Markov Random Fields in Visual Reconstruction: a Transputer-Based Multicomputer Implementation

Ola Siksik

Technical Report 90-40
September 1990

Department of Computer Science
The University of British Columbia
Vancouver, BC, Canada V6T 1W5

Abstract

Markov Random Fields (MRFs) are used in computer vision as an effective method for reconstructing a function starting from a set of noisy, or sparse data, or in the integration of early vision processes to label physical discontinuities. The MRF formalism is attractive because it enables the assumptions used to be explicitly stated in the energy function. The drawbacks of such models have been the computational complexity of the implementation, and the difficulty in estimating the parameters of the model.

In this thesis, the deterministic approximation to the MRF models derived by Girosi and Geiger [10] is investigated, and following that approach, a MIMD based algorithm is developed and implemented on a network of T800 transputers under the Trollius operating system. A serial version of the algorithm has also been implemented on a SUN 4 under Unix.

The network of transputers is configured as a 2–dimensional mesh of processors (currently 16 configured as a 4×4 mesh), and the *input partitioning* method is used to distribute the original image across the network.

The implementation of the algorithm is described, and the suitability of the transputer for image processing tasks is discussed.

The algorithm was applied to a number of images for edge detection, and produced good results in a small number of iterations.

Contents

List of Figures

Acknowledgement

I would like to thank Dr. Jim Little for his guidance, patience, and support throughout my work. Special thanks are also due to Jie Jiang for his helpful hints and wonderful monitor, Norm Goldstein for being there when Trollius was not, and Hilde Larsen for the graphical display and all the emotional support.

Chapter 1

Introduction

In order to give a viewer information about a three dimensional scene, many algorithms have been developed on several early vision processes, such as edge detection, stereoposis, motion, texture, and color. This information refers to properties of the scene such as shape, distance, color, shade, or motion, and it is usually noisy and sparse. Therefore more processing is necessary to extract the relevant information, and fill in the sparse data. This process is usually referred to as *visual reconstruction.*

Blake and Zisserman [2] , define *visual reconstruction* as the process of *reducing visual data to stable descriptions*, where *visual data* comes in various forms:

- Raw intensity data direct from photoreceptors, in the form of an array of numbers.
- "Optic flow" – measures of velocities of points of an image.
- A depth map, consisting of points embedded, usually sparsely, in the viewer's coordinate–frame. At each point, depth (distance from the viewer) is known. Depth maps may be produced by stereoposis, or they may be obtained by appropriate processing of optical flow.
- Sets of discrete points making up curves in a 2D image, or in 3D.

In each case, *data must be reduced in quantity, with minimal loss of meaningful content, with the compressed form being stable.*

Many researchers [26], [11], [18], [12], [34] have investigated the use of Markov Random Fields (MRFs) in computer vision in the last few years. MRF models can be generally used in the construction of a function starting from a set of noisy or sparse data. They can also be applied to integrate early vision processes to label physical discontinuities[23].

The essence of the MRF model is that the probability of a certain value of the field (for a set of data) at a given site depends only on neighboring sites, and that probability distribution is given as a Gibbs distribution. An "energy function" that contains some *a priori* information about the system and its probability distributions can be used to specify the model. In the standard approach, an estimate of the field and its discontinuities is given by the configuration that maximizes the probability distribution, or equivalently that minimizes the energy function. Since the discontinuity field is a discrete valued field, this becomes a combinatorial optimization problem that can be solved by methods of the Monte Carlo type [25] (simulated annealing [18], for example).

Girosi and Geiger [11, 10] introduce deterministic approximations to MRF models. They use the *mean field theory* to find deterministic equations for Markov Random Fields analytically. The solution of these non-linear equations approximates the solution of the statistical problem. This approach will be presented and discussed in detail in Chapter 2.

Early vision is the most computationally intensive part of a vision system. The prospect of near real-time image processing has only recently been raised by the introduction of fast, parallel computers. A transputer-based multicomputer [17] is a general-purpose parallel and distributed computing environment that offers potentially enormous computational power at a reasonable cost.

A transputer [22] is a VLSI building block for concurrent processing systems, comprising of a processor, the on–chip memory and four serial communication links. Two transputers can be connected by connecting a link of one transputer to a link of the other. The flexibility of the reconfiguration of the transputer network makes it possible to support various kinds of parallel and distributed applications. The algorithm was implemented on a 2–dimensional mesh of transputers.

The underlying operating system was Trollius 2.0 [3, 4]. Trollius is a topology-independent operating system. The algorithm was implemented using the C programming language which is supported by the operating system.

There are several approaches to implementing the algorithm on a parallel machine, but the one that is most suitable to the system's architecture is the input partitioning method. Using the input partitioning method, the image was divided into a number of subimages equal to the number of the nodes in the mesh. The subimages were distributed by a master node to the rest of the network nodes. Each processor then performs the computation on its subimage, and exchanges the borders with its immediate neighbors after every iteration. When the computation is completed, each processor returns its final subimage to the master transputer where the complete final image is assembled and written to an external file.

The purpose of this thesis is to evaluate the performance of the technique proposed by Girosi and Geiger, and investigate the suitability of the transputers for low level vision applications. The algorithm used for the experiments is an implementation of the one described by Girosi and Geiger.

The rest of the thesis is organized as follows:

Chapter 2 introduces the theory behind MRFs, the weak membrane energy function,

and the deterministic approach to minimizing it. **Chapter 3** presents some of the other approaches to image reconstruction, namely the graduated non convexity algorithm by Blake and Zisserman, and the stochastic relaxation algorithm by Geman and Geman.

Chapter 4 presents different approaches to programming parallel vision algorithms: The Apply programming model, the dataflow language approach, the parallel vector model, and the systolic approach. **Chapter 5** describes the software development environment as well as the details of the parallel implementation. **Chapter 6** contains results obtained by applying the algorithm to sample images, and some evaluation of the communication time, the processing speed, and other issues that arise when using a transputer-based multicomputer.

Chapter 2

Theory and Mathematical Background

Consider the problem of approximating a surface given a set of sparse and noisy data g on a regular $2D$ lattice. We think of the surface as a field f defined in the regular lattice, such that the value of the field at each site in the lattice is given by the surface height at this site [See Fig. 2.1].

We are interested in *the conditional probability of f given the data g*, $P(f \mid g)$. Bayes' theorem allows us to write

$$P(f \mid g) \propto P(g \mid f)P(f)$$

where $P(g \mid f)$ is related to the probability distribution of the noise, and $P(f)$ is the prior probability distribution of the field f. The noise is usually assumed to be Gaussian[12], so that $P(g \mid f)$ is known. The shape of $P(f)$ depends on our *a priori* information about the system and it is what differentiates one model from another.

The Markov property asserts that *the probability of a certain value of the field at a given site depends only on neighboring sites.* If we assume the Markov property, then

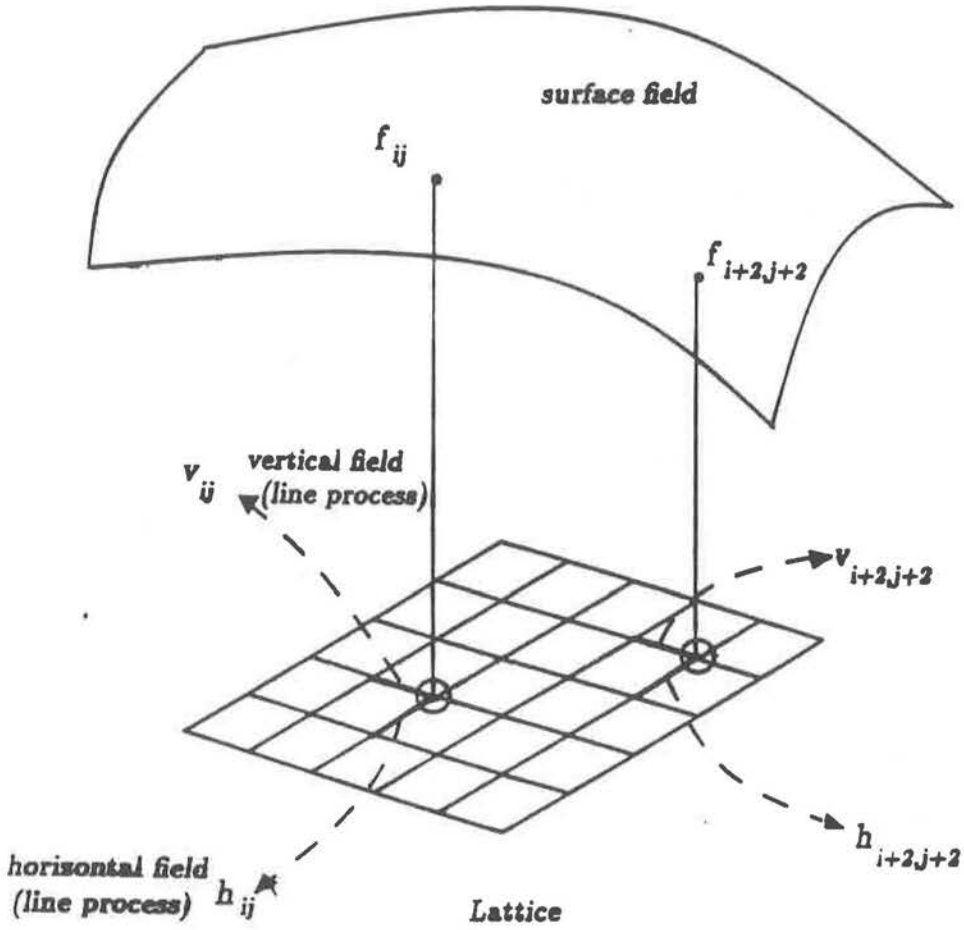


Figure 2.1: The surface field, the horizontal and vertical line processes

according to the "MRF Gibbs equivalence" [1] [11], [12], the prior probability of a state of the field f has the Gibbs form:

$$P(f) \propto e^{-\beta U(f)}$$

where $U(f) = \sum_i H_i(f)$ is an energy function that can be computed as the sum of the local contributions from each lattice site i and β is a parameter that is called the inverse of the natural temperature of the field[2]. As a result, the conditional probability can be written as

$$P(f \mid g) \propto \frac{1}{Z} e^{-\beta H_g(f)}$$

where $H_g(f)$ is usually called *the energy function of the model*, and Z is a normalizing constant called the partition function of the model [28], that is

$$Z = \sum_f e^{-\beta U(f)} \tag{2.1}$$

To include the discontinuities of the field f in this framework, another field called the *line process* l is introduced. This idea was initially proposed by Geman and Geman [12], where the line process provides an explicit representation for the absence or presence of discontinuities that break the smoothness assumption. The interaction between the fields f and l can be chosen so that the most likely configurations are piecewise continuous. The details of the method are discussed in the remaining sections of this chapter.

[1] Discussed by Geman and Geman. It states that if field f is a MRF, then the probability law of f is a Gibbs distribution. [see section 3.2]

[2] β is a measure of the certainty in the statistical model. When $\beta = \infty$ there is no uncertainty in the model.

2.1 Deterministic Approximation of MRF

Once the probability distribution has been written down, an estimate of the field is obtained with the field values that maximize it, or equivalently that minimize the energy function $H_g(f)$. A number of problems arise when this approach is taken. The first one concerns the energy function: it is often *not convex*. Because of that reason and the discrete nature of the line process fields [2], simulated annealing or similar Monte Carlo techniques must be used to solve the problem. The computational effort to obtain a good estimate of the fields is then very large.

Another problem comes from the fact that the energy function depends on some **parameters** that control the relative weights of various terms. The problem of parameter estimation has been attacked in many ways, but it is far from being completely solved. It is still not clear how they are related to the quality of the solution and to quantities of physical interest.

In their paper [11], Geiger and Girosi propose to approximate the solution of the problem formulated in the MRFs frame with its *"average solution"*. The *Mean Field Theory* is used to find deterministic equations for Markov Random Fields analytically. The solution of these non-linear equations approximates the solution of the statistical problem.

A justification to use the mean field (MF) as a measure of the field f resides in the fact that it represents the minimum variance Bayes' estimator [34]. More precisely, the variance of the field f is given by

$$Var_f = \sum_{f,l} (f - \bar{f})^2 P(f, l)$$

where $\bar{f}$ is the center of the variance, $P(f, l)$ represents a particular state of the system,

and the $\sum_{f,l}$ represents the sum over all possible configurations of f and l. Minimizing Var_f with respect to all possible values of $\bar{f}$, gives us

$$\frac{\partial}{\partial \bar{f}} Var_f = 0 \Rightarrow \bar{f} = \sum_{f,l} fP(f,l).$$

This implies that the minimum variance estimator is given by the MF value.

