segmentation using Perceptual Organization

An edge or pixel based segmentation involves detection of edge points representing discontinuities in pixel intensities in an image, and linking these edge points into chains of contiguous curves. However, this method often results in a fragmented segmentation in which the curves produced do not correspond to complete object boundaries in images of complex environments. There are two approaches that have been proposed to deal with this problem. One is suitable for applications in restricted domain and makes use of model-based techniques (Chin & Dyer, 1986). Model-based techniques rely on the prior knowledge of the objects in a scene, and predict their appearance in the low level descriptions that can be extracted from the fragmented segmentation. Other approach which has become popular in recent years and which appears promising even in complex environments is that of perceptual organization (Lowe, 1985).

Perceptual organization hierarchically organizes low level image features to higher level structures such as edge points to lines, lines to parallels, rectangles and polygons, and, rectangles and polygons to the object descriptions. Perceptual organization is formally defined as the ability of the human visual system to derive relevant groupings or structures from the input images without any prior knowledge about their contents (Lowe, 1985). The grouping process follows the laws of perceptual grouping such as proximity (closer elements are grouped together), similarity (similar elements are grouped together), continuation (elements lying on a common line or curve are grouped together), closure (curves tend to be completed to enclose a region), and symmetry (elements symmetric about some axis are grouped together). The human visual system is very good at detecting geometric relationships such as collinearity, parallelism, connectivity, and repetitive patterns in an otherwise randomly distributed set of image elements, and it can usually see shapes in arrangements of poor machine-generated edge outputs of even complex scenes (Lowe, 1985).

Perceptual organization has recently been applied to solve a number of practical computer vision problems. It has proved to be effective for extraction of straight lines (Boldt et al., 1989), extraction of curves (Dolan & Weiss, 1993), detection of buildings in aerial images (Huertas et al., 1993), (Mohan & Nevatia, 1989), searching geometric structures
in natural scene images (Reynolds & Beveridge, 1987), and detection of large man-made objects in non-urban scenes (Lu & Aggarwal, 1992). In this section, we discuss parallel implementation of the perceptual grouping steps as outlined in (Lu & Aggarwal, 1992), with specific emphasis on the line grouping process. The following section presents the sequential line grouping process as described in (Boldt et al., 1989), (Lu & Aggarwal, 1992), while the section following it presents its parallel implementation.

5.3.1 Sequential Line grouping algorithm

The input to the line grouping process is a set of fragmented line segments which are extracted using the existing edge detection, edge linking and linear approximation techniques. The output is a set of straight lines which represent linear structures at a higher level of granularity as shown in Figure 5.4. There are several existing techniques that could be used for extracting the initial line fragments in an image. We use the techniques described in the Scerpo vision system (Lowe, 1985) to perform the edge detection and linear approximation of the edge contours by piecewise linear segments. These operations constitute a prerequisite step to the line grouping process. We describe these operations briefly for the sake of completeness.

![Figure 5.4: Line Grouping](image)

We use two algorithms based on the Laplacian of Gaussian and the Sobel edge operator
to select the initial edge locations as described in (Lowe, 1985). We convolve the image with a Laplacian of Gaussian operator and assign to each pixel in the convolved image, a gray value proportional to the absolute value of the result of the convolution. We then apply a Sobel gradient operator to the convolved image and select as edge locations only those zero crossing pixels that are above a given threshold in the Sobel gradient image. We then perform edge thinning on the resultant image and link the edge points on the basis of connectivity to form the edge contours. We use a simple recursive endpoint subdivision method to approximate the edge contours by piecewise line segments as in (Lowe, 1985). In this method, a line segment joining the endpoints of an edge contour is recursively subdivided at the point of maximum deviation. This subdivision continues and eventually returns a set consisting of one or more line segments such that the maximum deviation of any point on the edge contour from its corresponding line segment is less than some threshold value.

The line segments extracted using the techniques described above are often fragmented and do not reflect the linear structures in the image well. A post-processing method based on the principles of perceptual grouping is needed to obtain the required linear structures. The line grouping process performs a repeated grouping of lines into longer lines using the principles or relational constraints of perceptual grouping. We use three basic relational constraints of perceptual grouping, namely, proximity, collinearity, and continuation, to implement the line grouping algorithm. The details of other finer constraints are given in (Boldt et al., 1989). Consider an arbitrary ungrouped line in the image. We call such a line as base line. A set of previously ungrouped lines are grouped with the base line if they satisfy the following relational constraints:

- **Proximity**: The end points of the lines should fall in the neighborhood of the base line. The size and shape of the neighborhood is controlled by the corresponding parameters. Figure 5.5 (a) shows a circular neighborhood drawn at the end points of the base line.

- **Collinearity**: The lines should approximately be collinear to the base line. The difference in the orientation of the base line and any other line in its proximity should be less than a threshold (Figure 5.5 (b)).
- Continuation or Overlap: The lines within the proximity of the base line must not overlap too much. The distance between the point $Q_1$ of the base line and the projection of point $P_2$ on $l_1$ must be smaller than a threshold (Figure 5.5 (c)), where $l_2$ is any line within the proximity of the base line $l_1$.

Figure 5.5: Relational constraints in the line grouping algorithm a) proximity b) collinearity and c) continuation

The line grouping algorithm searches the neighborhoods of the end points of each base line in order to find all lines within its proximity. Each line within the proximity of the base line needs to satisfy other two conditions in order to be considered for grouping with the base line. We call the set of lines $L$ that satisfy the conditions stated above with respect to the base line $l_1$, with $l_1 \in L$, a *token group*.
After finding a token group $L$ with respect to the base line $l_1$, a representative line $l$ of $L$ is computed. Line $l$ passes through the point that is geometric center of the line segments in $L$ (Lu & Aggarwal, 1992). The orientation of $l$ is the length-weighted average of the orientation of the lines in $L$. The endpoints of line $l$ are determined by orthogonally projecting the line segments in $L$ onto $l$. The two furthest apart projection points are the end points of $l$. The line $l$ replaces the lines in $L$ (see Figure 5.4). The line grouping process continues until no more merging is possible. It always terminates after a finite number of iterations as there are only finite number of lines in the image and their number declines in each iteration.

Note that in order to reduce the search space, a line segment is represented by two of its end points and is indexed by the image pixels corresponding to the end points (Figure 5.6). Hence, an index array of the size of the original image is constructed prior to the grouping process. When searching for lines close to a base line, the neighborhood of the end points of the base line in the index array is searched. Only those lines whose end points fall into this neighborhood are examined as shown in Figure 5.6(b).

Figure 5.6: Indexing technique used in the line grouping process. a) search area for the base line b) the index array
5.3.2 Parallel Line grouping algorithm

In the parallel implementation, we assume that the fragmented line segments or line tokens have been extracted from the input image using the existing methods of edge detection, edge linking and linear approximation. The input to the parallel perceptual grouping algorithm is therefore a set of line tokens which are communicated to each processor or workstation before starting the line grouping process. Each processor has a complete set of the token data consisting of all input tokens in the image. Each processor constructs an index array and uses it to partition the token data into a set of token groups. A load balancing procedure is then employed to assign each processor a finite number of token groups, in proportion to its corresponding speed factor. The token groups assigned to the processors are then processed in parallel. Each token group consisting of two or more line segments is replaced by a representative line to form a new token, using the merging procedure described in section 5.3.1.

After completion of the merging process, each processor communicates its tokens (those processed by it) to all other processors. Again, each processor has a complete set of new token data. This process is repeated for a fixed number of iterations or until no more tokens can be grouped and merged into representative line tokens. The parallel line grouping algorithm can be summarized as follows:

1. Broadcast token data from each workstation to every other
2. Form Token Groups at each processor
3. Assign a distinct set of token groups to each processor (for merging)
4. Perform merging of the token groups at each processor
5. Repeat step 1 to step 4 for a fixed number of iterations or until no more merging is possible

Note that the potential parallelism that can be exploited in the line grouping process is mainly during the merging process. After partitioning the input token data into token groups, the replacement of each token group by a representative line is essentially a parallel
local processes. A spatial partitioning of the index array (either horizontally or vertically) in order to parallelize the line grouping process may not always be feasible. For example, when the line segments in a token group span large portions of the image space, it is extremely difficult, if not impossible, to partition the index array spatially, in order to realize a parallel implementation. These line segments would spread themselves across several index array partitions.

