A PARALLEL IMPLEMENTATION OF HIERARCHICAL BELIEF PROPAGATION

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Master of Science

by

Yuan Tian

May 2013

ABSTRACT

Though Belief Propagation (BP) algorithms generate high quality results for a wide range of Markov Random Field (MRF) formulated energy minimization problems, they require large memory bandwidths and could not achieve real-time performance when applied to many real-life inference tasks. There is an increasing demand for efficient parallel inference algorithms as the size of problems increase and computer architectures move towards multi-core. In this work, we proposed a new high speed parallel computational structure for hierarchical Belief Propagation on shared memory architecture, which is based on a modification and generalization of the hierarchical BP algorithm presented by Felzenszwalb and Huttenlocher. Our parallel hierarchical belief propagation (PHBP) computational structure supports arbitrary grouping of nodes in multi-scale computation and works for graphs in general topologies (including non grid structure graphs). Secondly, a fully parallel framework of hierarchical BP using sequential asynchronous message updating scheme (accelerated message updating) is developed. We achieved parallelization of both pre-computation portion and computational intense message passing portion. Lastly, we empirically evaluated the performance of algorithm on several computer vision tasks where we achieved nearly linear parallel scaling and outperform other alternative algorithms. Specifically, for the task of restoring a 608*456 noisy image with 16 gray levels, our PHBP takes around 100ms while a comparable result needs around 30s using Parallel Splash on a same 8 core shared memory system.

BIOGRAPHICAL SKETCH

Yuan Tian was born and raised up in a happy family in Wuhan, which is the most populous city in Central China. When she was young, she was a curious girl showing a strong fascination for drawing and building blocks. During her school years in Wuhan No.3 Middle School, she was outstanding in both mathematics and writing. Upon graduation from high school, she enrolled at Zhejiang University in Hangzhou and luckily started her 4 year journey as a member of Chu Konchen Honors College. Starting from her junior year, she joined College of Electrical Engineering and focused her study on Power Electronics and Integrated Circuit Design. With an interest in power electronic devices, she conducted related research on high breakdown voltage devices and finished her undergraduate thesis on simulation and characterization of high-current gain SiC planar BJTs from working with Prof.Kuang Sheng. During the same time, she applied and was very luckily accepted by the School of Electrical and Computer Engineering at Cornell University with the honor of Olin Fellowship. Starting from the fall of 2010, she flew half of the hemisphere to reach town of Ithaca and started her dream there.

To my dearest family and friends. Thank you for being supportive all the time.

ACKNOWLEDGEMENTS

I would like to acknowledge, particularly, the patient guidance and support from my advisor, Prof. Rajit Manohar. It is him who gave me the opportunity to start my research project on hardware implementation of Graphical Models and led me to the spectacular field of asynchronous VLSI design and architecture and walked all the way with me reaching here at this point. I also would like to give special thanks to my special committee members Prof.Jose Martinez and Prof. Tsuhan Chen for giving me precious guidance and advice on parallel computer architecture field and computer vision filed respectively. Specially, Prof.Jose Martinez taught me many about parallel programming and how to think in a multi-core system. I am especially grateful to all my committee members for offering me the precious opportunity of involvement in Intel Science and Technology Centers (ISTC) research community.

I would like to express my sincere thanks to Prof.Edwin Kan for granting me the opportunity of joining Cornell and offering me generous advice on my graduate studies. And thanks to all my fellow graduate students Benjamin Hill, Nabil Imam, Rob Karmazin, Stephen Longfield, Carlos Tadeo Ortega Otero, Benjamin Tang, Jonathan Tse, Kyle Wecker and all dear CSL members for your supports and efforts making our offices and labs a wonderful place for research and live.

Of course, my dearest mom and dad, always be the people I should appreciate most.