2.1.1 The Effective potential and the Deterministic Equations

Let g be a given set of data, defined on a 2–D lattice, f a field associated with the field to be constructed, and l a field whose value is 1 where a discontinuity occurs and 0 elsewhere. We consider an energy function of the general form

$$H_g(f,l) = E_{fg}(f,g) + E_{fl}(f,l)$$

where the first term is usually $\sum_i (f_i - g_i)^2$, coming from the Gaussian distribution of the noise and the other terms contain the *a priori* information about the system. Due to the discrete nature of the line process field, the minimum of the energy function can not be found by computing derivatives with respect to the variables, unless we can eliminate the line process field from the probability distribution. Girosi and Geiger achieve this by using the partition function Z. They write the function Z as

$$Z = \sum_{f,l} e^{-\beta H_g(f,l)} = \sum_f e^{-\beta E_{fg}(f,g)} \sum_l e^{-\beta E_{fl}(f,l)} \tag{2.2}$$

where $\sum_{f,l}$ means the sum over all possible configurations of the fields f and l. The *effective potential* is defined as

$$E^{\beta}_{eff}(f) = -\frac{1}{\beta} ln \sum_l e^{-\beta E_{fl}(f,l)} \tag{2.3}$$

the data term plus the effective potential (2.5) represent the **free energy** of the system [10]. The mean field solutions are obtained by minimizing the free energy. In [11], the following equations were obtained as a solution to (2.5):

$$\begin{aligned}\bar{f}_{i,j} = g_{i,j} &- \alpha(\bar{f}_{i,j} - \bar{f}_{i,j-1})(1 - \bar{v}_{i,j}) + \alpha(\bar{f}_{i,j+1} - \bar{f}_{i,j})(1 - \bar{v}_{i,j+1}) \\ &- \alpha(\bar{f}_{i,j} - \bar{f}_{i-1,j})(1 - \bar{h}_{i,j}) + \alpha(\bar{f}_{i+1,j} - \bar{f}_{i,j})(1 - \bar{h}_{i+1,j})\end{aligned} \tag{2.10}$$

where

$$\bar{h}_{i,j} = \frac{1}{1 + e^{\beta(\gamma - \alpha(\bar{f}_{i,j} - \bar{f}_{i-1,j})^2)}} \qquad \bar{v}_{i,j} = \frac{1}{1 + e^{\beta(\gamma - \alpha(\bar{f}_{i,j} - \bar{f}_{i,j-1})^2)}} \tag{2.11}$$

Equation (2.10) gives the field at site (i, j) as the sum of data at the same site, plus an average of the field at its neighboring sites. This average takes into account the difference between the neighbours. The larger the difference, the smaller is the contribution to the average. This is captured by the term $(1 - l_{i,j})$, where $l_{i,j}$ is either the horizontal or the vertical line process. At the zero temperature limit ($\beta \to \infty$), the line process becomes [4] 1 or 0, and then only terms smaller than a threshold must be taken into account for the average.

2.3 Improving the weak membrane model

One property of physical images that has not been exploited in the previous model is the smoothness of the discontinuity field. Isolated discontinuities are very unlikely to occur, and the presence of a discontinuity at a site increases the probability of a discontinuity at a neighboring site. This smoothness constraint was incorporated in the model by adding a new term to the energy function [11], [10]. Thus the total energy becomes

$$E_2 = E_1 + E_{ll}$$

[4] It is equal to 1, only when $(f_{i,j} - f_{i-1,j}) \geq \sqrt{\frac{\gamma}{\alpha}}$.

where E_1 is given by (2.6), and the new term is defined as

$$E_{ll} = -\epsilon\gamma \sum_{i,j} (h_{i,j}h_{i,j-1} + v_{i,j}v_{i-1,j})$$

and ϵ is a new parameter related to the degree of smoothness of the discontinuity field, and will be discussed in more detail in the next section.

2.3.1 Averaging Out The Line Process

As in the previous case, we are interested in the contribution of the line process to the partition function. This task is more difficult than the previous one, but the mean field approximation is used in a similar fashion to obtain an approximate result for the effective potential. Girosi and Geiger obtain the following equation for the effective potential:

$$E_{eff}(f) = \sum_{i,j} \{\alpha((\bar{f}_{i,j} - \bar{f}_{i-1,j})^2 + (\bar{f}_{i,j} - \bar{f}_{i,j-1})^2)$$

$$- \frac{1}{\beta} ln[(1 + e^{-\beta(G^h_{i,j} - \epsilon\gamma \frac{h_{i,j-1} + h_{i,j+1}}{2})})(1 + e^{-\beta(G^h_{i,j} - \epsilon\gamma \frac{h_{i,j-1} + h_{i,j+1}}{2})})]\} \qquad (2.12)$$

where $G^h_{i,j} = \gamma - \alpha(\bar{f}_{i,j} - \bar{f}_{i-1,j})^2$ and $G^v_{i,j}$ is analogous.

> "The effect of this new effective potential can be understood if we think of the system as an ensemble of interacting particles and if we study the interaction force between particles (that is the negative of the derivative of the effective potential); In this case the gradient of the field should be thought of as the relative distance between the particles. We notice that when the gradient is low, the force is linear and attractive, as the force of the ideal spring. When the gradient increases, the force quickly decreases, and, unlike the usual spring, becomes repulsive, pushing the particles apart. This effect takes place only in a limited interval of values of the gradient; when it becomes too large, the spring breaks up and the force goes to zero."[11]

As a result, the overall effect will be of a *smoothing* where the gradient is smaller than a threshold, and an *enhancing*, where the gradient is "sufficiently large". Where the gradient is too large, no smoothing or enhancing will take place. The enhancing effect is due to the new term E_{ll} in the energy function, and its intensity is controlled by the parameter ϵ.

As in the previous case , a set of non linear equations that relates the mean values of the discontinuity field with the mean values of the surface field is obtained :

$$\bar{h}_{i,j} = \sigma_\beta(\alpha(\bar{f}_{i,j} - \bar{f}_{i,j-1})^2 - \gamma + \epsilon\gamma\frac{\bar{h}_{i,j-1} + \bar{h}_{i,j+1}}{2})$$

and

$$\bar{v}_{i,j} = \sigma_\beta(\alpha(\bar{f}_{i,j} - \bar{f}_{i-1,j})^2 - \gamma + \epsilon\gamma\frac{\bar{v}_{i-1,j} + \bar{v}_{i+1,j}}{2}) \tag{2.13}$$

where $\sigma_\beta(x) = \frac{1}{1+e^{\beta x}}$ (the sigmoid function).
An analogous equation is obtained for the surface field f:

$$\begin{aligned}\bar{f}_{i,j} = g_{i,j} &- \alpha\bar{\Delta}^{\bar{v}}_{i,j}(1 - \bar{v}_{i,j}) + \alpha\bar{\Delta}^{\bar{v}}_{i,j+1}(1 - \bar{v}_{i,j+1}) - \alpha\bar{\Delta}^{\bar{h}}_{i,j}(1 - \bar{h}_{i,j}) + \alpha\bar{\Delta}^{\bar{h}}_{i+1,j}(1 - \bar{h}_{i+1,j}) \\ &+\alpha\epsilon\bar{\Delta}^{\bar{h}}_{i,j}\bar{h}_{i,j}\sigma_\beta(\gamma - \alpha(\bar{\Delta}^{\bar{h}}_{i,j})^2) - \alpha\epsilon\bar{\Delta}^{\bar{h}}_{i,j+1}\bar{h}_{i+1,j}\sigma_\beta(\gamma - \alpha(\bar{\Delta}^{\bar{h}}_{i+1,j})^2) \\ &+ \alpha\epsilon\bar{\Delta}^{\bar{v}}_{i,j}\bar{v}_{i,j}\sigma_\beta(\gamma - \alpha(\bar{\Delta}^{\bar{v}}_{i,j})^2) - \alpha\epsilon\bar{\Delta}^{\bar{v}}_{i,j+1}\bar{v}_{i,j+1}\sigma_\beta(\gamma - \alpha(\bar{\Delta}^{\bar{v}}_{i,j+1})^2)\end{aligned} \tag{2.14}$$

We notice that the set of equations above now form a set of non linear equations and that they are not as simple as in the previous model.

The last set of equations has been used for the implementation of the algorithm. The implementation details are discussed in Chapter 5.

Chapter 3

Related Work

The weak membrane energy function has been studied by Blake and Zisserman [2] in the context of edge detection and surface interpolation. Their approach does not use Markov Random Field formulation, but they minimize the energy function. From a statistical mechanics point of view the mean field solution does not minimize the energy function, but this becomes true in the case of a zero temperature [10]. The GNC algorithm is presented below.

3.1 The Graduated Non Convexity Algorithm

The main problem with the weak membrane energy function is that it is not convex, and a classical optimization technique can not be used to find the minimum because one could be trapped in a local minimum.

The GNC algorithm provides a convex approximation $E^{(1)}$ to the energy E. A family of functions $E^{(p)}, p \in [0,1]$ is defined such that $E^{(1)} \equiv E$, and $E^{(p)}$ varies continuously, in a particular prescribed manner, as p decreases from 1 to 0

For $0 \leq p \leq 1$ the $E^{(p)}$ are non-convex. Of the whole family, only $E^{(1)}$ is convex. The $E^{(p)}$ are obtained by replacing the local interaction energy term by a new energy term

that is independent of the line process variable. The GNC algorithm for the weak string[1] model is given in Fig. 3.1.

The combination of boolean and real functions complicates the minimization of E. For this reason, the S and P energy terms in the weak string energy function are combined and a new total energy is defined:

$$F = \sum_{1}^{N}(u_i - d_i)^2 + \sum_{1}^{N-1} g(u_i - u_{i+1}) \tag{3.1}$$

where

$$g(t) = \begin{cases} \lambda^2 t^2 & \text{if } |t| < \sqrt{\alpha}/\lambda \\ \alpha, & otherwise. \end{cases} \tag{3.2}$$

The energy F is now minimized only over u_i. The optimal values for l_i can be recovered from the optimal u_i as follows

$$l_i = \begin{cases} 0 & \text{if } |u_i - u_{i+1}| < \sqrt{\alpha}/\lambda \\ 1, & otherwise. \end{cases} \tag{3.3}$$

In the weak string algorithm, the parameter p varies from 1 to 0 as the solution proceeds. The solution space is correspondingly transformed from convex to non-convex. In practice, p takes on a number of discrete values from 1 to 0. For each value of p, a gradient descent algorithm is used to determine the local minima. GNC actually incorporates the successive over-relaxation (SOR) algorithm, which has a faster convergence rate than the gradient descent.

---

[1]The *weak string* energy function is the one–dimensional case of the weak membrane. A reconstruction $U = \{u_i, i = 1, \cdots, N\}$, $L = \{l_i, i = 1, \cdots, N-1\}$ is obtained from data $d = \{d_i, i = 1, \cdots, N\}$ by minimizing the energy E : min E, where $E = D + S + P$ and $D = \sum_1^N (u_i - d_i)^2$, $S = \lambda^2 \sum_1^{N-1}(u_i - u_{i+1})^2(1 - l_i)$, $P = \alpha \sum_1^{N-1} l_i$. The constant λ controls the scale of reconstruction. Constant α is a penalty levied for the inclusion of a discontinuity and controls resistance to noise.

Choose λ and α
SOR parameter: $\omega = 2/(1 + 1/\lambda)$
Function sequence: $p \in (1, 0.5, 0.25, \ldots, 1/\lambda)$
Iterate n=1,2,....
For $i = 2, \ldots, N-1$;

$$u_i^{(n+1)} = u_i^{(n)} - \omega\{2(u_i^{(n)} - d_i) + g^{(p)'}(u_i^{(n)} - u_{i-1}^{(n+1)}) + g^{(p)'}(u_i^{(n)} - u_{i+1}^{(n)})\}/(2 + 4\lambda^2)$$

where

$$g^{(p)'} = \begin{cases} 2\lambda^2 t & \text{if } |t| < q \\ -\frac{1}{2p}(|t| - r)sign(t), & \text{if } q \le |t| < r \\ 0, & \text{if } |t| \ge r \end{cases}$$

and

$$r^2 = \alpha(4p + 1/\lambda^2) \qquad q = \alpha/\lambda^2 r$$

Initially $p = 1$. Can switch to successive p after convergence at current p.
Appropriate modification is necessary at boundaries.

Figure 3.1: The GNC algorithm for the weak string

3.2 Geman and Geman: Stochastic Relaxation

Geman and Geman [12] have used statistical mechanics to establish a link between mechanical systems and probability theory which they used in the field of image restoration. They have shown that signal estimation using Gibbs probability distribution is the right approach if you have certain *a priori* probabilistic beliefs about the world in which the signal originated. Specifically, the beliefs are: that the signal being estimated is sampled from a "Markov Random Field" and that Gaussian noise was added in the process of generating the data.

The stochastic relaxation approach of Geman and Geman [12] can be informally described as follows.

1. A local change is made in the image based upon the current values of pixels and boundary elements in the immediate neighbourhood. This change is *random*, and is generated by sampling from a local conditional probability distribution.

2. The local conditional distributions are dependent on a global control parameter T called "temperature". At *low* temperatures, the local conditional distributions concentrate on states that *increase* the objective function. At *high* temperatures, the distribution is essentially uniform. The limiting cases, $T = 0$ and $T = \infty$, correspond respectively to greedy algorithms (such as the gradient descent) and undirected (i.e. purely random) algorithms.

3. Local energy minima are avoided by beginning at high temperatures where many of the stochastic changes will actually decrease the objective function. As the relaxation proceeds, temperature is gradually lowered, and the process behaves

increasingly like iterative improvement[2].

In the Geman and Geman algorithm [12], the original image is referred to as a pair $X = (F, L)$ where F is the matrix of observable pixel intensities and L denotes a matrix of unobservable edge elements. F is referred to as the *intensity process*, and L as the *line process*. Both the line and intensity processes are said to be Markovian in nature.