The parallel line grouping algorithm presented in this section is similar to an earlier implementation proposed by Prasanna et al. (Prasanna & Wang, 1996). However, we use a different load balancing scheme which is based on the distribution of the token groups. The load balancing method used in (Prasanna & Wang, 1996) is based on the total search area of the input tokens. This method may not always lead to an even distribution of load, since, many base line tokens may span large portions of the image area and may not require grouping or merging with other line tokens (their token groups consist of only the base line). Also, the parallel line grouping algorithm presented in (Prasanna & Wang, 1996) is non-hierarchical, that is, it does not group the line tokens iteratively into higher levels of granularity.

The structure of the parallel line grouping algorithm is similar to that implemented by an iterative variant of the Controller-Worker pattern (section 3.5). Hence, the parallel line grouping algorithm can be parallelized using an iterative variant of the Controller-Worker pattern. The execution times for the parallel line grouping algorithm implemented on varying number of workstations are displayed in Table 5.3. From Table 5.3, it can be seen that the execution time of the parallel line grouping algorithm does not show any improvement over its corresponding sequential implementation.

Table 5.3: Execution time in (min:sec) for the line grouping process

<table>
<thead>
<tr>
<th>Image Size</th>
<th>No. of Tokens</th>
<th>Number of Workstations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>256x256</td>
<td>855</td>
<td>0:01</td>
</tr>
<tr>
<td>512x512</td>
<td>1454</td>
<td>0:02</td>
</tr>
<tr>
<td>1Kx1K</td>
<td>7921</td>
<td>0:07</td>
</tr>
</tbody>
</table>

The poor performance of the parallel algorithm is mainly due to inherent sequen-
Chapter 5. Intermediate level processing

tial nature of the line grouping process and the communication overheads in its parallel implementation. The only parallelism that can be exploited in this algorithm is during the merging operation, where, different token groups are replaced by their corresponding representative line tokens, concurrently. The time spent in the merging operation is however, significantly lower than the time spent in communicating the newly formed tokens between different workstations, during each iteration. Also, when the number of line tokens required to be processed increase, the communication overheads dominate the overall execution time as can be seen from the entries in the third row of Table 5.3. The communication overheads include time spent in packing and unpacking the line tokens into data packets, and the time spent in communicating these data packets between different workstations.

Nevertheless, the line grouping algorithm based on the principles of perceptual organization serves as a typical example of an intermediate-level operation in computer vision. It illustrates problems and difficulties encountered while parallelizing such algorithms, particularly on a cluster of workstations. Such algorithms are more suitable for sequential implementations in the workstation environments.

5.4 Summary

In this chapter, we have presented parallel implementations of two intermediate level vision algorithms, namely, region-based split and merge segmentation algorithm, and the line grouping algorithm based on the principles of perceptual organization. The segmentation algorithm has been parallelized using the Divide-and-Conquer (DC) pattern. The performance of this algorithm does not show a scalable improvement with increase in number of workstations used in parallelization beyond a certain limit. This is due to corresponding increase in time needed to merge the segmented subimages in the merging operation. The influence of the communication time on overall performance of the parallel segmentation algorithm is relatively insignificant. Hence, if communication time is not a dominant factor, the performance of an algorithm parallelized using a DC pattern is influenced mainly by the time complexity of the merging operation.
The line grouping algorithm has been parallelized using an iterative variant of the Controller-Worker pattern. Since this algorithm is inherently sequential in nature, the only parallelism that can be exploited in this algorithm is during the replacement of token groups by their corresponding representative line tokens. The time spent in this operation is however, significantly lower than the time spent in all-to-all worker communications in the Controller-Worker pattern. The performance of the parallel line grouping algorithm therefore, does not show any improvement over its corresponding sequential implementation. This example illustrates problems and difficulties encountered while parallelizing a typical intermediate level algorithm on coarse-grained machines, such as a cluster of workstations. It also shows limitations of the use of Controller-Worker pattern for parallelizing such applications on these machines.
Chapter 6

High level processing

In this chapter we discuss parallelization of a high level vision algorithm for object recognition using a Farmer-Worker pattern. We also discuss parallelization of an application in medical imaging using three different design patterns, namely, Temporal Multiplexing, Pipeline, and Composite Pipeline. High level processing in computer vision involves recognition of objects in a scene based on the knowledge acquired by the lower level processes from the images(s) of that scene. The tasks at this level are usually top-down or model-directed, and involve mainly symbolic and/or knowledge processing.

An example of a high level vision task is model-based object recognition. Given a database of object models, model-based object recognition involves finding instances of these objects in a given scene. A model-based vision system extracts scene features, such as edges and points from an image of a scene, and compares them with a database of object models in order to identify objects within that scene. Most model-based object recognition systems are based on hypothesizing matches between the scene and model features, predicting new matches, and verifying or changing the hypotheses through a search process (Grimson, 1990), (Lowe, 1985). The task becomes more complex if the objects are overlapped or occluded in the scene. A review and/or methods used in model-based object recognition in computer vision can be found in (Chin & Dyer, 1986) (Grimson & Huttenlocher, 1991).
Chapter 6. High level processing

In recent years, a new method based on geometric hashing has been proposed for model-based recognition of objects (Lamdan & Wolfson, 1988). This method offers a different and more parallelizable paradigm for model matching. The geometric hashing algorithm used for model matching consists of two phases: preprocessing and recognition. The preprocessing phase uses a collection of object models to build a hash table (described later) data structure. This data structure encodes the model information in a highly redundant and multiple viewpoint way. In the recognition phase, the properties of the extracted features in the scene image are used to index the hash table data structure for a possible match to candidate object models. Although geometric hashing still requires a search over the features in a scene, it obviates a search over the models and the model features. Hence, the recognition phase is computationally efficient and highly amenable to parallel implementation (Rogoutsos & Hummel, 1992).

In this chapter, we discuss parallel implementation of the recognition phase in the geometric hashing algorithm used for model matching. Section 6.1 describes the sequential algorithm for performing geometric hashing, while section 6.2 discusses its parallel implementation. We end this chapter with a section that discusses parallelization of an application in medical imaging, namely, multi-scale shape description of MR brain images in epileptic patients. We use three different approaches (based on Temporal Multiplexing, Pipeline, and Composite Pipeline patterns) to discuss parallelization of different modules in this application.

6.1 Sequential geometric hashing algorithm

We assume that the database has $M$ object models and each model is represented by $n$ feature points. The preprocessing and recognition phases of the geometric hashing algorithm work as follows:
Chapter 6. High level processing

6.1.1 Preprocessing Phase

In the preprocessing phase a hash table is created from the $M$ models in the database. For each model, two arbitrary feature points, referred as basis set, are used to define an orthogonal coordinate system as shown in Figure 6.1(a). Using this coordinate system, a new set of transformed coordinates of the remaining feature points in the model are computed using simple transformation equations in analytic geometry (Efimov, 1966). These new coordinates are then used to hash or generate entries into a hash table. Each entry in the hash table consists of a (model, basis) pair, representing the model number and the basis set. This process is repeated for all possible basis sets in a given model, and for all models in the database. As a result, the hash bins in the hash table will receive more than one entry. The final hash table contains a list of (model, basis) entries in each bin, as shown in Figure 6.1. The preprocessing procedure is executed off-line and only once. The steps in the preprocessing phase are outlined below:

1. Extract a set of $n$ feature points from a given model $m$.
2. Select as basis set, a pair of two distinct feature points $(i, j)$.
3. Compute the coordinates of the remaining feature points in the model with respect to the coordinate system defined by this basis set $(i, j)$.
4. Compute the hash bin locations using a hash function $h$ (described later) applied on the transformed coordinates in step 3.
5. Add (model, basis) pair, i.e. $(m, (i, j))$, to the list of entries in corresponding hash bin locations computed in step 4.
6. Repeat steps 2-5 for all possible basis sets in model $m$.
7. Repeat steps 1-6 for all models $m$ in the database.
6.1.2 Recognition phase

In the recognition phase, an arbitrary pair of feature points from the scene image is chosen as a basis set. The transformed coordinates of the remaining points in the scene are calculated relative to the coordinate system defined by this basis set. Each new coordinate is mapped to the hash table (the same as that in the preprocessing phase), and the entries in the corresponding bin receive a vote. The \((model, basis)\) pairs which receive sufficient votes (i.e. above a certain threshold value) are taken as potential matching candidate models. These are then passed to a verification module, which verifies the presence of matching models against the scene features.