TABLE OF CONTENTS

Biographical Sketch iii
Dedication iv
Acknowledgements v
Table of Contents vi
List of Tables viii
List of Figures ix

1 Introduction **1**
1.1 Probabilistic Graphical Models 2
1.1.1 Probabilistic Graphical Models and Inference in Graphical Models 2
1.1.2 Energy Minimization Methods for MRF 3
1.1.3 Belief Propagation Algorithm 4
1.2 Thesis Overview 6

2 Problem Formulation **8**
2.1 Pairwise MRF Model 8
2.2 Energy Model 11
2.3 Belief Propagation Algorithm 13
2.4 Energy Minimization Using Belief Propagation 17

3 Related Work **19**
3.1 Reduce Iteration Numbers 20
3.2 Reduce Number of Message Updates Per Iteration 22
3.3 Revisit: Message Update Schedule and Message Representations 24
3.4 Hierarchical Belief Propagation 27
3.5 Parallelism Opportunities in Belief Propagation 28

4 Implementation and Results **31**
4.1 Data Structures and model construction 31
4.2 Parallel Hierarchical Belief Propagation Implementation 39
4.2.1 Message Updating and Graph Partitioning 39
4.2.2 Parallel Algorithm 43
4.3 Experiment Setting 51
4.4 Results 52
4.4.1 Speed Analysis 52
4.4.2 Computation Time Breakdown 52
4.4.3 Comparison of Parallel Hierarchial BP with Other Parallel BP 56
4.4.4 Quality Analysis 62
4.5 Discussion 62

5 Conclusion **66**

A Co-processor design for Belief Propagation **68**
A.1 Introduction 68
A.2 Baseline Design 70
A.3 Alternative Designs 70
A.3.1 Overview and breakdown of belief propagation algorithm 71
A.3.2 Architecture design of Co-processor 72
A.3.3 processor/coprocessor interface 73
A.3.4 processor and coprocessor/memory interface 74
A.3.5 Data Flow driven Message Updating Engine 75
A.4 Evaluation: Initial Results 77

Bibliography **81**

LIST OF TABLES

4.1 Computation time breakdown for processing one frame of video input using 8 processors. First two columns are the execution results from parallel hierarchical BP using asynchronous message updating (PHBP-A) with three levels and three iterations of message updating per level (L3I3). The next two columns are for PHBP-A L3I2. The last two columns are results from PHBP-A L3I1. 55

4.2 Comparison of number of updates for image restoration task of "sun" image. 61

4.3 Error in disparity estimation for Middlebury dataset. 62

A.1 Comparison of cycle counts and number of instructions for baseline bypass processor design and alternative coprocessor design in different array lengths (which is L shown in Algorithm 7). This shows how coprocessor design is scaling for applications of different complexities. 78

LIST OF FIGURES

2.1 Pairwise MRF (a very common graphical model used for vision applications). The joint probability over all variables factorizes into a form like $P(X \mid Y) = \frac{1}{P(Y)} \prod_p \phi_p(x_p, y_p) \prod_{(p,q)} \psi(x_p, x_q)$ 10
2.2 A diagrammatic representation of computation of message vector $m_{pq}(f_q)$ sending from node p to node q. The "min" showing next to source node p indicates that we are using the minimum component of RHS (over f_p) as the newly calculated message value. 15
2.3 A diagrammatic representation of computation of belief vector $b_p(f_p)$ at node p: adding together all incoming messages $m_{sp}(f_p)$ and the data vector $D_p(f_p)$. 16

3.1 A diagrammatic representation of updating two sets of messages alternatively at iteration t (shown in left) and at iterative $t + 1$ (shown in right) for a bipartite graph. 23
3.2 An illustration of grid structure graphs at level i (shown in left) and at level $i + 1$ (shown in right). Every super-node at level $i + 1$ is a group of 2×2 nodes from level i. 28