In their answer to the question: "What does it mean for a process to be Markovian in nature ?" Geman and Geman explain:

> Let $Z_m = \{(i,j) : 1 \leq i,j \leq m\}$ denote the $m \times m$ integer lattice; then $F = \{F_{i,j}\}, (i,j) \in Z_m$, denotes the gray levels of the original digitized image. F is regarded as a sample realization of a random field, usually isotropic and homogeneous. Specifically, F is modelled as a Markov Random Field, or equivalently, the probability law of F is assumed to be a *Gibbs distribution.* That is, given a neighbourhood system $\Gamma = \{\Gamma_{i,j}, (i,j) \in Z_m\}$, where $\Gamma_{i,j} \subseteq Z_m$ denotes neighbors of (i,j), an MRF over (Z_m, F) is a stochastic process indexed by Z_m for which, for every (i,j) and every f,
>
> $$P(F_{i,j} = f_{i,j} \mid F_{k,l} = f_{k,l}, (k,l) \neq (i,j)) = P(F_{i,j} = f_{i,j} \mid F_{k,l} = f_{k,l}, (k,l) \in \Gamma_{i,j}) \tag{3.4}$$

In other words, a Markov Random Field is a probabilistic process in which all interaction is local. That is, the probability that a cell is in any given state is entirely determined by probabilities for states of neighboring cells. The stochastic relaxation (Heatbath) algorithm for the weak string case is shown in Fig. 3.2.

[2]This gradual reduction of temperature simulates "annealing", a procedure by which certain chemical systems can be driven to their low energy, highly regular states.

Each iteration consists of N visits made to randomly picked sites i, to update u_i and l_i. Successive new values of u_i, and l_i are generated by a Gibbs sampler [12]. Updating l_i is done by setting $l_i = l$ where l is picked randomly from the distribution:

$$P_{l_i}(l) = P(l_i = l \mid u_j, j = 1, \cdots, N; l_j, j = 1, \cdots, N-1, j \neq i), l \in \{0, 1\}.$$

For the weak string, this distribution turns out to be[1]

$$P_{l_i}(l) \propto \exp\left(-\frac{\alpha l + (1-l)(u_i - u_{i+1})^2\lambda^2}{T}\right)$$

Similarly u_i is updated to a value u chosen randomly from the distribution

$$P_{u_i}(u) = P(u_i = u \mid u_j, j = 1, \cdots, N; l_j, j = 1, \cdots, N-1, j \neq i).$$

For the weak string this is

$$P_{u_i}(u) \propto \exp\left(-\frac{(u_i - \mu_i^2)}{\sigma_i^2 T}\right)$$

where

$$\mu_i = \sigma_i^2(d_i + \lambda^2((1-l_{i-1})u_{i-1} + (1-l_i)u_{i+1}))$$

and

$$\sigma_i^2 = 1/((2 - l_{i-1} - l_i)\lambda^2 + 1)$$

The temperature T is lowered according to a truncated logarithmic schedule

$$T = T_0 \frac{log(2)}{log(2+n)}, n \geq 0$$

Figure 3.2: The heatbath algorithm for the weak string energy function

The work done by Geman and Geman has forged an unintuitive, and elegant link between mechanical systems and probability theory. Blake and Zisserman commented on the novelty of their work:

> "It comes as something of a shock, when happily using splines as a very natural, mechanical model for smooth, physical surfaces, to find that this is inescapably equivalent to making certain probabilistic assumptions! The most disturbing thing is that one is forced to accept that the surface model is a probabilistic one, and therefore includes an element of randomness."[2]

3.3 GNC Vs. Stochastic Methods

Stochastic methods were developed before deterministic ones, and they have offered crucial insights, and achieved good results. However, the Computational cost of stochastic methods is high. They have shown [1] to be two orders of magnitude slower than the GNC algorithm (for the weak membrane case).

Blake and Zisserman [2] list the following advantages of the mechanical viewpoint over the stochastic one.

- The MRF model is inherently discrete, whereas the mechanical one is continuous. A continuous model allows rigorous mathematical analysis using the calculus of variations and other tools.

- MRF parameters must be specified in the form of conditional probabilities. These parameters are unlikely to be known in advance. An understanding of the parameters, however, is possible since the MRF formulation depends on the energy function used to specify the model. In the mean field theory approach [11], the pa-

rameters control such measures as the trust in the data, the threshold for creating a line, the amount of propagation of the line, and the uncertainty of the model.

- The mechanical viewpoint (GNC) also requires some parameters to be specified, but these are the more natural ones of spatial scale, desired sensitivity to contrast, and magnitude of Gaussian noise.

- The continuous model allows viewpoint invariance, essential for veridical reconstruction of 3D surfaces from range data, to be incorporated into the energy function due to the explicit presence of differential geometric quantities in the continuous formulation.

- MRFs are not able to specify all probability distributions or equivalently, all possible energy functions.

The mean field theory approach can be regarded as a link between stochastic algorithms, and the GNC algorithm. It is based on the MRF formulation, like the stochastic algorithms, and yet, it is deterministic, like the GNC algorithm. Girosi and Geiger [11] show that the GNC algorithm can be regarded as a special case of their mean field approximation. In fact, the mean field formulation gives the GNC when $\beta = \infty$ (i.e. in the zero temperature limit).

Chapter 4

Implementing Parallel Vision Algorithms

One of the important goals of computer vision is to allow machines to undertake tasks that could previously be performed only by humans or that were too difficult or dangerous for humans to perform at all. These tasks include inspection of manufactured parts, robot guidance, and autonomous vehicle navigation. In many of these applications, the vision system must provide rapid responses to the real–time processes that interact with the environment. Even the fastest sequential processors are not fast enough to process the volume of data present in such environments. Thus parallel architectures are needed to make these real–time applications feasible.

> "Parallel architectures have been part of computer vision research for almost as long as computer vision research has existed. The Illiac series of machines, which were among the first parallel architectures, were intended in part for image processing applications." [29]

Since then, many parallel architectures have been developed including cellular array processors, pipeline machines, and pyramid machines. Some of these machines are single

instruction, multiple data (SIMD) architectures, some are multiple instruction, multiple data (MIMD) architectures, and others are hybrid. Some are inexpensive, board level products, and others are multi-million-dollar computers.

A network of transputers [22] comprises an inexpensive MIMD computer that has become increasingly popular in the last few years. A detailed description of the network used for the implementation is given in Section 5.1.1.

Parallel architectures have been difficult to program because it is not yet understood how to "cover" parallelism (hide it from the programmer) and get good performance. Therefore the programmer has to write her programs using a special language that exploits features of the computer, and that can not run on other computers, or she uses a general-purpose language which runs on many computers, but does not make use of the special features of the parallel computer.

The programmer is then faced with a dilemma: she must either ignore the special features of her computer, to increase the generality, portability and understandability of her program, or take advantage of those features to increase the efficiency and speed-up at the cost of generality and portability.

The following sections present some of the work that has been done in the area of programming parallel vision algorithms. These approaches are mainly aimed at low level vision since this is usually the most computationally expensive part of a vision system. Some of the approaches focus on providing better control and synchronization for a given type of architecture rather than generality and abstraction. An example of such an approach is the data flow model. Other approaches try to abstract away from the machine architecture allowing the programmer to concentrate more on the problem and the algorithm. They provide better portability, generality, and ease of programming.

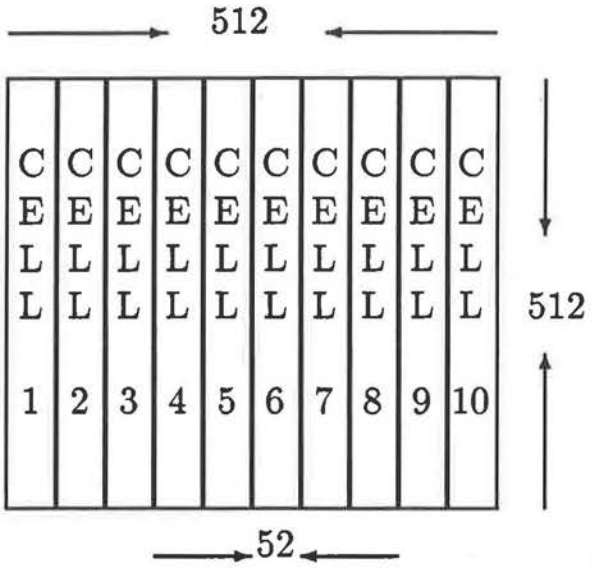


Figure 4.1: Input partitioning method on Warp

The Apply model is one example of such approaches. Some of the models were developed for fine-grained machines, with fixed or variable interconnection topologies. Others are better suited for coarse-grained parallel machines.

4.1 The Apply programming model

The *Apply* language was designed by Hamey, Webb, and Wu [13] for implementing parallel low level algorithms on the Warp machine [1], which has been developed for image and signal processing.

Low level vision algorithms are mapped onto Warp by the input partitioning method; On a Warp array of ten cells, the image is divided into ten regions [13], by columns as shown in Fig. 4.1. This gives each cell a tall narrow region to process. The advantage of such partitioning is that each cell sees a connected set of columns of the image which is

[1]The Warp machine is a linear array of ten cells called Warp cells, which are identical, and which include local data, and microcode memory, input and output ports, and a 5 MFLOPS multiplier, for a total of 10 MFLOPS per cell.

useful in many vision algorithms (e.g., median filter), and the memory requirements at a cell are minimized, since each cell must store only 1/10th of a row.

The Apply programming model is a special purpose programming approach which simplifies the programming task by making explicit the parallelism of low level vision algorithms. The Apply programming language embodies this approach. When using the Apply language, the programmer writes the procedure which defines the operation to be applied at a particular pixel location. The procedure conforms to the following:

- It accepts a window or a pixel from each input image.
- It performs arbitrary computation, usually without side effects.
- It returns a pixel value for each output image.

The Apply compiler converts the procedure into an implementation which can be run efficiently on Warp, or on a uni-processor machine in C under Unix.

An example of an Apply program is given in Fig. 4.2. In this example, a procedure for image reduction is presented. The *sample* parameter used in the procedure specifies that the Apply operation is to be applied not at every pixel, but regularly across the image, skipping pixels as specified in the integer list after sample. The window around each pixels refers to the underlying input image. For example, the program in figure 4.2 performs image reduction using overlapping 4×4 windows, to reduce an $n \times n$ image to an $n/2 \times n/2$ image.

4.2 The Dataflow Language Approach

Shapiro, Haralick, and Goulish [30] proposed a *Reconfigurable Computational Network* (RCN) machine as an inexpensive, and flexible architecture that can execute a variety of

```
procedure reduce(inimg : in array (0..3, 0..3) of byte sample (2, 2),
                 outimg: out byte)
is
    sum : integer;
    i,j : integer;
begin
    sum:= 0;
    for i in 0..3 loop
        for j in 0..3 loop
            sum := sum + in(i,j);
        end loop;
    end loop;
    outimg := sum /16;
end reduce;
```

Figure 4.2: An Apply program for image reduction

algorithms from low to high level vision. The RCN is a multi–instruction, multi-data–stream network of processors and memories with the ability to reconfigure connections from processor to processor, and from processor to memory.

The operation of the RCN involves the flow of sequences of values through a network of architectural primitives. More formally, a configuration consists of a set of processors P and a specification C of the interconnections between processors [29]. Each processor $p \in P$ is a pair $p = (I_p, O_p)$ where I_p is a named set of input lines and O_p is a named set of output lines. Each connection $c \in C$ is a quadruple $c = (o, p_1, i, p_2)$ specifying that output line o of processor p_1 connects to input line i of processor p_2. There is also a state associated with each data line where a state is a pair of values $s = (r, q)$ where r is an indication of readiness, and q is an indication of acceptance. Legal values of r are: *preactive* (the processor has not produced any values yet), *active and ready* (valid data, ready to be used by other processors), *active and not ready* (the new data is not

yet ready to be consumed), and *postactive* (the processor has terminated production). Legal values of q are *consumed*, and *unconsumed*.

A process can execute when all of its inputs are in the state *(active and ready, unconsumed)*, and all of its outputs from its previous execution have been consumed by every process to which they are inputs. A *sequence* is an ordered stream of values that are either input to the RCN, or are produced by one of the processors of the RCN.

The INSIGHT language was developed by Shapiro et.al. [29] as a dataflow language to run on the RCN.

INSIGHT programs specify relationships among sequences that will translate to relationships among architectural primitives of the RCN. A program is a sequence of configurations of the RCN designed to achieve some goal. The arguments of a program are supplied by, and its results must be received by, entities outside the RCN such as frame buffers, other external memories and CRTs.

INSIGHT programs are modular; they may invoke INSIGHT functions to perform subtasks. An INSIGHT function translates into a subgraph of the configured hardware that is useful in one or more parts of the total algorithm. It is, however, closer to the usual concept of a macro than a function in a procedural language, since a new copy of the subgraph must be included wherever the results of the function are needed. This allows all such copies of the function to operate in parallel.

Example: Convolution

Convolution is one of the most frequently used neighborhood operations. The output of a digital convolution operator at pixel (i, j) of an image can be written as [29]

$$O_{ij} = \sum_{k=0}^{n} a_k N_k(i, j),$$

where the a_k's are coefficients, $N_0(i,j) = I_{i,j}$ (the value of the input pixel), and $N_k(i,j), i = 1,\ldots,n$ are the n neighbors of pixel (i,j). The $N_k(i,j)$'s constitute the *kernel* of the convolution [29].

The convolution function can be implemented in a pipelined manner by considering the image as a linear sequence of pixels and thinking of the neighbors of a given pixel as other pixels in the sequence whose position are offset from its position by a constant amount. For example, in a 512×512 image, the pixel above a given pixel is offset from it by 512 positions in the linear sequence of pixels. An INSIGHT operator *delayed by* (dby) allows the offset idea to be efficiently used to program a pipelined convolution.