The main goal of the voting scheme is to reduce the number of candidates used in the verification step. The execution of the recognition phase corresponding to a basis set is termed as a \textit{probe}. The steps in the recognition phase are outlined below:

1. Extract a set of \(S\) feature points from the scene.
2. Select as basis set, an arbitrary pair of feature points \((i, j)\) from \(S\).
3. Perform a *probe* using sequence of following steps:

- Compute the transformed coordinates of remaining feature points in \( S \) with respect to the coordinate system defined by this basis set \((i, j)\).
- Compute the hash bin locations in the hash table using a hash function \( h \) (described later) applied on the transformed coordinates.
- Form a list of all the \((model, basis)\) pairs stored in the corresponding hash bin locations computed in the previous step.
- Select the \((model, basis)\) pairs (winning models) receiving a count of votes above a given threshold value (if any).

4. Repeat from step 2 until some winning \((model, basis)\) pairs are found or until com-
pletion of some specified number of iterations.

5. Verify the potential models found in step 3 (if any) against the set $S$ of features in the scene.

6. Remove feature points of the matching model(s) from the scene (if applicable) and repeat steps 2-6 until some specified condition or for a fixed number of iterations.

The selection of ($model$, $basis$) pairs receiving maximum votes in step 3 may be performed by histograming (i.e. counting) these entries using corresponding ($model$, $basis$) counters. Alternatively, the ($model$, $basis$) pairs may be sorted in order to find winning models having a count above a given threshold value.

6.2 Parallel geometric hashing algorithm

In this section, we present a parallel implementation of the recognition phase of the geometric hashing algorithm. The preprocessing phase is a one time process and can be carried out off-line. The parallel implementation of the recognition phase may be realized by either a) performing the operations of a single probe across several processors, concurrently, or b) performing multiple probes on several processors, concurrently. In the latter case, each probe may in turn be implemented on a set of one or more processors. The suitability of each method depends on the size of the hash table and the amount of memory available on each processor of the underlying parallel architecture.

There have been several prior efforts in parallelizing the recognition phase of the geometric hashing algorithm. Bourdon et. al. (Bourdon & Medioni, 1988) and Rogoutsos et. al. (Rogoutsos & Hummel, 1992) have proposed parallel implementation of a probe step (method (a)) across several processors on SIMD hypercube-based machines. Their implementations employ large number of processors in proportion to the size of the model database. Wang et. al. (Wang et al., 1994) have proposed several parallel implementations of the recognition phase on CM-5 and MP-1, using both methods (a) and (b).

Each implementation uses a different strategy for distributing the hash table entries.
Chapter 6. High level processing

Each implementation uses either a histograming or a sorting method to compute the winning \((model, basis)\) pairs receiving the maximum number of votes during the recognition phase. Their implementations are independent of the size of the model database and achieve improved performance over earlier efforts. They have achieved a single probe time of about 200 milliseconds on a 32-node CM-5 connection machine, while Rogoutsos et al. (Rogoutsos & Hummel, 1992) have reported a single probe time of 1.52 sec on a 8K processor connection machine. Both have used a synthesized model database containing 1024 models (each model consisting of 16 feature points or dot patterns) and a scene consisting of approximately 200-256 feature points.

Due to limited local memory on individual processors, all the above implementations involve distribution of the hash table entries across several processors and, parallelizing the operations of a single probe on these processors. In workstation environment, the time required for performing a probe step on a single workstation is of the order 1.2 to 1.3 secs. Parallelizing the operations of a single probe across several workstations as in previous approaches would not lead to any significant improvement in the performance due to high communication costs. Parallelizing a probe step involves computing local \((model, basis)\) winning pairs and communicating these winning pairs between different processors in order to find the global \((model, basis)\) winning pairs. Since an object requires around 100-250 probes for recognition (Rogoutsos & Hummel, 1992), we perform multiple probes on various workstations, concurrently. The operations of each probe are however performed on a single workstation.

We now discuss the actual parallelization of the recognition phase on a cluster of workstations. As in (Rogoutsos & Hummel, 1992), we use a synthesized model database containing 1024 models. Each model consists of 16 randomly generated feature points (dot patterns). These model pairs are generated using a Gaussian distribution with zero mean and unit standard deviation. Similarly, we construct a scene consisting of 200 scene points using a normal distribution. In order to make the recognition process as efficient as possible, we apply two enhancements as mentioned in (Rogoutsos & Hummel, 1992). Firstly, we apply a rehashing function to the transformed coordinates (step 3 of the recognition phase described in section 6.1.2) so that the expected list lengths of the entries in the hash bins become as even as possible. For each transformed coordinate \((u, v)\), the
following hash function is applied:

\[ f(u,v) = (1 - \exp \frac{u^2 + v^2}{2\sigma^2}, \tan2(v, u)) \] (6.1)

where, \( \sigma \) represents the standard deviation of the model points. The values of the two coordinates in equation 6.1 lie in intervals \((0,1)\) and \((-\pi, \pi)\), respectively. These coordinate values can be quantized into a two dimensional hash array as shown in Figure 6.3(a). Each hash location contains a pointer to a list or bin of \((m, (i,j))\) entries.

Secondly, we use certain symmetries in the hash table to reduce the number of entries in the hash lists. If an entry of the form \((m, (i,j))\) hashes to a location \((x,y)\) in the hash table, then there will be a mirror-entry of the form \((m, (j,i))\) in location \((x, 99-y)\) as shown in Figure 6.3(b) - (c). We can therefore store only those \((m, (i,j))\) entries in the hash table for which \(i < j\). This will reduce the number of entries in the hash table by nearly half, thereby halving the memory required to store the hash table during the recognition phase. For such a hash table, if \(f(u,v)\) hashes to location \((x,y)\) in the probe step of the recognition phase, the entries in locations \((x,y)\) and the mirror-entries in location \((x, 99-y)\) are collected in a list, in order to compute the winning \((model, basis)\) pairs in subsequent processing. The mirror-entry of \((m, (i,j))\) is \((m, (j,i))\).

![Figure 6.3](image.jpg)

Figure 6.3: Hash table data structure a) symmetric indexing in hash table b) hash entries in normal hash table c) reduction in hash entries using symmetries
Chapter 6. **High level processing**

For a database consisting of 1024 models, with each model containing 16 feature points, the size of the normal hash table would be about 20 Mbytes, assuming 6 bytes for each \((m, (i, j))\) entry. Using the symmetries mentioned above, the size of the hash table would be reduced to 10 Mbytes. The workstations (SUN SPARCstation 5) that we used in implementing the parallel geometric hashing algorithm have 32 Mbytes of local memory. Hence, unlike in previous approaches, each processor (workstation in this implementation) can store a separate but complete copy of the hash table during the parallel execution of the recognition phase. Note that although we have used a synthesized model database, the size of this database is nearly the same as the size of a typical model database used in the state-of-the-art image understanding techniques employing geometric hashing (Wang et al., 1994).