4.1 An example of initializing message vectors at level i from coarser level $i + 1$. Every father node at level $i + 1$ has 2×2 children nodes from level i. The correspondence between father node 1_{i+1} and children nodes $1_i, 2_i, 5_i, 6_i$ are shown with arrowed pointers. The arrows which represent messages are showing how the message vectors ("right" and "down") at level i are inherited from coarser level $i + 1$. For example, message vectors from 1_i to 2_i, from 2_i to 3_i, from 5_i to 6_i and from 6_i to 7_i are initilized with the message vector values from 1_{i+1} to 2_{i+1} calculated at level $i + 1$. 35
4.2 An illustration of a 16-node graph's linked list representation at level i (shown in left) and at level $i + 1$ (shown in right). Every super-node at level $i + 1$ is a group of 2×2 nodes from level i. . . 36
4.3 A noisy image where a large portion are expected to have similar label values. 37
4.4 An example of aribitrary grouping of a grid structure graph from level i (16-node, shown in left) to level $i+1$ (5 super nodes, shown in right). The nodes $p_i \in \mathcal{P}_i$ at level i are shown as white circles. The nodes $p_{i+1} \in \mathcal{P}_{i+1}$ at level $i + 1$ are shown as dashed circles. These super nodes at level $i + 1$ have different numbers of children nodes. 37

4.5 An example of computation work partition in a grid structure graph. Left image is showing two processors responsible for processing one row in the original graph respectively. Right image is showing a certain part of the adjacency list which is going to be written to by a specific processor. $p1$ and $p2$ denote processor 1 and processor 2 respectively. 38
4.6 One iteration for two processors using round robin asynchronous message updating belief propagation (BP-A) in grid structure. Every iteration is consisted of four phases, where messages in "right", "left", "up", "down" directions are updated respectively. For first two phases where messages in "right" and "left" direction get updated, we cut the grid structure graph in x direction so that the distributed work for two processors are independent. Similarly we cut the grid structure graph in y direction for "up" and "down" phases. 42
4.7 An illustration showing how adjacent nodes are tagged with DirectionKey. Adjacent nodes shown in yellow are tagged with a same Direction Key signifying that the directions of messages sent from their source nodes to them are the same (or we "define" them to be the same). 45
4.8 An illustration showing how adjacent nodes in different levels are tagged with Direction Key out of a direction key set. Adjacent nodes shown in yellow are tagged with a Direction Key signifying that the messages sent from their source nodes to them are in "right" while green stands for "down". 46
4.9 Speed-up for parallelized hierarchical belief propagation with round robin (sequential) asynchronous message updating (PHBP-A). Asynchronous message updating is performed in an up-down-left-right manner. HBP-A is used as a sequential baseline here. 53
4.10 Computation time breakdown for processing one frame of video input using 8 processors. All results are obtained from PHBP-A. 55
4.11 Image segmentation task with different number of iterations. First one is the input. Results from PHBP-A with L3I3, L3I2, L3I1 are shown from (b) to (d). 56
4.12 Energy of stereo solutions as a function of running time. Both PHBP-A and PBP-A are parallel versions working on 8 shared-memory processors. 57
4.13 Image restoration task with "sun". (a) noisy picture. (b) clean picture. (c) Our outputs with PHBP-L3I2-P8. Average message update counts per processor is 363888. (d) Outputs with Parallel Splash when set residual to 0.02. Average message update counts per processor is 636780. 59

4.14 Outputs of four stereo pairs from Middlebury datasets. From top to bottom: Tsukuba, Teddy, Cones, Venus. Three columns are outputs of parallel HBP-L3I1-P8 (hierarchial belief propagation with 3 levels and1 iteration per level, 8 processors), outputs of parallel HBP-L3I2-P8, and corresponding ground truth respectively. 60
4.15 Image segmentation experiment for frame29 and frame35 images. Column one is original scene captured by Kinect. Column two is the output using PHBP-A (Parallelized HBP with Asynchronous Message Updating) output performed with 3 levels, 3 iterations/level (L3I3) on 8 processors. Number of labels is set to 3: 0 stands for hands(target), 1stands for body, 2 stands for background. 61
4.16 (a) is showing the Non-occluded regions (white) and boarder regions (black) for the Venus image. (b) is showing the distribution of bad pixels (with an absolute disparity error larger than 1) in the PHBP-P8-L3I1 output of Venus. 63