Fig. 4.3 illustrates the INSIGHT code for the convolution operation [29]. The coefficients and delays for the particular convolution are assumed to be stored in the integer memories *coefficients* and *delays*. The processing is carried out by a sequence array of processing subnets, each of which multiplies the value of the appropriately delayed pixel by the corresponding coefficient and adds the result into the overall sum. Fig. 4.4 shows the architectural configuration that would be generated from this INSIGHT program.

4.3 The Parallel Vector Model

The parallel vector model [24] is used by Little, Belloch, and Cass to describe a variety of vision algorithms to run on fine-grained parallel machines such as the Connection machine [14]. The algorithms are implemented using a set of primitive parallel operations. These primitives include general permutations, grid permutations, and the scan operation.

In a parallel vector model, all the primitive operations are defined to work on a vector of atomic values. This model is well-suited to problems where the data are numerous and the computations on the data elements are similar. Early vision problems have such

```
function convolve(image_in: integer sequence,
                  delays: integer memory,)
                  :integer sequence;
      where
        declare
          size,stages: translator integer;
          result[0:size], image[0:size]: integer seqarray;
        relations
          result[0] = coefficient[0] * image_in;
          image[0] = image_in;
          foreach stage = 1 to size
             image[stage] = image[stage-1] dby delays[stage];
             result[stage] = result[stage-1]
                           + coefficient[stage]*image[stage];
          endfor;
          convolve = result[size];
      endwhere
```

Figure 4.3: INSIGHT code for a convolution operation

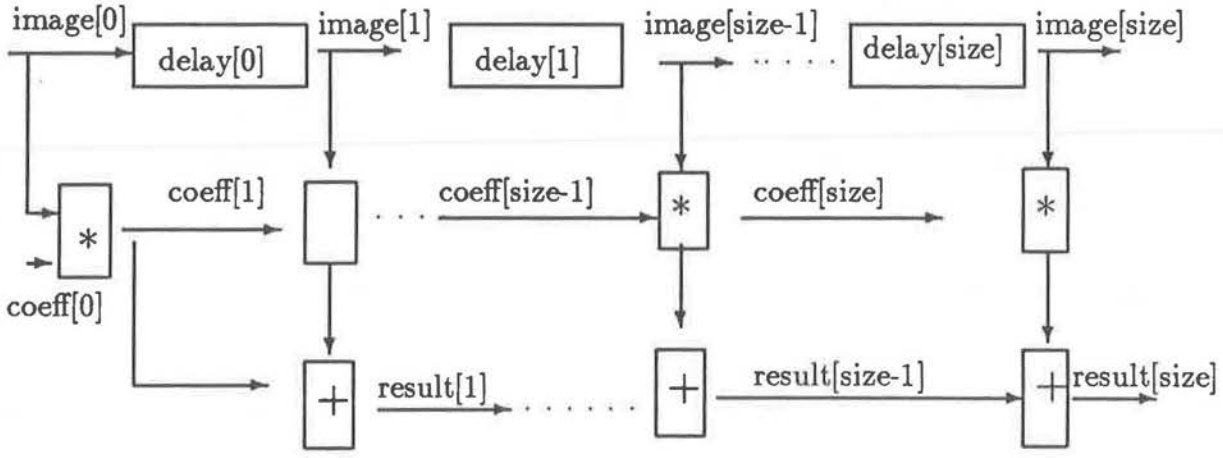


Figure 4.4: Architectural configuration for the convolution operation

properties. Early vision algorithms can then be expressed in terms of routines using primitives of the parallel vector model and modules composed of these routines.

Little et.al. [24] also show that using the routines and modules in the model, middle and high level vision algorithms can also be formulated in a natural manner for implementation on a fine-grained architecture. The primitives defined are:

- *Elementwise Arithmetic and Logical Primitives*: Each primitive operates on equal length vectors providing a result vector of the same length. Such primitives include $+$, $-$, $\times$, **OR**, and **NOT**.

- *Permutation primitives*: The permutation primitive takes two arguments: a data vector, and an index vector. It then permutes each element in the data vector to the location specified by the index vector.

- *Grid Permutation Primitives*: A grid permutation maps a vector onto a grid and permutes elements to the closest neighbour in some direction on the grid.

- *Scan Primitives*: The scan operation takes a binary associative operator $\oplus$, and a vector $[a_0, a_1, \ldots, a_{n-1}]$ of n elements, and returns the vector $[a_0, (a_o \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-1})]$. Such operators include $+$, **max**, **min**, **AND**, or **OR**.

- *Global Primitives:* The global primitives reduce the value in a vector using a binary associative operator. Applying $+$ as a global primitive produces the sum of all the elements of the vector.

- *Segmented Primitives:* The segmented primitives are useful when working on many sets of data. A vector can be broken into contiguous segments in which the beginning of each segment is marked with a flag.

In this model, routines are functions composed of the various primitives described above. These routines include pointer jumping, ordering, region summation, outer product, and histograms. As an example, a brief description of the region summation routine is given below.

For region summation [24], each pixel in a grid is required to sum a $(2m+1)\times(2m+1)$ square region around it. This procedure can be implemented using a constant number of grid scans and permutations. First, for each element, the sum of of the $2m$ pixels around it along a row is calculated using the following steps: perform a grid scan using $+$ in x, then *permute* elements at $+m$, or $-m$ offset in x in the scan result to the central element, and take their difference. Repeat this in the y-direction on the result of the first step. This procedure based on scan operations takes a constant amount of time, independent of m. Region summation and other routines can be used in the formulation of many vision algorithms. In [24], the authors show that region summation can be used to efficiently implement Boxcar convolution.

4.4 The Systolic Approach

A systolic system [19] consists of a set of interconnected processing elements (or cells), each capable of performing some simple operation. Because simple and regular structures have substantial advantages over complicated ones in design and implementation, cells in a systolic system are typically interconnected to form an array or a tree. Information in a systolic system flows in a pipelined fashion, and communication with the outside world occurs only at the "boundary cells". For example, in a systolic linear array, inputs flow into the first cell, are subjected to a series of operations, and eventually become outputs that flow out of the last cell.

This method requires that the algorithm be regular, so that each cell can do nearly an identical operation. This is not always the case in image processing, so this method has not been as widely used as other image partitioning methods. The advantage of the systolic approach is that there is no duplication of data structures between cells; each cell maintains only the data structures necessary for its stage of the computation. Also the input and output sets are not divided, so that there is no extra cost associated with splitting them up or recombining them.

Some of the image processing tasks suited to this mode of computation are one-dimensional convolution, fast Fourier transform, and relaxation which is discussed in the next subsection.

Relaxation

In some image processing algorithms, the input image is subject to multiple passes of the same operations [20]. This process is called *relaxation*, in which pass $i+1$ uses the results of pass i. A natural way to implement relaxation on a systolic array (such as Warp) is to have cell i perform pass i and send results to cell $i+1$.

Relaxation could also be implemented on Warp using input partitioning. But this requires more communication and control between cells. For example, as each cell completes a pass of the relaxation method over its portion of the image, it must communicate the results of its computation to its neighbors – both its predecessor, and its successor. This requires bidirectional communication on the Warp array, and special handling of boundary conditions. Therefore, it is better to implement relaxation by pipelining, with each cell performing one stage of the relaxation.

4.5 Summary

In this chapter, we have presented four different approaches to programming parallel vision algorithms on a variety of parallel architectures. Although all the models presented were designed for the purpose of running parallel low level vision algorithms, each model seemed to focus on a specific goal for a particular architecture, or class of machines. Thus no one model can be regarded the best, the easiest, or the fastest in a general sense. Rather, the best model for any low level vision task depends on the given problem, and the programming environment.

The Apply programming model and the Apply language have been developed for the Warp machine. The Apply model tries to minimize memory requirements at a cell by dividing images into contiguous regions, and communication requirements by mapping adjacent regions to neighboring processors. Therefore the Apply model is suitable for coarse-grained parallel machines.

The Apply language provides a level of abstraction in which programs are easier to write, and are more comprehensible. Apply also allows the programmer to get good efficiency in low-level vision programming, by incorporating expert knowledge on how to implement such operations. Apply has also been used on a number of different parallel machines, including a distributed memory machine.

Webb [33] developed another programming model based on the Apply programming model, called the split and merge model. A new language (Adapt) was also developed which is machine-independent. Unlike Apply, Adapt enables the programmer to implement global algorithms such as connected component, and histogramming.

INSIGHT is a dataflow language that can be used to program parallel vision algorithms. It allows relational expression of algorithms, provides operators that work with

sequences of data, and translates into graph structures representing configurations of an RCN machine. The RCN is a coarse-grained machine with a reconfigurable topology. The INSIGHT language has been designed to run on the RCN exploiting all the special features of the machine to provide a greater degree of control and synchronization, but also limiting the generality and portability of its applications.

INSIGHT programs have been written for both low-level, and mid-level vision algorithms, as well as some high-level vision tasks where the data can be naturally arranged in sequences. The INSIGHT language, together with the RCN have been designed for the purpose of building cost-effective machines for industrial vision applications.

In the parallel vector model, algorithms can be described in terms of primitives which are defined for a fine-grained parallel machine (the Connection Machine [14]). These primitives provide a simple and uniform way of describing algorithms. This makes the task of programming the parallel machine an easy and efficient one. This generality also does not allow the algorithms to exploit specific properties of a particular architecture, the interconnection topology, or the machine structure.

In contrast to the Apply programming model, the issues of memory requirements and communication constraints are not as critical in the parallel vector model. This is due to the fact that the number of cells is much greater for a fine-grained machine than a coarse-grained one, and each cell is responsible for the processing of one pixel of the image instead of a larger region of it. Many of the algorithms implemented using the parallel vector model had very short run-times on the Connection Machine. The Connection Machine, however, is one of the most expensive parallel computers and for that reason, can not be used in many industrial applications.

The systolic approach has been used in low-level vision applications, as well as many

other matrix computations, graph algorithms, language recognition, dynamic programming, and relational database operations.

The systolic approach relies on simple and regular structures. It can be used in a fine-grained or coarse-grained environment, although special-purpose fine-grained systolic architectures [19] provide the best performance for most applications. Using the systolic approach, no input partitioning is required, and each input data can be multiply used. Thus high throughputs can be achieved with modest I/O bandwidths.

The processing power of systolic systems comes from the parallel processing of different data elements, as well as pipelining the stages involved in the computation of each result. Data and control flows are also simple and regular in systolic systems. The only problem that might arise is the problem of synchronization of large systems since they are controlled by a global clock. Wave-front systems [21] have been proposed as a solution to this synchronization problem.

The implementation approach used here is similar to the Apply programming model. The Apply model is well suited for the transputer-based multicomputer architecture for the reasons discussed above. The C programming language is used, since an Apply compiler is not available. The implementation details are presented in the following Chapter.

Chapter 5

Software Implementation

5.1 Software development environment

Based on the mean field theory approach [10, 11], a sequential algorithm was implemented and run on a Sun-4 workstation. This chapter, however, presents the details of the parallel implementation and the parallel development environment.

5.1.1 Hardware Architecture

The complete architecture of the transputer-based multicomputer system used for the implementation is shown in Fig. 5.1. The system consists of a host machine (a Sun-4), and 17 Inmos T-800 transputers, each of which has a 32-bit 10 MIPS processor, 1 or 2 MBytes of DRAM and four 20 Mbit/sec bi-directional serial links [22].

One of the transputers is connected to the host via a VME bus interface and acts as a master node (does most of the input/output), the other 16 transputers are interconnected through programmable cross-bar switches to form a two–dimensional mesh. The interconnection topology of the transputer network can be dynamically reconfigured by having the controlling software running on the Sun send switch setting commands to the crossbar switches [17].

This two–dimensional mesh configuration has been chosen because it provides the most efficient communication pattern for this class of algorithms. This is because the image is divided into 16 subimages (approximately square[1] and of equal sizes). In this type of algorithm, each pixel has to communicate its value with its nearest neighbors. Hence, for each subimage, the processor has to communicate the borders of the subimage (top and bottom rows, left and right sides) with the four processors to which it is directly connected.

5.1.2 The Trollius Operating System

The Trollius operating system [3], [4] developed out of dissatisfaction with the standard software environment for transputer–based multicomputers. It was developed at Cornell University and Ohio State University to run on distributed memory multiprocessors. Trollius is a dynamic programming environment that is easy to use, and one that uses standard FORTRAN and C programming languages.

It consists of two parts, running on the front-end workstation and transputer nodes respectively. Trollius operates over UNIX on the host, providing a user command interface to boot the multicomputer nodes, load parallel programs to transputers, and kill processes, among other facilities.

Trollius is designed in layers of functionality, starting with message passing within a node, extending to nearest neighbor communication, then to arbitrary node to node communication at the higher level. Along with standard programming languages, Trollius provides for the inclusion of standard UNIX libraries, access to standard I/O from all processes, and dynamic memory allocation.

[1] A square's geometry provides the greatest area (number of pixels) with the least circumference (communication constraints), and hence increases the efficiency.