The algorithm for performing multiple probes in the recognition phase can easily be parallelized by using the Farmer-Worker pattern. Each worker workstation in the Farmer-Worker pattern has a copy of the hash table and a set of scene features in its local memory, prior to the start of the recognition phase. These are loaded from a file created during the preprocessing phase. The Farmer generates arbitrary basis sets, and assigns each to a different worker for processing. Each worker performs corresponding probe step using its assigned basis set. Each worker communicates the winning \((model, basis)\) pair(s) (if any) to the Farmer controlling the whole process. When no winning \((model, basis)\) are found, each worker is assigned another basis set to perform a new probe. This processes continues until winning \((model, basis)\) pairs are found or for a fixed number of iterations. The algorithm is outlined below:

1. Generate basis sets and assign each to a different workstation (worker).

2. Perform probe step using the assigned basis set on each worker.

3. Select the \((model, basis)\) pairs that receives a count of votes above certain threshold value (if any). If no such \((model, basis)\) pairs exist, repeat the procedure from step 1 for a certain number of iterations or until some specified condition.

4. Verify the potential models found in step 3 (if any) against the set \(S\) of features in the scene.
5. Remove the feature points of matching model(s) (if applicable) from the scene and repeat steps 1-5 until some specified condition.

Table 6.1: Execution time in (min:sec) for the geometric hashing algorithm

<table>
<thead>
<tr>
<th>No. of Probes</th>
<th>Number of Workstations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>50</td>
<td>1:03</td>
</tr>
<tr>
<td>100</td>
<td>2:00</td>
</tr>
<tr>
<td>150</td>
<td>3:06</td>
</tr>
<tr>
<td>200</td>
<td>4:00</td>
</tr>
<tr>
<td>250</td>
<td>5:03</td>
</tr>
</tbody>
</table>

Figure 6.4: Performance of the geometric hashing algorithm for object recognition

The execution times for the recognition phase of the geometric hashing algorithm parallelized using different number of workstations are shown in Table 6.1. A plot of these execution times and the speedups achieved for this phase are shown in Figure 6.4. Since the communication time in the algorithm is negligible compared to the computation time, the observed speedups are quite close to the ideal speedups as can be seen from Figure 6.4.
For comparison, we compute the time required to perform 200 probes in previous implementations. Using a 8K processor connection machine, the time required for performing 200 probe steps is approximately 5 mins (based on 1.52 secs/probe time as reported in (Rogoutsos & Hummel, 1992)). The time required to perform the same number of probes on 32-nodes of a CM-5 connection machine would be around 38 secs (assuming minimum time of 188 millisecs/probe as reported in (Wang et al., 1994)). Using 512 processors (the maximum on CM-5) and performing multiple probes concurrently (each probe implemented on a partition of 32 processors), the time required to perform 200 probes may be reduced to 2 to 3 secs. However, the latter implementation (assuming such an implementation is possible) may need significant programming effort in order to exploit the hardware of the underlying parallel machine.

From Table 6.1, it can be seen that the time required to perform 200 probes using 16 workstations is only 23 secs. Hence, as this example illustrates, a workstation environment can provide a reasonable or sometimes even better performance compared to the conventional dedicated parallel machines. Note that the earlier implementations of the parallel geometric hashing algorithm are fine-grained. In contrast, the parallel implementation of the geometric hashing algorithm presented in this section is relatively coarse-grained.

### 6.3 Multi-scale active shape description - an application

All our previous discussions in this thesis have so far been concentrated upon parallelizing individual vision algorithms using corresponding design patterns. In this section, we discuss parallelization of complete modules, each comprising a collection of several algorithms, at an application level. We take an application from the field of medical imaging, namely, multi-scale active shape description of MR (magnetic resonance) brain images using active contour models. This application forms a part of the research work carried out in the Department of Computer Science, University College London, UK (Schnabel, 1997). We present a brief overview of this application and discuss parallelization of some of its modules.
6.3.1 An overview of the shape description process

Detecting and describing brain deformations in certain brain diseases (e.g. epilepsy) is a major task in MR imaging. A conventional method of detecting and describing these deformations is to first segment the cross-sectional images (image slices) of the brain into different regions. These regions correspond to different parts of the brain. After identifying relevant regions(s), a set of shape measurements (e.g. area, perimeter, etc.) is applied on these region(s) in order to detect and describe appropriate shape deformations. This task is performed manually by the expert clinicians. The conventional method of finding and describing shape deformations is however time consuming and tedious. It usually involves processing large volumes of volumetric brain data. Also, due to shortage of expert clinicians, it is difficult to diagnose each patient in a given time constraint. As a result, there is great demand to automate the shape description process in order to produce meaningful shape descriptions reliably and quickly.

The research work in (Schnabel, 1997) aims to automate the shape description process, and attempts to present it as a shape description tool for diagnosis. The shape description tool enables both quantitative and qualitative shape analysis at different levels of image resolution (or scale). The shape analysis process uses concepts in multi-scale image processing (Marr, 1982), (Witkin, 1983) to describe shape changes across several scales. These concepts are based on the fact that global shape features of objects in an image can be visualized at coarser levels of image resolution (higher scale). But finer shape features of these objects can be observed only at finer levels of image resolution (lower scale). The shape description tool enables description of the shape characteristics and shape changes across several different scales, starting from either end of the scale. The actual shape extraction from the image slices is performed by using active contour models. Active contour models or snakes are energy-minimizing spline contours used for image segmentation (Kass et al., 1987).

The main steps/modules in the shape description process are: a) preprocessing b) propagation c) shape focusing and d) shape analysis. The preprocessing step involves application of simple image processing techniques such as thresholding, histogram equalization, and morphological operations (opening), on each image slice in the volumetric brain data.
These operations are applied for enhancing the objects of interest in each image slice. The propagation step computes shape contours for each image slice as shown in Figure 6.5(a). The process begins by first computing a shape contour for some intermediate image slice. An intermediate image slice is the one at the center or near the center of the set of all image slices. The shape contour for the intermediate image slice is computed by applying an optimization procedure (Williams & Shah, 1992) on a given initial contour (usually a circle), superimposed on a Gaussian-blurred output of the image slice. The optimized shape contour of the intermediate image slice is then propagated to both its neighboring image slices as shown in Figure 6.5(a).

Using the optimized shape contours as initial contours (superimposed on the Gaussian outputs of corresponding image slices), the shape contours of both neighboring image slices are computed by applying the same optimization procedure. This process is repeated by propagating the shape contours to both sides of the brain volume (i.e. two image slice partitions defined by the intermediate image slice) as shown in Figure 6.5(a). At the end of the propagation process, each image slice has an associated initial shape contour which is used as an input in the subsequent shape focusing step.

The shape focusing step operates on each image slice separately. It begins with the construction of a scale-space for each image slice. A scale-space of an image slice consists of a set of images obtained by convolving the image slice by a Gaussian function using increasing values of $\sigma$, where $\sigma$ represents the scale or width of the scaling operator. Using the multi-scale active contour model (Schnabel, 1997), the shape focusing process extracts a shape of interest from various images in the image scale-space of each image slice. This is performed by propagating the initial shape contour (computed in the propagation step) through various images in the image scale-space (starting at lowest resolution or highest scale), and regularizing the active contour model's energy function with respect to the scale. The initial, intermediate, and final shape focusing results form a multi-scale shape stack as shown in Figure 6.5(b). An illustration of the shape focusing process applied on four different images (scales) in the image scale-space of an image slice is shown in Figure 6.6.

In the final shape analysis step, each multi-scale shape stack is analyzed using classic
Figure 6.5: Multi-scale shape description process a) propagation step applied on a set of five image slices b) multi-scale shape stack of an image slice computed in the shape focusing step (Figure (b) adapted from (Schnabel, 1997)).

Figure 6.6: Shape focusing performed at different scales in the image scale-space of an image slice using active contour models: (a) $\sigma = 8$ (b) $\sigma = 4$ (c) $\sigma = 2$ (d) $\sigma = 1$. Image (a) also contains the initial contour superimposed in black. All images are taken from (Schnabel, 1997).

shape descriptors in order to find the global and local changes in the shape. The shape contour at each layer or scale of the multi-scale shape stack is used to compute the mean and slope measurements for finding shape changes between the layers. These shape
contours also are stacked and visualized (volume visualization) for qualitative inspection (Figure 6.7). Also, for each scale, the corresponding shape contours across all the multi-scale shape stacks are stacked and visualized for global inspection.