A.1 A diagram representation of message engine with val-rdy interface. 79
A.2 A diagram representation of interface between processor, coprocessor and memory. 79
A.3 Data flow graph for message engine. Circles represent logic or arithmetic operators. Rectangles represent operators with states. Rounded rectangles represent token(rdy,val) flow operators. . . 80

CHAPTER 1

INTRODUCTION

With probabilistic graphical models serving as a powerful visual representation of statistical dependencies between variables, problems in different fields such as computer vision, artificial intelligence, speech and image process can all be reformulated as the computation of marginal probabilities on graphical models. This computation process is often referred to as inference. The belief propagation (BP) algorithm is an efficient way to solve these problems which would give an exact solution when the factor graph is a tree structure, while lead to approximation outputs when the it is a cyclic graph. Though graphical model based global methods often generate high quality results, they usually takes long time to converge. For classical synchronous BP, for example, we often need to set the number of iterations to be equal to half of the largest dimension (of the graph) to get a reasonable result. In order to generate high quality outputs as fast as possible, we would like to speed up inference tasks in graphical modes without losing accuracy through designing an effective parallel computation structure. As computer architectures move towards multi-core era, a large number of applications are gaining advantage from exploring the parallelization possibility lying inside their computation expensive portions. It would be very interesting to see how much performance improvement we can gain from doing parallel inference in graphical models, which would make it possible to perform inference tasks in real time and finally to map those applications onto mobile or embedded devices.

In this thesis, I focus on exposing the parallelism to iterative Belief Propagation algorithms which can be represented by passing messages (partial marginal probabilities) in graphs. Especially, I present a parallel approach for applying belief propagation algorithm to graphs hierarchically. This parallel hierarchical BP computation works in two steps: first is to build up a multi-scale graph set based on the original graph that was given to infer in a parallel manner; Second to apply parallel BP to each graph out of the graph set in a coarse to fine manner and finally finalize the outputs on the original finest graph. This parallel implementation is designed to provide an algorithm that can be executed in clusters of computers or multiprocessors in order to reduce the total execution time. For the rest of this chapter, I would first briefly review the important concepts in Probabilistic Graphical Models and belief propagation algorithms, then I would provide a road map of this thesis.

1.1 Probabilistic Graphical Models

1.1.1 Probabilistic Graphical Models and Inference in Graphical Models

Probabilistic graphical models, a diagrammatic representation of probability distributions, is a very useful way of representing the statistical relationships between random variables of a concrete problem. A graph comprises nodes connected by edges. In a probabilistic graphical model, each node represents a random variable (or a group of variables), and each edge represents probabilistic relationship between variables. There are two types of graphical models. The first one is directed graphical models (Bayesian networks), in which the links carrying particular directions express causal relationships between random variables, and the second major class is undirected graphical models, which is often known as Markov random fields (MRF), in which the edges express soft constraints between random variables.

Given a graphical model, the most fundamental task (which is also quite challenging) is to compute the marginal distribution of variables or subsets of variables. This task is often referred to as inference. In an inference problem, some of the nodes in a graph are clamped to observed values, while we wish to compute the posterior of some subsets of other nodes.

As a powerful visualization of dependencies and an useful facility to obtain factorizations of the probability distributions, graphical models are widely used in various problem domains, such as statistical physics, computer vision, error-correcting coding, artificial intelligence and general optimization.

1.1.2 Energy Minimization Methods for MRF

During this section, we are going to take stereo vision problem as an example and explain how a corresponding energy minimization problem would be built and solved. Stereo technique is based on finding a correspondence between the pixels of two (or several) images taken from different view points. This is called the correspondence problem and an optimization process needs to be applied in order to find the best correspondence between pixels. In a stereo matching problem, we are expected to find out the best label value for each pixel given all the information we collect from two (or several) images.