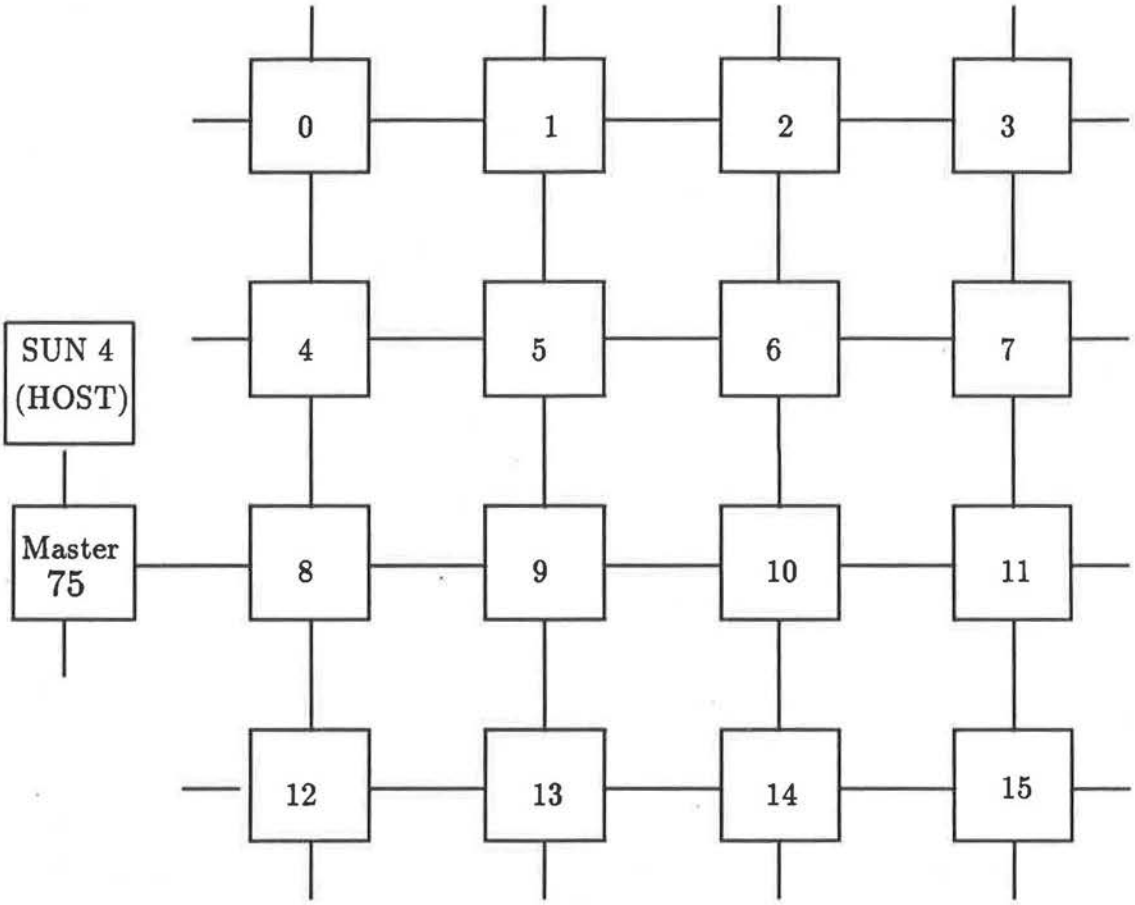


Figure 5.1: The Hardware architecture

A Trollius process sending a message does not directly specify the process to receive the message, or vice versa. Instead, each process specifies an event type in the header of the message. If the event type specified by the sending and receiving processors match, the message will be passed from the sender to the receiver. The Trollius Interprocess Communication follows a semi–blocking send/receive model. In network level message passing, the sender also has to specify the destination node of the message. Since the recipient of the message does not have to specify the source node, it is capable of receiving messages from a variety of senders.

Trollius is a topology-independent operating system which also provides Stand-Alone Trollius. Stand-Alone Trollius allows users to debug their programs on the host without tying up valuable computing resources. It can also simulate as many nodes as desired through the use of a special router on the host.

In this implementation, the network communication layer has been used for most of the communication. It provides a high degree of network transparency, but also an increasing degree of communication overhead.

5.2 The Parallel Implementation

In this section, the parallel algorithm based on the mean field theory approach is presented. The first problem that has to be considered is the one of mapping the input data to the processors of the network.

5.2.1 Mapping of images to processors

Input partitioning is natural in low level image processing. Image operations are local and regular, or produce data structures that are easy to combine. Image sizes tend to be

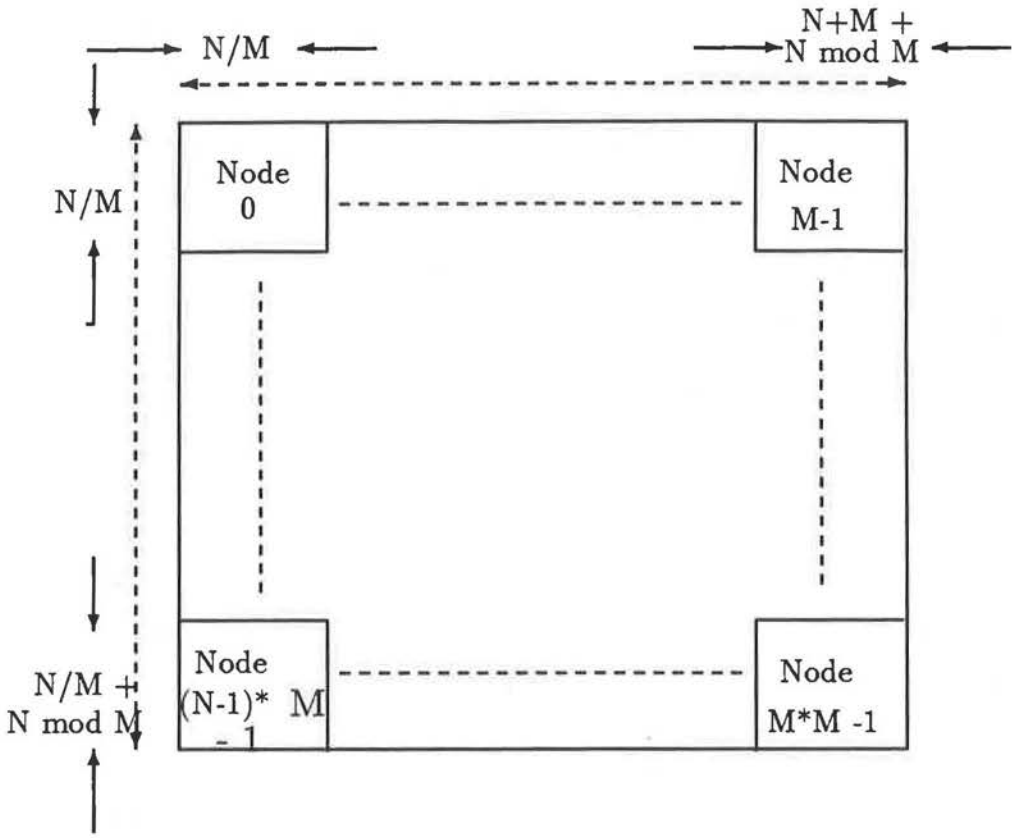


Figure 5.2: Input partitioning of an $M \times M$ image on an $N \times N$ mesh of transputers

large (for example 512×512) so that much parallelism is available even if the image is divided along one dimension (i.e., each processor gets a set of adjacent columns). Every processor does the same operation on its portion of the input. The input partitioning of an $N \times N$ image on an $M \times M$ mesh is illustrated in Fig. 5.2.

The advantages of input partitioning are its ease of programming, and the good speed-up it usually gives to the algorithm. Using a simplified cost model [20], we can calculate the speed-up from input partitioning as follows:

Suppose that the computation of the output for the complete input image on one processor takes time n. Then computation of the input on k processors takes time $n/k + kc$,

where c is the time required to combine two processors' outputs. If it is possible to know the time n, then we can choose the optimal number of processors to do the computation which is $k = \sqrt{n/c}$. Computation is wasted if the number of processors is greater than this, and as the number of processors approaches this point, adding more processors becomes less cost–effective. A cost model for the transputer-based implementation is constructed in Section 7.2.

The overhead in computation time for input partitioning comes from three sources:

- The cost of dividing the image into subimages, and distributing them across the system.
- The cost of additional bookkeeping on the part of each processor to allow outputs to be recombined later in the right order.
- The cost of recombining outputs at the end of the computation

These costs are usually negligible if the image is large and the operation is a computationally intensive one. Other potential disadvantages are the replication of data structures at each processor, wasting memory, or the fact that there may not always be efficient ways of recombining the output. In our implementation, image partitioning is done by the one transputer in the network designated the *master node*, whose role will be described in the next section.

5.2.2 The Algorithm

Our implementation follows the Apply programming model (Section 4.1) to a certain extent. Some restrictions apply when using a distributed memory multiprocessor message passing system. In this case, each processor returns a region instead of a pixel value at

the end of the computation. The programs were written in C, and compiled to run on the transputers under Trollius.

The implementation consists of two parts which we will refer to as the *master part* and the *computing part.* The Master part runs on the master transputer (labelled 75 in Fig. 5.1), and it is responsible for the following tasks:

1. **Image Input** The master node reads in the input image from the host's file system. Trollius enables any node in the network to access the file system. But giving that responsibility to the master node which is directly connected to the host helps avoid communication delays.

2. **Image Distribution** The master node does the input partitioning as described above, and distributes the subimages across the network. It sends a contiguous set of rows (or segments of rows) to the appropriate processor. This step forms the bottleneck of the computation and further work can be done to improve the performance at this stage.

3. **Image Collection** This task is carried out when the participating processors finish their computation, and start sending back their output. The master node receives a number of messages which contain the processed subimages. The messages contain enough information to enable the master to recombine them in the right order to produce the final image. Notice that the master node may be idle between the second and third steps.

4. **Image Output** This step is very similar to the first one. At this stage, the participating processors have completed their tasks, and the final image is written to a file on the host's system.

The computing part is the procedure that satisfies the Apply model requirements. The computing program runs on each one of the participating processors, and performs the following tasks:

1. **Receive Subimage** Each processor waits for the master node to send it its share of the input data. The processor receives a number of messages, each containing one row of the subimage. These rows are then stored in a 2–dimensional array to be processed. As soon as a terminal signal is sent, the processor starts the computation on its subimage.

   REPEAT THE FOLLOWING

2. **Communicate Borders** After every iteration, and before the first iteration, each processor needs to receive the borders of its neighboring processors. Every processor sends four messages to its neighbors. Each message is given an event number that matches the node id of its destination. These messages contain the updated borders of the subimage. (The number is less than four for nodes lying on the borders of the mesh). For example, processor 6 in Fig 5.1 sends the top row of the array to node 2, the bottom row to node 10, the leftmost column to node 5, and the rightmost column to node 7.

3. **Iterate on Subimage** This is the procedure where the main computation is performed on each pixel of the subimage. The mean field values of the surface field and the line process at every pixel in the image are calculated. Equations (2.13), and (2.14) are used. A lookup table is also built at every node for some of the values required repeatedly during iteration.

   UNTIL CONVERGENCE

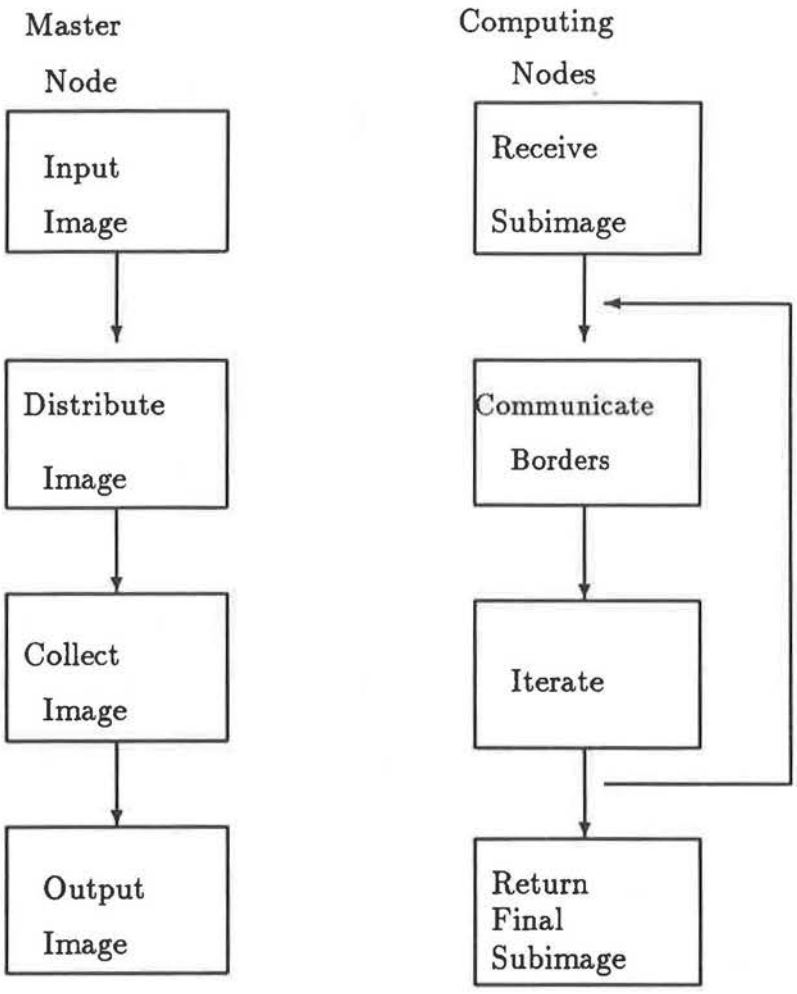


Figure 5.3: The Program Structure

4. **Return Final Subimage** Send the final subimage back to the master who by now is idle and awaiting to collect the final outputs. The rows of the subimage are concatenated to form an array of intensity values. This array is then sent as one message back to the master. The sending node also identifies itself to the master node, so that the master can place the subimage in the appropriate location.