6.4 Parallelization of the shape description process

In this section we discuss parallelization of some of the modules in the multi-scale shape description process applied on the volumetric brain data of epileptic patients. The task is to obtain shape descriptions of the grey matter/cortical interface of the brain in order to enable the study of its structural abnormalities (cortical dysgenesis) related to the symptoms of epilepsy. The number of image slices in the volumetric brain data involved in this application is 124 (for each patient), of which only 96 image slices contain the image of the actual grey matter. Each image slice is of the size 256X256 pixels (with slice thickness 5 mm, and pixel size 0.9375 mm²).

We discuss three different approaches for parallelizing the shape description process. Each approach uses a different design pattern, namely, Temporal Multiplexing, Pipeline or Composite Pipeline. Of the three approaches, we provide experimental results only for the first approach. For the remaining two approaches, we provide estimates of the corresponding parallel execution times. These estimates represent reasonable approximations of the corresponding parallel execution times. This is because the components in the Pipeline/Composite Pipeline implementations use existing sequential codes. Using sequential execution time of each component, it is easy to estimate the overall parallel

Figure 6.7: Visualization of the stack contours (those displayed in Figure 6.6) stacked using triangulation. Image taken from (Schnabel, 1997).
execution time in these implementations (ignoring the communication overheads). The communication overheads are however relatively negligible and can therefore be safely ignored (they involve communication of 256x256 images and/or simple data structures (e.g. contours, etc.)). Also, in all the three approaches, we do not discuss parallelization of the final shape analysis step. The shape analysis step requires data from multi-scale shape stacks of all image slices, and therefore can only be performed sequentially.

Using the sequential code developed in (Schnabel, 1997), the time required to perform the preprocessing step on each image slice is 3 secs, while the time required to perform the corresponding propagation step is 16 secs. The shape focusing step comprises a sequence of operations such as Gaussian smoothing, computing of image potentials ('Compute Potential'), and optimization. These operations are applied on each image (total 16) in the image scale-space of a given image slice. The Gaussian smoothing operation produces a smoothed image, while 'Compute Potential' operation extracts certain image features such as the magnitude and direction of the image gradient, the image curvature, and distance-transformed ridges of the gradient magnitude, from the smoothed image. The 'Compute Potential' operation stores these image features in a data structure called 'Potential', which, along with the smoothed image, is used for computing the shape contour of the image during the optimization operation.

The three shape focusing operations require average processing time of 2 secs, 7 secs, and 7 secs, respectively. Therefore the total time required to perform the shape focusing step on a single image slice is 256 secs (16*16). Hence, for a set of 96 image slices, the total sequential time required to perform the preprocessing, propagation and the shape focusing steps is 26400 secs (7 hrs, 20 mins). In order to maintain consistency with earlier discussions, we assume availability of at the most 16 workstations for the parallelization process.

6.4.1 Parallelization using Temporal Multiplexing pattern

The simplest form of parallelism that can be implemented without major changes to the existing sequential code is realized by using the Temporal Multiplexing pattern. In
this approach, we assume that the preprocessing and propagation steps are performed sequentially. We parallelize only the shape focusing step by processing the image slices on different workstations, concurrently. The sequential algorithm to perform the shape focusing process is outlined below (starting at the coarsest level of scale):

1. Generate an image in the image scale-space for the current image slice, using a Gaussian smoothing function.

2. Using the Gaussian image generated in previous step, compute image potentials (i.e. relevant image features) required for the optimization operation in the next step.

3. Taking active contour model from the previous image in the image scale-space as an initial shape contour, optimize the shape contour for the current image using fast local optimization method (Williams & Shah, 1992). Note that for the first image in the image scale-space, the initial shape contour is the one, which is computed in the propagation step.

4. Repeat steps 1-3 for all scales in the image scale-space of the current image slice.

5. Repeat the process from step 1 for all image slices, starting from the coarsest level of scale.

Since the computations of each image slice in the shape focusing step are independent of each other, they can be performed in parallel. Using a set of 16 workstations and a Temporal Multiplexing pattern to process each image slice concurrently, the observed parallel execution time required for processing all the image slices in the shape focusing step is 1656 secs (Table 6.2). The total time required to perform the preprocessing (sequential implementation), propagation (sequential implementation) and the shape focusing (parallel implementation) step is 3480 secs (58 mins). Hence, concurrent processing of the image slices in the shape focusing step leads to a significant reduction in overall application time, although we parallelized only part of the application.
6.4.2 Parallelization using Pipeline pattern

Although Temporal Multiplexing pattern may also be used for parallelizing the preprocessing step, the propagation step does not enable concurrent processing of image slices for computing the initial shape contours. In the propagation step, the output shape contour of an image slice serves as an input for the computation of the final shape contour of either one or both of its neighboring image slices (Figure 6.5(a)). In such situations, a Pipeline pattern may be used to exploit potential parallelism in an application. One possible implementation of the shape description process using a Pipeline pattern is shown in Figure 6.8. We assume that each component of the Pipeline pattern is implemented on a separate workstation.

Figure 6.8: Parallelization of the shape description process using a Pipeline pattern. The integer values denote sequential time (in seconds) required for executing corresponding components of the Pipeline pattern.

The processing in the Pipeline pattern begins by passing the intermediate image slice through the preprocessing component, followed by the adjacent image slices in either of the two image slice partitions shown in Figure 6.5(a). The preprocessing component processes a given image slice (called current image slice) and passes it to the propagation component. The propagation component optimizes shape contour of the current image slice. It stores this shape contour for using it as an input during processing of the subsequent image slice. The propagation component passes the optimized shape contour and the current image slice to the ‘Gaussian Smoothing’ component in the shape focusing step.

The shape focusing step computes a multi-scale shape stack for the current image slice as follows. The ‘Gaussian Smoothing’ component of the Pipeline pattern generates
Gaussian-blurred images of the current image slice, using decreasing values of sigma (total 16 sigma values). These Gaussian-blurred images are then sequentially passed from the ‘Compute Potential’ component to the ‘Optimization’ component. The ‘Optimization’ component optimizes the shape contour of the current Gaussian-blurred image, and stores it for using it as an input for computing shape contour of the subsequent Gaussian-blurred image. These operations in the shape focusing step are repeated for 16 different sigma values. After completion of the shape focusing step on current image slice, the resulting multi-scale shape stack of the current image slice is passed to the shape analysis step. The shape analysis step can be performed separately and is therefore enclosed in a dotted box.

A single Pipeline pattern may be used to process both image slice partitions (defined by the intermediate image slice) one after the other. Alternatively, two Pipeline patterns (Multiple Pipelines) may be used for processing each partition concurrently. We use the second approach since it reduces the overall execution time by almost half. Assuming 48th image slice as the intermediate image slice, we divide the set of 96 image slices into two partitions. Each partition contains 48 and 49 image slices, respectively (both partitions contain the intermediate image slice for propagating the initial shape contour). We estimate the time required to process image slices in larger of the two image slice partitions (i.e. the one containing 49 image slices). This estimate also represents the total time required for processing all the 96 image slices, since the smaller image slice partition can be processed concurrently along with the larger one.