The stereo matching problem can be solved with Markov random fields

(MRF) based models. Generally, MRF models provide a robust framework for early vision problems, including stereo, optical flow and image restoration [1]. Unfortunately, most problems built on MRF models usually lead to an NP-hard energy minimization problem in the end. The development of energy minimization algorithms for pixel-labeling tasks is one of the most exciting advances in early vision field. Although it has been known for decades that these early vision problems such as depth or texture computation can be efficiently expressed using Markov random fields, the resulting energy minimization problems is usually intractable [2]. During the last decade, two inference algorithms - Belief Propagation (BP) and graph-cuts have been proposed and studied to get an approximate solve for this NP-hard optimization problem. These methods are demonstrated to be powerful in the sense that they lead to a low minimized global energy value over ”large neighborhoods” and in the sense that they produce accurate results in various benchmarks. Despite the decent results belief propagation and graph-cuts could get, both methods are too slow for practical use. This comes from the fact that global methods are inevitably computational expensive and thus too slow for real-time applications. We will talk more about the speed problem in the parallelization motivation section.

1.1.3 Belief Propagation Algorithm

Belief Propagation is a widely recognized method for solving graphical model inference problems. It is applied to two different types of situations: (1) to maximize the marginal probabilities for every variable using the minimum mean square error(MMSE) estimator, and (2) to estimate the best (most probable) states of all variables in the problem (compute the maximum a posteriori (MAP) estimator). These two algorithms are often referred to as *sum-product* algorithm [3] and *max-product* (or *min-sum* when working with the negative logarithm of probabilities) algorithm respectively. The Belief Propagation[4] algorithm is a local message passing algorithm which would converge to a fixed point on graphs with no loops (for example, tree structure) while would provide an approximated (but often good) solution on graphs with loops. Through message passing in max-product BP algorithm the most probable values of the unobserved variables given the observed ones can be obtained through assignment based on the fixed (approximated) points.

The main characteristic of the Belief Propagation algorithm is that the inference is calculated using iterative message passing between nodes. Each node sends and receives messages until a stable situation is reached. Messages, locally calculated by each node, comprise statistical information concerning one node's neighbor nodes.

To tackle the problems in early vision, a scheme of running BP on grid structure (which is a cyclic graph) needs to be used. This approach is known as loopy Belief Propagation, which is an approximate inference in graphs with loops. The idea is to apply the sum-product algorithm even though there is no guarantee of yielding good results. This approach is possible because the message passing rules for the sum-product algorithm are purely local. However because the graph has cycles, information can flow many times around the graph. For some models, loopy belief propagation would converge at some point, whereas for others it will not. BP algorithm has been provided to be efficient on tree structures; And many experiments have shown good approximate results for some applications when applying loopy BP to cyclic graphs such as grid structures.

In order to apply loopy belief propagation to problems containing cyclic graphs, we need to define ”pending” messages [3]. We will say there is a ”pending” message on the edge from node *a* to node *b* if node *a* has received any message on any of its other edges since the last time it send a message to *b* . For graphs having a tree structure, any message schedule that only sends pending messages will eventually terminate once a message has traveled across every edge in both directions. Because there are no pending messages, the product of the received messages at every variable would give the exact marginal. However, for graphs containing cycles, the algorithm may never terminate because there might always be pending messages. For most applications, loopy BP is generally found to converge within a reasonable iteration times, or once it has been stopped when reaching the stopping criteria, the (approximate) local marginals can be computed using the most recently incoming messages to each node and generate (approximate) outputs.