The structure of the system is illustrated in Fig. 5.3.

Chapter 6

Results

6.1 Parameter Estimation

The parameters α, γ and ϵ must be estimated in order to develop an algorithm that smoothes, enhances, and finds the discontinuities in a given set of data. In Section 6.2, the results of running the program on a test image with different values of these parameters will be shown.

6.1.1 The parameter α

The parameter α controls the balance between the "trust" in the data and the smoothing term. The noisier are the data, the less we want to "trust" them, so α is larger. If the data are less noisy, α should be smaller. To estimate α, various mathematical methods are available. Girosi and Geiger used the Generalized Cross Validation method introduced by Wahba [32]; it states that the optimal value of α can be obtained by minimizing the functional

$$V(\alpha) = \frac{1}{n}\sum_{i=0}^{n}\frac{[f_{n,\alpha}(t_i - g_i)]^2}{(1 - a_{kk}(\alpha))^2}\omega_k^2(\alpha)$$

where $f_{n,\alpha}(t_i)$ is the smoothed solution, $\omega_k(\alpha) = (1 - a_{kk}(\alpha))/(1 - 1/n\sum_{j=0}^{n} a_{jj}(\alpha))$ and

$a_{kk}(\alpha) = \frac{\delta}{\delta f_k}(f_{n,\alpha})(t_k)$.

In their implementation [10], Girosi and Geiger use $\alpha = 4$. It was found, however, that for a value of $\alpha > 1/4$, the method did not converge for most experiments. If we look back at the simplified form of the solution:

$$\begin{aligned}\bar{f}_{i,j} = g_{i,j} &- \alpha(\bar{f}_{i,j} - \bar{f}_{i,j-1})(1 - \bar{v}_{i,j}) + \alpha(\bar{f}_{i,j+1} - \bar{f}_{i,j})(1 - \bar{v}_{i,j+1}) \\ &- \alpha(\bar{f}_{i,j} - \bar{f}_{i-1,j})(1 - \bar{h}_{i,j}) + \alpha(\bar{f}_{i+1,j} - \bar{f}_{i,j})(1 - \bar{h}_{i+1,j}) \end{aligned} \tag{6.1}$$

where

$$\bar{h}_{i,j} = \frac{1}{1 + e^{\beta(\gamma - \alpha(\bar{f}_{i,j} - \bar{f}_{i-1,j})^2)}} \qquad \bar{v}_{i,j} = \frac{1}{1 + e^{\beta(\gamma - \alpha(\bar{f}_{i,j} - \bar{f}_{i,j-1})^2)}} \tag{6.2}$$

It seems that giving α a larger value than $1/4$ gives more weight to the gradient differences than needed for the averaging process. The effect of this will be too much smoothing at the discontinuities, or equivalently propagation of noise. This is one aspect of the algorithm that can be further studied and analyzed. Further analysis of the above equations may lead to more accurate values of α required for convergence.

6.1.2 The parameter γ

From equation (2.11), we can see that $\sqrt{\frac{\gamma}{\alpha}}$ is the threshold for creating a line in the weak membrane energy. From the expression of the effective potential, we can see that if the gradient $\Delta f_{i,j}$[1] is above the threshold, there is no smoothing, and if the gradient is below the threshold, then smoothing is applied. The value of the threshold defines the resolution of the system. Once α has been estimated, a value of γ can be chosen to give the desired resolution.

[1] $\Delta f^h_{i,j} = f_{i,j} - f_{i-1,j}$, $\Delta f^v_{i,j} = f_{i,j} - f_{i,j-1}$

6.1.3 The parameter ϵ

The parameter ϵ allows the energy to be more general by controlling the amount of propagation of the line. So, once a line is created, the price to pay for another line next to it will be lowered by the amount of $\gamma\epsilon$. In other words, from the definition of E_2 (Section 2.3), we can see that the difference in the energy corresponding to the creation or not of a line at pixel $(i-1, j)$ is given by $\gamma\epsilon$. This is what characterizes the threshold and the suprathreshold, or the hysteresis phenomena [10]. The threshold is given by $\sqrt{\frac{\gamma(1-\epsilon)}{\alpha}}$ and the suprathreshold by $\sqrt{\frac{\gamma}{\alpha}}$, ϵ varying from 0 to 1. When $\epsilon = 0$, lines are created everywhere, since when a line is created, there is no cost for creating another line. The value of ϵ should be chosen to guarantee that $\sqrt{\frac{\gamma(1-\epsilon)}{\alpha}}$ is below the desired edge threshold.

6.1.4 The parameter β

The parameter β controls the uncertainty of the model. The smaller β is, the more inaccurate the model is. This suggests that for solving the mean field equations, a rough solution can be obtained for a small value of β (high uncertainty) and therefore, we can increase β (small uncertainty) to obtain more accurate solutions [10].

6.2 Implementation results

The performance of the algorithm on a synthetic noisy image is shown in Fig. 6.1. The original image consists of a square of intensity 150 on a background of intensity 100. The image is corrupted by adding random noise in the range $(-30, 30)$. The top left figure is the original synthetic image. The top right one is the noisy image, the bottom left is the reconstructed image, and the bottom right image is the edge file. The parameters used

were $\alpha = 0.12$, $\beta = 100$, $\gamma = 150$, and $\epsilon = 0.8$. It took 15 iterations for the algorithm to converge.

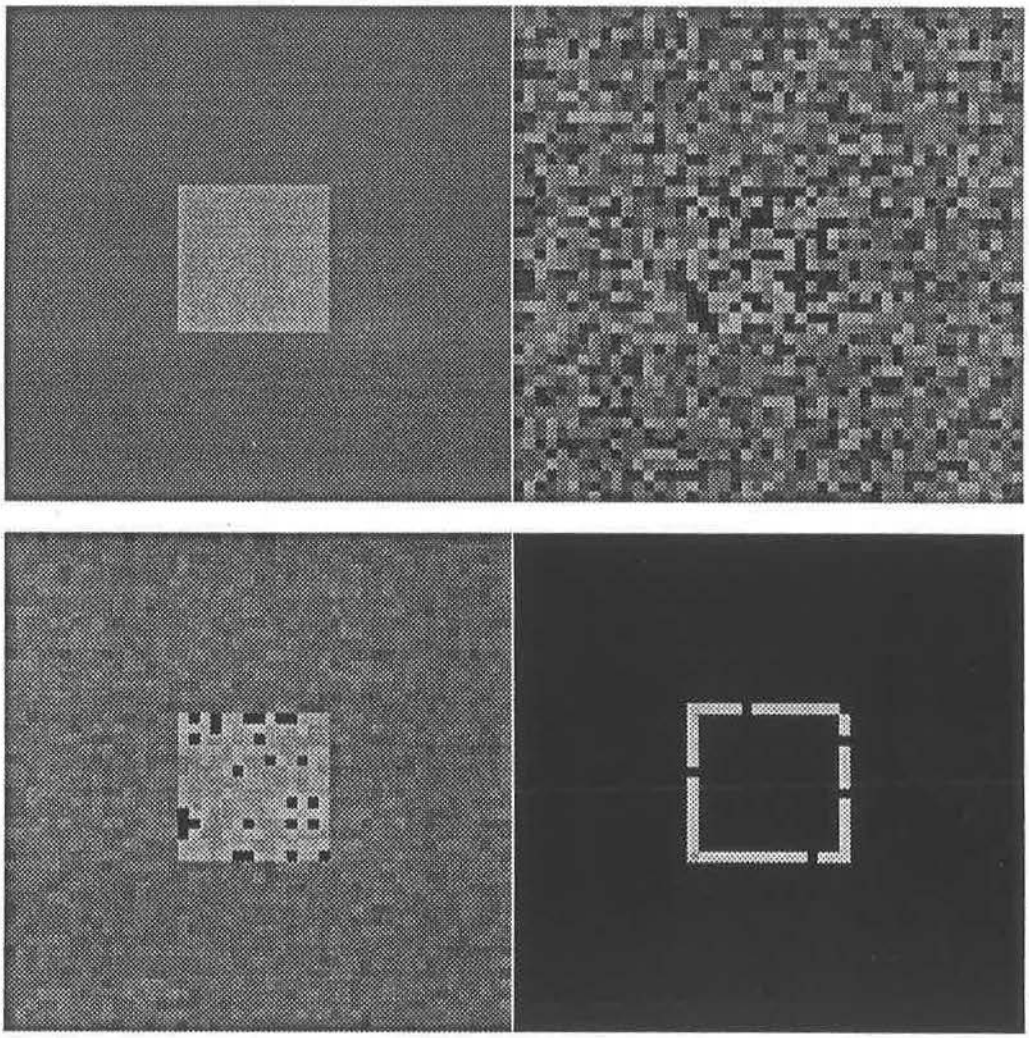
The algorithm was tested on several images. In many cases, 30 iterations were enough to achieve convergence. This, however, depended on the choice of the parameter values. Changing one or more of the parameter values could lead to divergence, or inappropriate thresholding. In fact it was found that there was a small range of the value of α for each image for which convergence was guaranteed.

Figure 6.2 shows a 512×512 image, which contained someone's hands holding a a manual drill. Figure 6.3 contains the edges obtained by running the program on the original image. We can see that the algorithm worked well at detecting the edges of the wheel where other algorithms could fail. It took less than 15 iterations and about 8 minutes on a Sun-4 to produce the edges. Notice that an ϵ value of 0 gives the weak membrane energy function.

The behavior of the algorithm was examined for a variety of different parameter choices. The program was run on 256×256 image. The original image is shown in Fig. 6.4. The edge files obtained from different parameter values are shown in Fig. 6.5.

6.3 Parallel Performance

Parallel systems are not only hard to program, but they also do not provide adequate support for users to understand the run-time behavior of their programs and detect performance bottlenecks in their applications. A new performance monitoring tool "Tmon" [5] developed at UBC has been used to monitor the performance of the programs. It is a real-time performance monitor designed to run on the transputer system. A graphical interface to Tmon has also been developed by Hilde Larsen at the department of Computer



Top Left: Synthetic image 47×47
Top Right: Noise added randomly
Bottom Left: The reconstructed image
Bottom Right: The detected edges