The preprocessing and propagation component operations can be overlapped with the operations in the shape focusing step (except for the first image slice). We therefore concentrate on the shape focusing step. Assuming overlap of computations in three different operations of the shape focusing step, the time required to perform the shape focusing step on 49 image slices is \( 5497 \text{ secs} (2 (\text{latency}) + 7 (\text{latency}) + (7*16)*49) \). The latency terms in the expression represent execution times required for performing corresponding operations (i.e. Gaussian Smoothing and Compute Potential), for the first Gaussian-blurred image of the first image slice. The term ‘\((7*16)\)’ denotes time required for performing the ‘Optimization’ operation on all Gaussian-blurred images of the current image slice. This also represents the time required for performing shape focusing step on the current image slice (except for the first image slice), since other operations in
the shape focusing step are executed concurrently. Hence, the total time required for performing the preprocessing, propagation, and the shape focusing step is 5516 secs (1 hr, 31 mins, 56 secs). Note that the times for the preprocessing and propagation steps (as shown in Table 6.2) in the Pipeline implementation, represent execution times required to perform these steps only on the first image slice.

The total execution time of the application parallelized using two simple Pipeline patterns is relatively higher than the execution time in previous parallel implementation. This drop in performance is due to the time-complexity of the shape focusing step and inability to use additional workstations in the parallelization process. As there are only five components in a Pipeline, the two Pipeline patterns together can use only 10 workstations. The parallel implementation using a Temporal Multiplexing pattern can utilize all 16 workstations. Hence, although the percentage of the application code parallelized using a Pipeline pattern is relatively higher than in the previous approach, the inability to scale the number of workstations used in parallelization does not lead to any improvement in the overall performance of the application over earlier approach.

6.4.3 Parallelization using Composite Pipeline pattern

The limitations in both the Temporal Multiplexing and Pipeline patterns can be resolved by using a Composite Pipeline pattern. The main bottleneck in the simple Pipeline pattern is the shape focusing step, which requires parallel execution time of 112 secs (7*16) or approximately 2 mins for computing a multi-scale shape stack for each image slice (assuming overlapping of computations of individual operations in the shape focusing step). The performance or throughput in a simple Pipeline pattern depends on the speed of its slowest component. Hence, using a Temporal Multiplexing pattern at the shape focusing step, significant performance gains can be achieved in the simple Pipeline pattern. The resulting pattern constitutes a Composite Pipeline pattern as shown in Figure 6.9.

In the Composite Pipeline pattern, we implement the preprocessing and the propagation steps on a single workstation. The remaining 15 workstations can be used for parallelizing the shape focusing step using a Temporal Multiplexing pattern. The process-
Figure 6.9: Parallelization of the multi-scale shape description process using a Composite Pipeline pattern

The processing of each image slice in the Composite Pipeline pattern begins with the execution of preprocessing and the propagation steps. Each image slice that passes through the first stage (preprocessing and propagation) can immediately use a free workstation to perform the shape focusing step. This is because the time required to process each image slice in the first stage is 19 seconds. The shape focusing step requires 256 seconds (sequential time) to process each image slice. As there are 15 workstations in the second stage of the Composite Pipeline pattern, the average time required to perform the shape focusing step on each image slice is approximately 17 seconds (256/15), which is lower than the time spent in the first stage. Therefore, any image slice that passes through the first stage can use some free workstation that completes processing on its previous image slice (if applicable).

Also, with the exception of the last image slice, the operations of the shape focusing step can be completely overlapped with the operations of the preprocessing and propagation steps. The shape focusing time shown in Table 6.2 for the Composite Pipeline implementation therefore, represents time required to process only the last image slice. The time required for the preprocessing and propagation steps in this implementation is the same as that in the sequential version. Hence, the total execution time of the application parallelized using the Composite Pipeline pattern is 2080 seconds (34 minutes, 40 seconds), which represents a significant improvement over earlier approaches.

The shape description example illustrates that using simple design patterns and most of the existing sequential code, the workstation environment can offer significant benefits
Table 6.2: Execution times in \textit{seconds} for different implementations and individual steps of the shape description process

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Preprocessing</th>
<th>Propagation</th>
<th>Shape Focusing</th>
<th>Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td>288</td>
<td>1536</td>
<td>24576</td>
<td>26400</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(7 hrs, 20 mins)</td>
</tr>
<tr>
<td>Temporal Multiplexing</td>
<td>288</td>
<td>1536</td>
<td>1656</td>
<td>3480</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(58 mins)</td>
</tr>
<tr>
<td>Multiple Pipelines</td>
<td>3</td>
<td>16</td>
<td>5497</td>
<td>5516</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(1 hr, 31 mins, 56 secs)</td>
</tr>
<tr>
<td>Composite Pipeline</td>
<td>288</td>
<td>1536</td>
<td>256</td>
<td>2080</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(34 mins, 40 secs)</td>
</tr>
</tbody>
</table>

of parallelizing many vision applications. Although workstation clusters may or may not be used in the final system implementation, they can provide significant support for developing and prototyping applications requiring a large amount of computing time, in many research and other organizational setups which do not have dedicated parallel computing facilities.

6.5 Summary

In this chapter, we discussed parallel implementation of the \textit{recognition phase} of the geometric hashing algorithm used for object recognition. We also discussed parallelization of multi-scale active shape description process using three different patterns, namely, Temporal Multiplexing, Pipeline, and Composite Pipeline. The recognition phase of the geometric hashing algorithm performs several probe steps for identifying an object in a scene image. Each probe step (associated with a basis set) comprises a sequence of operations for finding potential models that match the scene features. We have developed a coarse-grained parallel algorithm for the recognition phase. This algorithm performs multiple probes on different workstations, concurrently. The operations of each probe are however performed on a single workstation. The performance of this parallel algorithm parallelized using the Farmer-Worker pattern has shown encouraging results. The performance results are sometimes even better than those in earlier implementations performed
on dedicated parallel machines.

The parallelization of the multi-scale active shape description process for MR brain images in epileptic patients has also shown promising results. The sequential execution time required to process 96 image slices is 7 hrs, 20 mins. This includes time required for performing preprocessing, propagation, and shape focusing steps in the shape description process. The corresponding observed/estimated parallel execution times using Temporal Multiplexing, Pipeline (Multiple Pipelines), and Composite Pipeline patterns are 58 mins; 1 hr, 31 mins, 56 secs; and 34 mins, 40 secs, respectively. Of the three patterns, Temporal Multiplexing is the simplest to implement. However, not all modules can be parallelized using this pattern alone. Pipeline pattern has limited scalability with respect to increase in number of workstations used in parallelization. Using Multiple Pipelines solves this problem partially but not completely. Composite Pipeline pattern resolves limitations in both Temporal Multiplexing and Pipeline patterns, and therefore achieves better performance results in comparison with other two patterns.

The examples in this chapter illustrate that using simple design patterns and most of the existing sequential code, the workstation environment can offer significant benefits for parallelizing many high level vision algorithms and/or applications. They can provide significant support for developing and prototyping applications requiring large amount of computing time, in many research and other organizational setups which do not have dedicated parallel computing facilities.
Chapter 7

Conclusion

7.1 Aims and Motivation

The research work in this thesis is aimed at presenting and evaluating a set of design patterns intended to support parallelization of vision applications on coarse-grained parallel machines, such as a cluster of workstations. Workstation environments have recently proved to be effective and economical platforms for high performance computing compared to the conventional parallel machines. They offer several advantages for parallelizing and executing large applications on a relatively low-priced and readily available pool of machines. However, developing parallel applications on such machines involves complex decisions such as dividing the applications into several processes, distribution of these processes over various processors, scheduling of processor time between competing processes, and synchronization of the communication between different processes.

Developing parallel programs to control these decisions usually involves writing explicit program code for process scheduling, process communication, and sometimes even computation in a single routine. This style of parallel code development increases program complexity, and reduces program reliability and code reusability. Writing explicit parallel code for parallelizing various applications on a cluster of workstations has some additional problems. The available machines and their capabilities can vary dynamically during
program execution or from one execution to another. This can sometimes lead to a significant reduction in overall performance of an application. Also, most developers do not wish to spend time in low level programming details in order to gain advantages of potential parallelism in an application. About 69% of parallel programmers (Pancake, 1996) modify or use existing blocks of code to compose new programs. Moreover, the modification or partial reuse of existing code or program design is often restricted to individual developers. There is very little sharing of design knowledge among developers.