1.2 Thesis Overview

The focus of this work is to provide a general parallel computational structure supporting the effective hierarchical BP computation for general graphical models. The rest of our discussion is organized as below. Chapter 2 shows how a real-world computer vision problem can be formulated as a labeling problem using pairwise Markov Random Field framework. Specifically, Chapter 2 guides readers through the process of defining energy functions for pairwise MRFs and using the iterative belief propagation algorithm to minimizing the defined energy function. Chapter 3 reviews existing research efforts for speeding up BP from different perspectives. Some recent research have been focusing on exploring the scheduling of messages in BP algorithms which has a large influence on both convergence speed and convergence possibility, such as residual message updating proposed by Elidan et al.[5], residual splash for parallelizing BP proposed by Gonzalez et al. [6]. While some other research are more focusing on reducing the memory storage and bandwidth requirements of BP which limit the performance of BP in hardware implementations. These efforts include storing beliefs at each node rather than on edges proposed by Larsen et al. [7], efficient message representation presented by Yu et al. [8] and etc. Chapter 4 starts with a description of our representation of graph models, and then presents how our parallel hierarchical Belief Propagation is going to work on MRFs. Specifically, we present how graph partitioning is performed in our parallel computational model using sequential asynchronous message updating and show how this computational structure supports general graph and arbitrary grouping of nodes in multi-scale computation. Finally through testing with several real-world tasks in computer vision, Chapter 4 evaluates the performance of this parallel algorithm on cyclic graphical models where it achieves linear parallel scaling and outperform alternative implementations, which demonstrates that our parallel hierarchical BP is a good candidate for real-time applications. Chapter 4 also discusses and compares different optimizations for BP. Chapter 5 concludes our work and possible future efforts.

CHAPTER 2

PROBLEM FORMULATION

2.1 Pairwise MRF Model

Many low-level computer vision (early vision) problems are about estimating some spatially varying quantities such as disparity or intensity from the data we are given. Problems such as image segmentation, image restoration or disparity estimation in stereo can all be formulated as labeling problems in the MRF framework. The labels correspond to quantities that we are desired to estimate at each node, such as intensities for image restoration and disparities for motion or stereo.

Throughout this thesis, we would focus on early vision problems, where a pairwise MRF provides attractive theoretical models. Typically only pairwise MFR are used for these problems because considering more neighbors quickly makes inference on MRF computationally intractable. Although the compatability functions are defined over two neighboring nodes in a pairwise MRF, each node is still able to influence all nodes in the MRF [9]. To take a concrete example, we would describe a max-product belief propagation algorithm for stereo vision problems, such as computing the depth or disparity of pixels in a reference image by matching them with pixels in the supporting image(s) capturing the same scene from different positions.

In the pairwise MRF model, the set of nodes V is comprised of two subsets: $Y = \{y\}$ representing the observed quantities corresponding to every node consisting the image and $X = \{x\}$ representing hidden quantities corresponding to every node consisting the underlying scene, which is the labeling f we are trying to resolve. The set of edges E represent the dependencies between pairs of these nodes (variables). For each pixel p or a small patch of pixels in the image, there would be a corresponding observed node y_p and a hidden node x_p, with $(x_p, y_p) \in E$. All hidden nodes in $\{x\}$ are connected in a grid structure thus we have $(x_p, x_q) \in E$ over any two nearest neighbors p and q in the grid structure. For these two types of node pairs (edges) in E, we have defined *local evidence* $\phi(x_p, y_p)$ representing the joint probability of hidden node x_p and its corresponding observed node y_p, and *compatability function* $\psi(x_p, x_q)$ representing the joint probability of two neighboring hidden nodes x_p and x_q. The MRF is said to be "pairwise" because this compatability function is only dependent on pairs of positions p and q. Writing p, q as a short hand for edge (x_p, x_q), the overall joint probability of a scene X and an observation Y would be

$$p(\{x\}, \{y\}) = \frac{1}{Z} \prod_{p} \phi_p(x_p, y_p) \prod_{(p,q)} \psi(x_p, x_q) \tag{2.1}$$

where Z is a normalization constant. We can consider the observed nodes $Y = \{y\}$ to be fixed and write $\phi_p(x_p)$ as a short hand for $\phi_p(x_p, y_p)$. Then the joint probability for the set of hidden variables $X = \{x\}$ can be written as

$$p(\{x\}) = \frac{1}{Z} \prod_{p} \phi_p(x_p) \prod_{(p,q)} \psi(x_p, x_q) \tag{2.2}$$

A graphical depiction of this model is shown in Fig 2.1. The shaded nodes represent the observed quantities $Y = \{y\}$, while the white nodes represent the hidden quantities $X = \{x\}$ we are trying to infer. This MRF is pairwise because the compatability functions only exist between pairs of node x_p and node x_q.