Figure 6.1: The Performance of the algorithm on a synthetic image

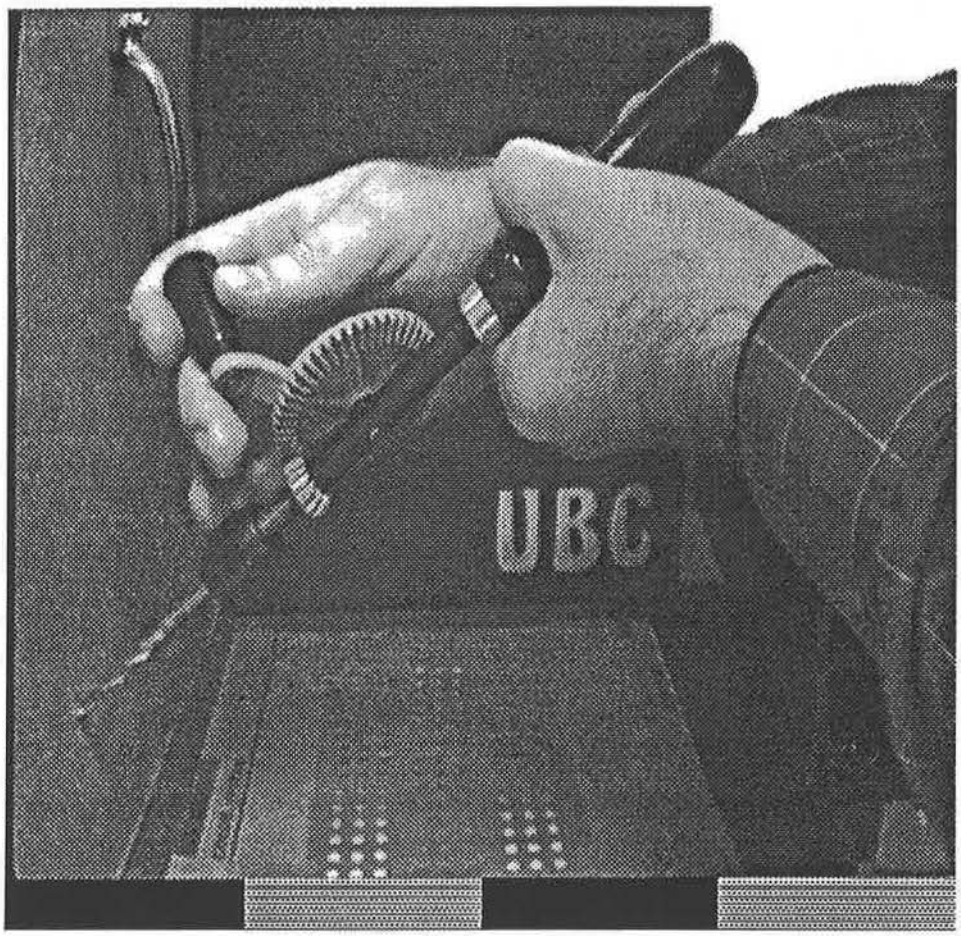


Figure 6.2: The Original 512 × 512 Image

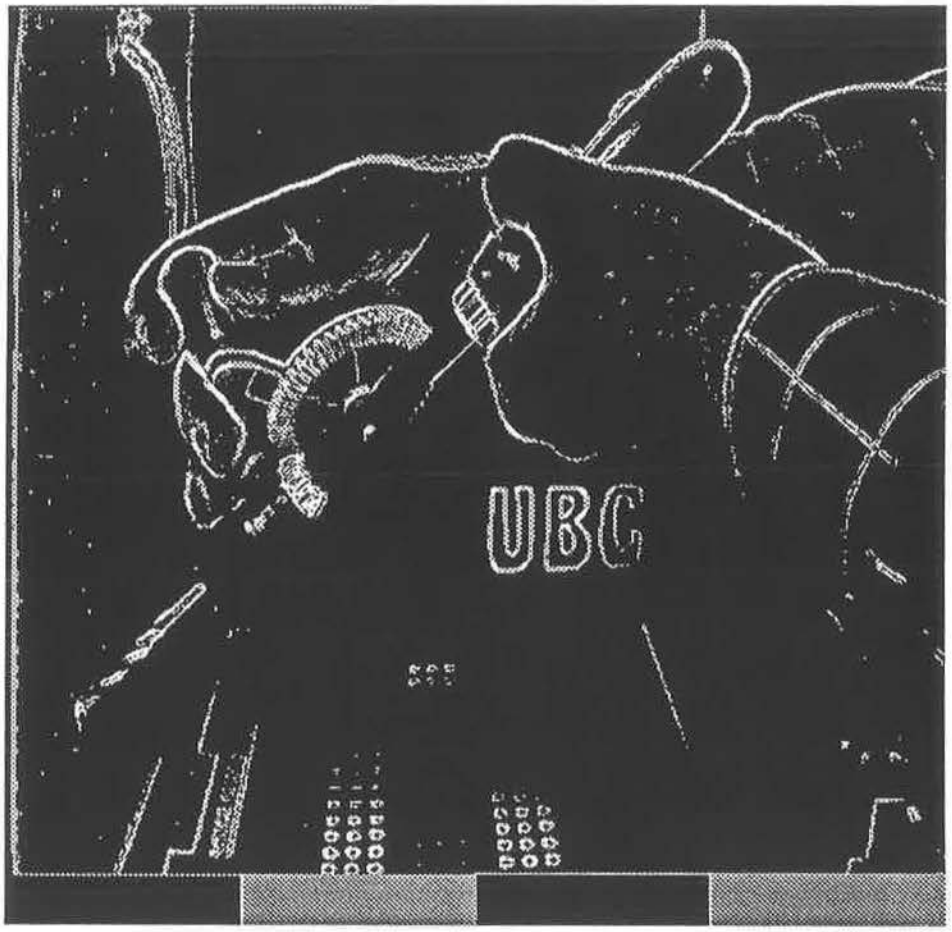


Figure 6.3: The edges; $\alpha = 0.15, \beta = 1000, \gamma = 35, \epsilon = 0$

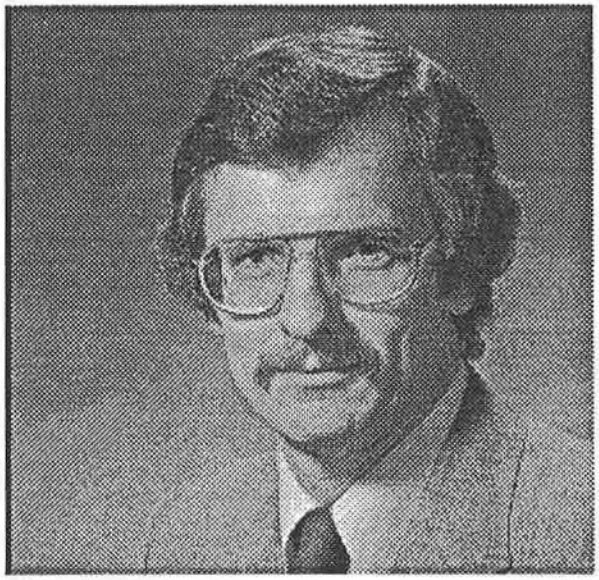


Figure 6.4: The original 256 × 256 image

Science (UBC), which made understanding the performance results much easier.

6.3.1 Monitoring Results

Tmon [5] uses a new performance analysis method called weighted critical path analysis (WCPA). WCPA incorporates parallelism into critical path analysis[2], and provides several performance metrics such as program execution time, speed-up[3], and efficiency[4].

When the monitor was used initially, with the program performing only one iteration on an image of size 64 × 64, the speed-up on 16 nodes was approximately 3, but the efficiency was less than 20%. The ratio of computation to communication in the program

[2]A critical path is defined as the path through the program that consumed the greatest amount of execution time.

[3]Speed-up is defined as the ratio $S = T(1)/T(N)$, where $T(1)$ is the time it takes to run the algorithm on one node, and $T(N)$ is the time it takes to run the algorithm on N nodes.

[4]Efficiency is defined here as the ratio of computation to communication.

(top left) $\alpha = 0.1, \beta = 1000, \gamma = 40, \epsilon =, 0.0$
(top right) $\alpha = 0.2, \beta = 1000, \gamma = 40, \epsilon = 0.0,$
(bottom left) $\alpha = 0.1, \beta = 1000, \gamma = 80, \epsilon = 0.1,$
(bottom right) $\alpha = 0.1, \beta = 1000, \gamma = 40, \epsilon = 0.8$

Figure 6.5: The edge files

was 20 : 80, which meant that 80% of the execution time was spent in the communication routines. Examination of the critical path revealed that most of the delay was caused by the communication activities of distributing and returning subimages. The critical path was divided into five segments, and the weight of each phase on the critical path was calculated. The weights are shown in the table below [5].

Phase of Computation	Weight on Critical Path
Read Image	29%
Distribute Image	5%
Process Image	8%
Return Image	47%
Write Image	11%

The input and output of the image weigh 40% on the critical path. This is due to the inherent serial nature of reading and writing the image to the host file system. Because only one transputer is connected to the host, other nodes can not obtain data directly from the host system. The processing of the subimages over the mesh weighs only 8% on the critical path. This is because a high degree of parallelism is achieved when all nodes are busy processing the subimages.

The one area where the performance of the program could be improved was at the distribution and return of subimages. The subimages were initially returned to the master node as a sequence of messages, each message containing one row of the subimage. It was later found out that in Trollius network level message passing, the header attached to each message was more than 50 bytes in size. Therefore the overhead of sending a message that is less than 100 bytes is very high (up to 40%). This problem was then solved by combining all the rows and sending the whole subimage back to the master as one message. This change in the program resulted in a 55% improvement over

the initial implementation in program execution time [5]. The ratio of computation to communication has been improved to 63 : 36, which indicates that 63% of the time is spent in effective computation tasks. Speed-up and efficiency have improved to become 6, and 40% respectively. That is, an improvement of a 100% due to the improvement of program execution time, which resulted form lowering the overhead in communication by combining smaller messages into one large message.

Graphical Interface

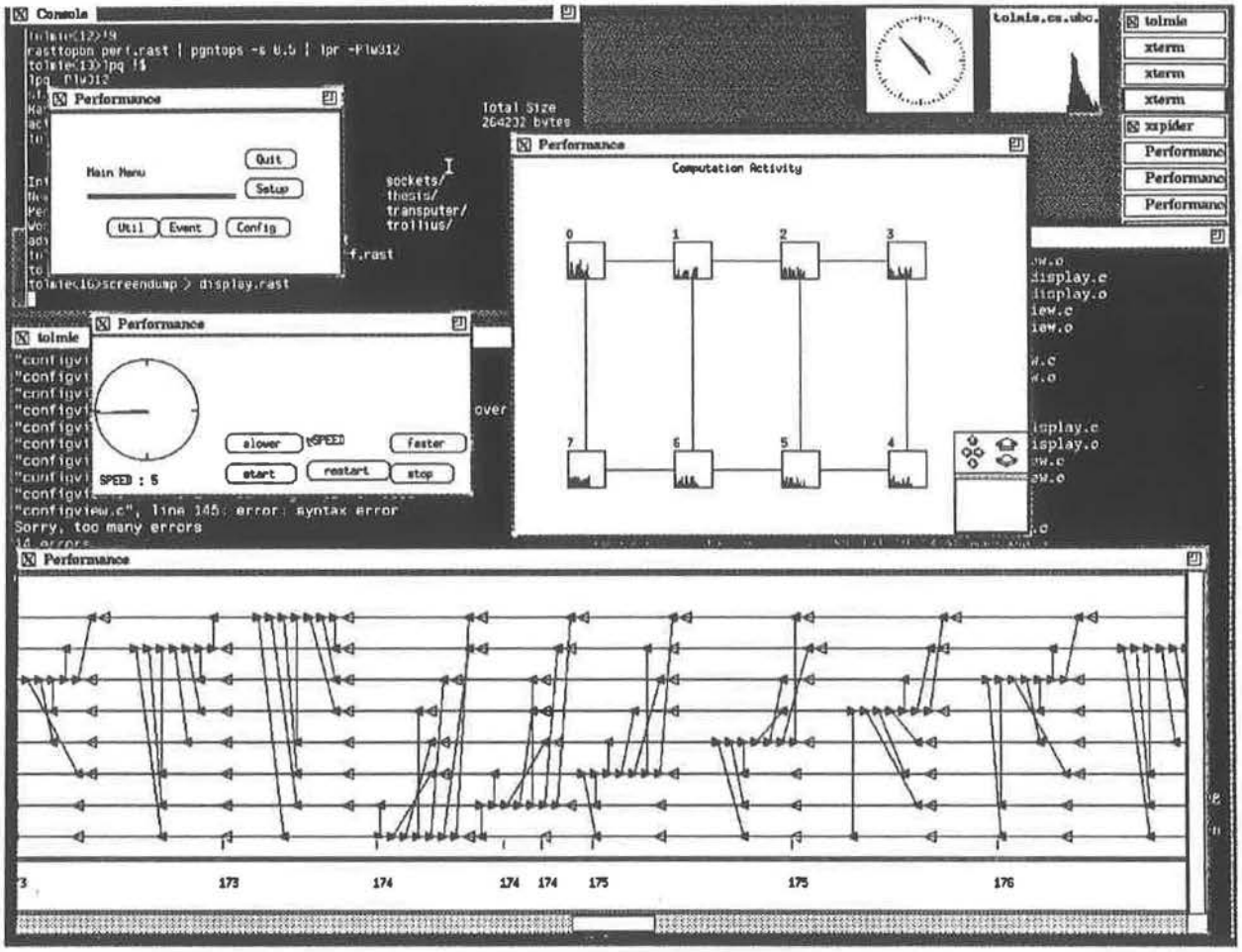
An X window-based graphical interface has been developed to display performance results to the user as easy-to-read charts and graphs. A brief description of the graphical display is given below.

The output of the graphical display includes: network topology, global clock, system load and event history. The graphical display of the programs running on the 4×4 mesh at a certain period during execution is shown in Figure 6.6. The network topology window displays the interconnection of the transputer network as a graph. The global clock window shows the current time relative to the elapsed time of the whole computation. The clock can be set, reset, started, stopped or the speed adjusted by clicking on the buttons in the window. Clicking on a node in the network topology graph will display the CPU utilization of the node selected with regard to the global clock. The event[5] history display the execution graph of the parallel program reconstructed from the event traces. The zoom in/zoom out and scrolling ability allows users to browse through the event history or focus on a portion of the execution graph conveniently. Communication patterns of the parallel program such as multicast can be easily visualized on the event

[5]An event is one of: process creation, process exit, a message sent, a message received, and a call to receive a message.

history. The user can also examine the details of each event in the execution graph by clicking on the node, which will pop up a new window with detailed description of the event selected.

The monitoring results together with the graphical interface make it much easier to debug and monitor the execution of the parallel programs.



Each horizontal line in the bottom window represents the activities of one processor. A triangle pointing to the right indicates that a message is being sent out. A solid triangle pointing to the left indicates that the node is waiting to receive a message. When the message is received, a left-pointing triangle appears in the display.

Figure 6.6: Graphical Display of the System's Performance

Chapter 7

Conclusions

In the following sections we present some characteristics and criticisms of both the mean field theory approach, and the transputer-based multicomputer as an image processing environment. In the last section, possibilities for future work to improve the system's performance are discussed. Improvements can be applied to both the algorithm and the transputer-based implementation.

7.1 The Mean Field Theory approach

The work done by Girosi and Geiger proposed a link between the statistical algorithms [12] and the alternative deterministic graduated non convexity algorithm [2]. The algorithm is a deterministic one, yet it uses Markov Random Fields in a similar fashion to the stochastic algorithms.

The algorithm was fast, and provided good results for reconstruction and edge detection tasks. The only drawback is the difficulty in estimating the parameters of the model, considering how much the solution depends on those parameters. The algorithm has the following characteristics:

- The surface field is smoothed when its gradient is not too high.

- Contrast will be enhanced at discontinuities.
- The discontinuity field is likely to be smooth (isolated discontinuities are inhibited).
- Hysteresis and adaptive multiple thresholding arise naturally from the model.
- Edge localization is good. However, edges are frequently missed if inappropriate parameters are specified.
- It is naturally extendable to the case of sparse data.
- It provides edge magnitudes (from the line process variables) instead of binary values.
- An understanding of the parameters needed to specify the model is possible.