The parallel programs used for implementing majority of vision tasks utilize a finite set of recurring algorithmic structures or parallel programming models. Our research has aimed at capturing and articulating the design information in these algorithmic structures in the form of design patterns. We have specified various aspects of parallel behavior of each design pattern (e.g. structure, process placement, communication patterns, etc.) in its definition or separately as issues to be addressed explicitly during its implementation. Design patterns decouple the code for implementing low level parallel programming details (i.e. process scheduling, communication, etc.) from the code for managing the actual computation. Such decoupling ensures program reliability and code reusability. Design patterns capture design information in a form that makes them usable in different situations and in future work. The design patterns presented in this thesis would enable researchers and developers to implement many interactive and batch applications in computer vision on workstation clusters.

A cluster of workstations is characterized by high communication costs and a variation in speed factors of individual machines in the network. A key factor that minimizes the effect of high communication costs on performance is 'granularity' (section 1.1) of a parallel algorithm, which describes the amount of work associated with each process/task relative to the communication. A cluster of workstations is inherently coarse-grained. We have formulated the design patterns so that they implement coarse-grained parallelism. Also, due to variation in speed factors of individual machines, an application parallelized on such machines needs to include proper load balancing strategies in order to obtain maximum performance gains. The design patterns presented in this thesis attempt to distribute the work load according to the speed factors of individual machines in the network. This load balancing is performed either statically (i.e. before the start of the computation), or
Chapter 7. Conclusion

dynamically (during the computation).

We began our work by analyzing the computation and communication characteristics of vision tasks. We identified various forms of parallelism in vision tasks and formulated design patterns to implement these tasks. Each design pattern captures common designs used by developers to parallelize their tasks. We presented a catalogue of design patterns to implement various forms of parallelism in vision tasks on a cluster of workstations.

Our next goal in this thesis has been to evaluate the use of these design patterns for parallelizing vision tasks on a cluster of workstations. We have implemented representative vision algorithms in low, intermediate and high level vision processing, and presented the experimental results of corresponding parallel implementations. The results of these implementations have helped us to critically assess the use of design patterns for achieving performance gains in various algorithms. It has also enabled evaluating the viability of using workstation clusters for implementing parallel vision applications.

7.2 Research Review

The literature on parallelization of vision algorithms/applications is vast, but there have been no previous efforts to abstract and document the design information from their corresponding parallel implementations. In chapter 3, we have attempted to capture and document this design information in the form of design patterns so that they can be used for parallelizing many vision algorithms/applications on coarse-grained parallel machines, such as a cluster of workstations. A catalogue of key design patterns for parallel vision applications would give standard names and definitions to the techniques used in parallelization of these applications. Each pattern has been described in a uniform way using a template which provides description of how each pattern works, where it should be applied and what are the trade-off in its use.

The design patterns presented in chapter 3 include, Farmer-Worker, Master-Worker, Controller-Worker, Divide-and-Conquer, Temporal Multiplexing, Pipeline, and Composite Pipeline. The Farmer-Worker pattern is used for implementing data parallel algo-
Chapter 7. Conclusion

rithms which require no communication during computation. Both Master-Worker and Controller-Worker patterns are used for parallelizing problems exhibiting data parallelism, but which require communication of intermediate results during processing. Divide-and-Conquer pattern is used for parallelizing algorithms that use a recursive strategy to split a problem into smaller subproblems and merge the solution to these subproblems into a final solution. Temporal Multiplexing pattern is used for processing several data sets or image frames on multiple processors. Finally, Pipeline and Composite Pipeline patterns are used for parallelizing applications that can be divided into a sequence (pipeline) of several independent subproblems which are executed in a determined order. In the Composite Pipeline pattern, each subproblem may be further parallelized using other relevant design patterns.

After presenting a catalogue of design patterns, our next task in this thesis has been to evaluate the use of these patterns for parallelizing vision algorithms/applications on a cluster of workstations. We have implemented various representative algorithms in low, intermediate and high level vision processing, and presented the experimental results. In chapter 4, we presented parallel implementations of some representative low level vision algorithms. Low level algorithms parallelized using the Controller-Worker pattern (e.g. histogram equalization and 2D-FFT) do not result in any significant speedups due to the time complexity of all-to-all worker communications in this pattern. But other low level algorithms parallelized using the Farmer-Worker pattern (e.g. convolution and rank filtering) and the Master-Worker pattern (e.g. 'iterative' image sharpening and image restoration) have shown encouraging results. However, applications parallelized using the Master-Worker pattern on enterprise clusters (section 2.5.3) may result in dynamic load imbalances and subsequently reduction in overall performance of the application.

In chapter 5, we presented parallel implementations of two intermediate level vision algorithms, namely, region-based split and merge segmentation algorithm, and the line grouping algorithm based on the principles of perceptual organization. The segmentation algorithm parallelized using the Divide-and-Conquer (DC) pattern does not exhibit performance scalability owing to increase in corresponding time required for merging the segmented subimages. If communication time is not a dominant factor, the performance of an algorithm parallelized using a DC pattern is in fact influenced mainly by the time com-
plexity of the merging operation. The line grouping algorithm has been parallelized using an 'iterative' variant of the Controller-Worker pattern. The performance of the parallel line grouping algorithm, however, does not show any improvement over its corresponding sequential implementation. The time spent in actual computation is significantly lower than the time spent in all-to-all worker communications in the Controller-Worker pattern. In fact, it is very difficult to achieve any significant performance gains using Controller-Worker pattern, specially when it involves frequent all-to-all worker communications.

In chapter 6, we discussed parallel implementation of the recognition phase of the geometric hashing algorithm used for object recognition. The recognition phase performs several probe steps for identifying an object from a scene image. Each probe step (associated with a basis set) comprises a sequence of operations for finding potential models that match the scene features. We developed a coarse-grained parallel algorithm for the recognition phase by performing multiple probes on different workstations, concurrently. The operations of each probe are however performed on a single workstation. The parallel implementation of the recognition phase (using the Farmer-Worker pattern) has in certain cases achieved better results than earlier implementations performed on dedicated parallel machines.

We also discussed parallelization of multi-scale active shape description process using three different patterns: Temporal Multiplexing, Pipeline, and Composite Pipeline. All three implementations have shown promising results. Of the three patterns, Temporal Multiplexing pattern is the simplest to implement since it allows most of the existing sequential code to be used in the parallel implementation. However, not all modules of this application can be parallelized using Temporal Multiplexing pattern alone. Use of Pipeline pattern increases degree of parallelization but this pattern has limited scalability. Composite Pipeline pattern resolves limitations in both Temporal Multiplexing and Pipeline pattern, and therefore achieves better application performance compared to other two patterns.

To summarize, the examples in this thesis have shown that for most low level and high level algorithms in vision the workstation environment offers reasonable and sometimes significant benefits for parallelizing these algorithms. Intermediate level algorithms, how-
ever, do not represent ideal candidates for parallel implementation on workstation clusters due to their 'communication-intensive' nature. Most of the applications parallelized using Farmer-Worker, Temporal Multiplexing, and Composite Pipeline patterns have shown encouraging results. Applications parallelized using Master-Worker, Divide-and-Conquer, and Pipeline patterns have shown satisfying results. Applications parallelized using the Controller-Worker pattern have, however, not resulted in any significant performance gains. Also, the medical imaging application in chapter 6 illustrates that workstation environments can provide significant support for developing and prototyping applications requiring large amount of computing time, in many research and other organizational setups which do not have dedicated parallel computing facilities.

7.3 Contributions of the Research work

The contributions of this dissertation can be evaluated in terms of: a catalogue of design patterns for parallel vision systems, coarse-grained parallel algorithms for representative vision applications, and critical assessment of the use of design patterns in implementing these applications on workstation clusters. We repeat/summarize these contributions again as follows:

• **Catalogue of design patterns:** We presented a catalogue of design patterns for parallel vision systems, describing each pattern in terms of intent, motivation, structure, interaction amongst the components, and applicability. This description enables selection and use of a design pattern in different situations and in future work.