7.2 The Transputers and Low Level Vision

The transputer-based implementation can be considered a general framework under which other low-level vision algorithms can be implemented. The only part that may have to be changed would be the ITERATE module (refer to Fig. 5.3). It is the procedure that contains the actual operation to be performed on each pixel of the image.

The transputer-based system offers a great amount of flexibility in terms of interconnection topology. This flexibility, together with the network transparency provided by Trollius make the task of programming the transputers an easy and efficient one. We have also seen from the monitor results (Section 6.3.1) that the parallelism of the network provided a speed-up of approximately 6 when 16 processors were used.

The speed-up factor can be higher for larger images distributed on a greater number of transputers. The optimal number of processors can be determined and used to achieve

the maximum speed-up. According to the cost model presented in Section 5.2.1, the maximum speed up can be obtained by minimizing the equation

$$T = n/k + kc_1 + sc_2$$

where T is the time it takes to perform the computation on k processors, n is the time it takes for one processor to do the computation, c_1 is the time required to combine two processors results (distribution and collection of data), s is the number of iterations needed to do the computation, and c_2 is the time it takes for two processors to communicate their intermediate results (communication requirements). Intermediate communication is done in parallel and is assumed to take a constant amount of time.

If we assume that it takes one time step to compute one pixel of the image, then for an image of size $m \times m$, partitioned on a $\sqrt{k} \times \sqrt{k}$ mesh of processors, $n = sm^2$. Optimal T is then achieved for a value of $k = m\sqrt{s/c_1}$. If we take $c_1 \propto m/\sqrt{k}$ (since each processor has to communicate a constant number of columns each of size $\approx m/\sqrt{k}$), then the optimal number of processors $k = (ms)^{1/3}$.

We can see from the equation above that the optimal number of processors is directly proportional to m. If we take $s = 50$ iterations, then for a 64×64 image, the optimal number of processors $k \approx 14$ processors. For a 256×256 image, $k \approx 24$ processors. For a 512×512 image, $k \approx 30$ processors. These results show that for larger images, a greater number of processors can be used to achieve a smaller run-time and hence greater speed-up.

If we disregard the time it takes to distribute and collect the data on the network, then the model can be simplified further, and we can look at the computation involved in one iteration only. In this case, the maximum speed-up per iteration can be obtained

by minimizing

$$T = n/k + c$$

where T is the time it takes to perform one iteration on k processors, n is the time it takes for one processor to do the computation, and c is the time it takes for two processors to communicate their borders (c_2 in the previous model). Again, if we assume that it takes one time step to compute one pixel of the image, and d time steps to communicate one pixel between two neighboring processors, then for an image of size $m \times m$, partitioned on a $\sqrt{k} \times \sqrt{k}$ mesh of processors, $n = m^2$, and the time $c \approx 4md/\sqrt{k}$ since each processor has to communicate a border of size $\approx m/\sqrt{k}$. These values, however, are very approximate and oversimplified. For example, the time it takes to send a message between two nodes does not simply increase linearly with the size of the message (in this case the border); rather, there is a constant overhead associated with every message (see Section 6.3.1). Thus, concatenating a group of smaller messages to form a large one is much more efficient than sending them individually.

To continue with this approximate model, minimizing T with respect to k now gives $k = m^2/4d^2$. If we take d to be one time step, then for a 64×64 image, $k \approx 1,000$ processors. For a 256×256 image, $k \approx 16,000$ processors, and for a 512×512 image, $k \approx 64,000$ processors. Therefore, by disregarding the communication with the "outside world", we notice that a massively parallel, fine grained machine provides the maximum speed-up for such an algorithm.

7.2.1 Drawbacks of the System

The transputer system under Trollius had a few drawbacks. One of the drawbacks was the congestion of messages over the links. This sometimes did not allow for dynamic

memory allocation. This was a problem only when larger images were used, and it was mainly caused by Trollius network layer message passing. The problem can be solved by direct routing of the messages. This, however, was not favourable since other facilities like the monitor, and the graphical display require that network layer communication is used.

Another drawback that appears when using smaller images is that the time to load several copies of the program (in this experiment 16) on 16 nodes is large compared to the time it takes to operate on a small image. In such a situation, it is more efficient to use a sequential version of the algorithm on a uni-processor machine.

One drawback of link communications in transputers is that communication takes place only when both the receiver and the sender are ready. This can result in the idleness of a processor waiting to communicate for as long as it takes the other processor to get ready for communication. This was observed in the graphical display of the system's execution. This aspect of the system's performance can be improved, and is discussed in the following section.

7.3 Directions for Future Work

There are possibilities for improvements and further research in both areas: the algorithm, and the implementation. In terms of the algorithm, these are:

- Analysis of the convergence of the algorithm, and the effect of changing the parameter values.

- Introduction of extra terms to the energy function, for example, one that gives the interaction between the horizontal and vertical line process.

- More sophisticated neighborhood cliques can be used to provide more information about the neighbors.

For the transputer-based implementation, these are the possible directions for improvement:

- Minimizing the processor idle time by finding the optimal communication pattern. Currently, the communication at each node is done by first sending all the messages to the neighbors, then waiting to receive the incoming messages. Alternating sends and receives in the appropriate manner could decrease the processor idle time significantly.

- The idea of activity flags was mentioned by Blake and Zisserman [2]. Activity flags are used in the later iterations to indicate whether a change in the pixel value has taken place in the last iteration, if the value has not changed, then the activity flag is switched off. If there is an activity at a certain pixel, the activity flags for that pixel and all of its neighbors are switched on. Pixels whose activity flags are switched off are not updated until the flag is on again. Activity flags speed up the computation on a serial machine, but they impose extra communication constraints in a message-passing distributed-memory machine. This is particularly true for kernel level message passing. The overhead associated with each message sent is ≈ 50 bytes. Thus sending activity flags as 1-bit arrays is still an inefficient way of communicating them.

  It would be of interest to measure the trade-off costs, and perhaps try to find a fast way of exchanging activity flags on different transputers (kernel level message passing may be fast enough).

- Building a user interface that would allow the programmer to specify an image, and an operation to be performed on that image, and according to the image size and the cost of the operation, the optimal number of transputers is chosen, and configured into a 2-dimensional mesh topology.

- Investigating the possibility of each transputer being responsible for reading its share of the data and writing results directly to the host's file system. This would result in a reduction of the reading/writing time which in our experiment (Section 6.3.1) represented 40% of the total execution time.

7.4 Summary

Based on the mean field theory approach of Geiger and Girosi [10, 11], an algorithm for image reconstruction and edge detection has been implemented. The algorithm was tested on a synthetic noisy image (Fig. 6.1), and the results show that the algorithm works well for both edge detection, and reconstruction of the noisy image. The algorithm was also tested on a real still life image (Fig. 6.2), and it can be seen that specular, shadow, and contour edges have been detected and enhanced, while the noise has been smoothed away (Fig. 6.3). In addition to that, the algorithm was tested on another image with different values of the parameters. The results (Fig. 6.5) agree with the expected performance of the algorithm after varying the parameter values (as discussed in Section 6.1).

A parallel transputer-based multicomputer version of the algorithm was also constructed and implemented on a 16-node network of transputers. A monitoring tool developed at UBC (Tmon) allowed us to monitor the parallel performance of the algorithm, and measure the speed-up and efficiency rates. The experimental results show

that a speed-up rate of 6 was obtained when 16 processors were used to run the algorithm on a small test image (Section 6.3.1).

The parallel implementation can be regarded as a prototype for many other low level vision algorithms. In fact, all the modules responsible for image partitioning, image collection, communication between the master node and other transputers, communication of subimage borders among neighboring processors, and image input/output, all of these modules can be left without any change. Only the ITERATE module which contains the computations to be performed at every pixel in the image needs to be modified to contain the new operations.

Bibliography

[1] Andrew Blake, *Comparison of the efficiency of deterministic and stochastic algorithms for visual reconstruction*, IEEE Transactions on Pattern Analysis and Machine Intelligence, PAMI 11, 1989.

[2] Andrew Blake and Andrew Zisserman, *Visual Reconstruction*, MIT Press, Cambridge, Massachusetts, 1987.

[3] M. Braner, *Trollius User's Reference*. Research Computing Center at Ohio State University and Advanced Computing Research Institute at Cornell Theory Center, Document series 2/1, January 16 1990.

[4] M. Braner, *Trollius Reference Manual for C Programmers*. Research Computing Center at Ohio State University and Advanced Computing Research Institute at Cornell Theory Center, Document series 2/2, January 22 1990.

[5] S. Chanson, J. Jiang, and A. Wagner, *Tmon: A real-time Performance Monitor for Transputer-based Multicomputers*, Computer Science Dept., UBC. To appear in the proceedings of fourth NATUG conference, Oct 1991.

[6] H. Derin, H. Elliot, R. Cristi, and D. Geman, *Bayes' Smoothing Algorithms for Segmentation of Binary Images Modelled by Markov Random Fields*, IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 6, No. 6, Nov. 1984.

[7] Shirin Eghtesadi and Mark Sandler, *Implementation of the Hough transform for intermediate-level vision on a transputer network*, Microprocessors and Microsystems, April 1989.

[8] D. L. Fielding, J. R. Beers, M. Braner, and R. Leibensperger, *The Trollius programming environment for multicomputers*, Transputer Research and Application, NATUG 3, edited by Alan Wagner, April 1990.

[9] J. A. Fortes and B. W. Wah, *Systolic Architectures: From Concept to Implementation*, IEEE Computer, Vol. 20, No. 7, July 1987.

[10] D. Geiger and F. Girosi, *Parallel and deterministic algorithms for MRFs: Surface reconstruction and integration*, AI Lab Memo 1114, MIT, April 1989.

[11] D. Geiger and F. Girosi, *Mean field theory for surface reconstruction* AI Lab, MIT, April 1989.

[12] S. Geman and D. Geman: *Stochastic relaxation, Gibbs distribution,and the Bayesian restoration of images*, IEEE Transactions on Pattern Analysis and Machine Intelligence, PAMI 6, 1984.

[13] L. G. C. Hamey, J. A. Webb, and I. C. Wu, *Low Level Vision on WARP and the APPLY programming model*, CMU Tech. Rep., Carnegie-Mellon University, Dept. of Computer Science.

[14] William D. Hillis, *The Connection Machine*, MIT Press, Cambridge, Ma.; 1985.

[15] C. A. R. Hoare, *Communicating Sequential Processes* Englewood Cliffs, N.J. Prentice-Hall, 1985.

[16] B. K. P. Horn, *Robot Vision*, MIT Press, Cambridge, MA. & McGraw-Hill, New York, NY. 1986.

[17] Jie C. Jiang and H. V. Sreekantaswamy, *Transputer based multicomputer user's manual*, Dept. of Computer Science, UBC, September 1989.

[18] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, *Optimization by Simulated Annealing*, Science 220, 1983.

[19] H. T. Kung, *Why systolic architectures* , IEEE Computer, Vol. 15, No. 1, Jan. 1982.

[20] H. T. Kung and J. A. Webb, *Mapping Image Processing Applications onto a Linear Systolic Machine*, CMU-CS-86-137, Pittsberg: Carnegie-Mellon University, Dept. of Computer Science, 1986.

[21] S. Y. Kung et. al. *Wave Front Array Processors: From Concept to implementation*, IEEE Computer, Vol. 20, No. 7, Jul. 1987.

[22] *IMS T800 transputer: Engineering Data*, Inmos, January 1989.

[23] J. J. Little, *Integrating Vision Modules on a Fine-Grained Parallel Machine*, Machine Vision: Algorithms, Architectures, and Systems, edited by H. Freeman, Academic Press Inc., 1988.

[24] J. J. Little, G. E. Belloch, and T. A. Cass, *Algorithmic techniques for Computer Vision on a Fine-Grained Parallel Machine*, IEEE Transactions on Pattern Analysis and Machine Intelligence, PAMI 11, 1989.

[25] N. Metropolis, A. Rosenbluth, A. H. Teller, and E. Teller, *Equation of state calculations by fast computing machines*, J. Chem. Phys., Vol. 6, 1953.

[26] J. Marroquin, S. Mitter, and T. Poggio, *Probabilistic Solutions of Ill–Posed Problems in Computational Vision*. J. Amer. Stat. Assoc., Vol 80, 1987.

[27] Jeffrey Mock, *Processes, Channels and semaphores* (Version 2), Pixar.

[28] G. Parisi, *Statistical Field Theory*, Addison-Wesley, Reading, Mass., 1988.

[29] Linda Shapiro, *Programming Parallel Vision Algorithms: A Dataflow Language Approach*, The International Journal of Super Computer Applications, Vol. 2, No. 4, Winter 1988.

[30] L. Shapiro, R. Haralick, and M. Goulish, *INSIGHT: a dataflow language for programming vision algorithms in a reconfigurable computational network.*, Internat. J. Pattern Recognition Artificial Intelligence, Vol. 1, 1987.

[31] D. Terzopoulos, *Visible Surface Representations*, AI Memo No. 800, MIT AI Lab Cambridge, Mass., March 1985.

[32] G. Wahba, *Practical approximate solutions to linear operator equations when the data are noisy*, SIAM J. Numer. Anal., Vol. 14, 1977.

[33] J. A. Webb, *Architecture–Independent Global Image Processing*, Proc. 10th Int. Conf. on Pattern Recognition, 1990.

[34] Alan Yuille and Davi Geiger, *A common framework for image segmentation by Markov Random Fields and Nonlinear Diffusion*, AI Lab, MIT, 1989.