• **Coarse-grained parallel algorithms:** We presented coarse-grained parallel algorithms and implementations for several vision tasks such as convolution, image filtering, image restoration, region-based segmentation, line grouping, and geometric hashing algorithm for object recognition. We also presented different parallel implementations of the multi-scale active shape description process (an application in medical imaging) using different design patterns.
• *Implementation on a cluster of workstations:* Using relevant design patterns, we
performed parallel implementations of the selected representative vision tasks stated
above. The results of these implementations enable critical assessment of the design
patterns for achieving improvements in application performance. It also enables
evaluating the viability of using workstation clusters for implementing parallel vision
applications.

### 7.4 Comparison with related work

Although the concept of abstracting common parallel programming designs in the form
of design patterns is new, there have been several prior efforts to identify and capture
general parallel programming designs/models (Chandy & Kesselman, 1991), (Kung, 1989)
as software components (e.g. *implementation machines* (Zimran et al., 1990), *templates*
(Singh et al., 1991), *assets* (Schaeffer et al., 1993), and *skeletons* (Darlington et al., 1993)).
These software components comprise 'ready-to-use' software routines for implementing
low level programming details (e.g. process scheduling, communication, etc.) in the
corresponding parallel programming models. The software systems based on these software
components allow programmers to write their parallel programs in terms of these software
components. These systems automatically insert the necessary code for process scheduling
and communication in order to realize the corresponding parallel implementation.

However, these systems do not choose the type of parallelism to apply; this choice is
left to the developer who judges and selects the best form of parallelism in a particular
application. Also, most of these systems have limited applicability. For example, the
Enterprise system (Schaeffer et al., 1993) does not support data parallelism, one of the
most important form of parallelism in computer vision. Most of these systems do not
support complex and/or domain-specific parallel programming models (e.g. parallelism
represented by the Composite Pipeline pattern in vision).

Our research work of presenting design patterns for parallel vision systems differs from
these approaches. We do not present 'ready-to-use' program code that can be simply in­
serted as software routine in a parallel implementation. We instead identify and document
explicitly various parallel programming models commonly occurring in parallel solutions of problems in certain domain, such as computer vision. The ‘intent’, ‘motivation’, and the ‘applicability’ aspects of the design pattern descriptions enable the user to select appropriate design pattern(s) for parallelizing a given application. The other aspects of the design pattern descriptions provide guidelines for the actual implementation of the design patterns for a particular problem.

A design methodology for parallelizing complete vision systems has also been presented by Downton et al. (Downton et al., 1996). Their design method, based on pipeline of processor farms (PPF), enables parallelization of complete vision systems (with continuous input/output) on MIMD parallel machines. The parallelization process in their design model is performed in a top-down fashion, where parallel implementations of individual algorithms are treated as components in the design model. While the design methodology in (Downton et al., 1996) has been implicit, our work has concentrated on making this design methodology explicit. We have documented the PPF design method in the form of Composite-Pipeline pattern in this thesis. Also, the design method in (Downton et al., 1996) discusses parallelization at mostly application level. Our work has attempted to discuss parallelization at both algorithmic and application levels in vision.

The main disadvantage of the design patterns is that they do not provide a detailed solution. A pattern provides a generic scheme for solving a class of problems, rather than ‘ready-to-use’ software module which can be inserted in program. A user needs to implement this scheme according to the requirements of a given problem. A pattern only provides guidance for solving problems, but it does not provide complete solutions.

7.5 Future work

The research work in this thesis has aimed at presenting a set of design patterns intended to support parallelization of vision application on a cluster of workstations. Using these design patterns we have also parallelized representative vision algorithms in order to demonstrate their usefulness in implementing these algorithms on workstation clusters. The research work however raises further questions and brings up research topics in a
number of research areas such as:

- **Fault tolerance:** The available resources in workstation environments (especially in enterprise clusters) can change dynamically during parallel execution of an application. A workstation may become overloaded, or may be powered off for maintenance purposes or, in the worst case, may crash. The first two cases may be predicted or known in advance. The third case is unexpected and may result in significant loss of processing time. The use of common methods, such as checkpointing and, error detection and recovery, have high overheads. An alternative method is to include fault tolerance mechanisms in each design pattern. Some such attempts (for workstation environments) have been explored in the 'processor farm' (Clematis, 1994) and the 'supervisor-worker' (Magee & Cheung, 1991) models (both models represent Farmer-Worker form of computation).

Detecting a failure in some worker component of a Farmer-Worker pattern is relatively easy. The Farmer component can detect (and rectify) such a failure when some worker component does not respond within a certain time limit. Other strategies for detecting failures in either Farmer component or process communication may be similarly devised. Detecting and rectifying failures in other patterns (e.g. Master-Worker and Pipeline) is however complicated. Each worker component in these patterns sends/receives messages from other worker components. A failure in any worker component can lead to deadlock. Devising mechanisms for handling such situations is a challenging task.

- **Load balancing:** The Farmer-Worker and Temporal-Multiplexing patterns have an inherent load balancing property. However, other patterns may suffer from load imbalances during their execution, especially when implemented on enterprise clusters. There is a need for devising mechanisms in order to minimize the effect of load imbalances in these patterns. Load balancing schemes may be incorporated in the pattern itself. For example, when a workstation executing a worker component of the Master-Worker pattern is overloaded with external processes, the worker component may be transferred to another free workstation. The overloaded worker components may be detected after every cycle of fixed number of iterations, until the completion of computation. The code for performing load balancing operations
may be included in the pattern implementation, or may be part of a separate design pattern implementation.

- **Performance prediction:**

Designing practical models for predicting parallel execution time of an application implemented on an enterprise cluster has been a challenging research area (Yan et al., 1996). We intend to study the feasibility of designing such models for the design patterns in parallel computer vision. Each design pattern may include a performance prediction model as in *skeletons* (Darlington et al., 1993) or *implementation machines* (Zimran et al., 1990). The complexity of the performance prediction model depends on the structure of the underlying design pattern. For example, using sequential time of an algorithm, it is relatively easy to predict the approximate parallel execution time in the Farmer-Worker and Temporal-Multiplexing implementations. Similarly, if the sequential execution time of each component in the Pipeline and Composite Pipeline patterns is known, it is relatively easy to predict the parallel execution time of the corresponding application. However, predicting performance in Master-Worker or Divide-and-Conquer pattern is relatively difficult.

Some important factors which need to be considered while designing performance prediction models for each design pattern (implemented on a workstation cluster) include computational complexity of the problem, number of workstations used in parallelization, relative speed factors of individual machines and the network bandwidth. The complexity of the performance prediction models is also influenced by nature of the vision algorithms. While it is relatively easy to predict performance in well-structured low level vision algorithms, predicting performance in intermediate and high level vision algorithms is relatively difficult due to uncertainties in computations.
Appendix A

Notation

A.1 Pattern Diagram

We use a variant of the object model to describe the components and their relationships in a design pattern (Buschmann et al., 1996).

The components are shown as rectangular boxes, denoting the name of the components and the associated procedures within the components. A line that connects the components denotes an association.
A.2 Object Interaction Charts

We adapt the Object Message Sequence Chart notation (OMSC) given in (Buschmann et al., 1996) to describe the object interactions among the components of a pattern.

The components in a pattern are drawn as rectangular boxes. They are labeled with their corresponding names. The activities of the components are denoted by the vertical bars attached to the bottom of the box (activity lines). The messages between the components are denoted by the horizontal arrows. The time elapsed is shown from top to bottom, however, the time scale is not scaled. An iterative computation is shown by an upward arrow, while, a procedure call within a pattern component, is shown by a small downward arrow.
Bibliography


182
Bibliography.


Efimov, N. V. (1966), An elementary course in analytical geometry, Pergamon Press, Oxford.


Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1994), Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, Reading, MA, US.


Bibliography.


Bibliography.


Wang, C. L. (1995), High performance computing for vision on distributed memory machines, PhD thesis, University of Southern California, USA.
Bibliography.