We can define the *energy function* $E(\{x\})$ using Boltzmann's law in statistical physics $p(\{x\}) = \frac{1}{Z} e^{-E(\{x\})/T}$. In our context, the "temperature" T is just a parame-

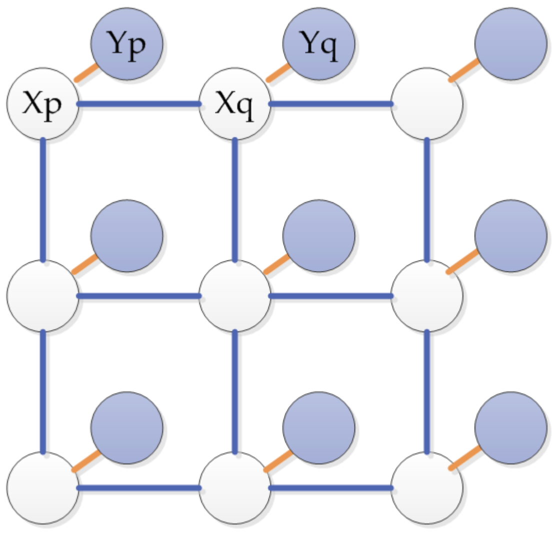

Figure 2.1: Pairwise MRF (a very common graphical model used for vision applications). The joint probability over all variables factorizes into a form like $P(X \mid Y) = \frac{1}{P(Y)} \prod_p \phi_p(x_p, y_p) \prod_{(p,q)} \psi(x_p, x_q)$

ter that changes the scale of units for the energy, and for simplicity, we choose our units and set $T = 1$ [10]. Energy function can also be viewed as the negative log probability of the postereior distribution of an MRF. The energy of a pairwise MRF can be expressed as

$$E(\{x\}) = -\sum_p ln\phi_p(x_p) - \sum_{(p,q)} ln\psi(x_p, x_q). \tag{2.3}$$

In this context, $\phi_p(x_p)$ and $\psi(x_p, x_q)$ can be viewed as *potentials*.

In this MRF framework, each node x_p can take one of k values (one of k discrete states), which are often called *labels*. These labels are in accordance with the properties that we are trying to solve in a specific task. In stereo matching, for example, the labels stand for possible disparities at one point and we would have $f_p \in \{0, 1, \cdots, k-1\}$, which is the full set of all possible label values. We are expected to compute the most probable assignment of labels for every node x_p given $\phi_p(f_p)$ representing the probability that node x_p is labeled with f_p and $\psi(f_p, f_q) = \psi(x_p = f_p, x_q = f_q)$ representing the probability that node x_p is labeled with f_p while node x_q is labeled with f_q.

2.2 Energy Model

As shown from the previous section, an energy function can be defined corresponding to every problem formed in the MRF framework. Now we are going to focus on the MRF defined in terms of energy functions. Let's denote the set of nodes (hidden nodes to be solved) as $\mathcal{P}$ and the set of labels as $\mathcal{L}$. Also, we would use p (or q) to denote one node in $\mathcal{P}$. The goal is to find a labeling f that assigns a label $f_p \in \mathcal{L}$ to every node $p \in \mathcal{P}$. The best labeling (most probable state of all variables) is found by maximizing a joint probability shown in 2.2, which becomes minimizing the energy function which can be viewed as negative log of joint probability. Thus optimal label assignment f can be obtained through minimizing the energy function given by

$$E(f) = \sum_{p \in \mathcal{P}} D_p(f_p) + \sum_{(p,q) \in \mathcal{N}} V_{p,q}(f_p, f_q) \tag{2.4}$$

where $\mathcal{N}$ is the set of all neighboring node pairs in the graph. $D_p(f_p)$ is a function derived from observed data representing the cost of assigning the label f_p to the node p, which is often referred to as the data cost. In other words, D_p measures how well label f_p fits node p given the observed data for p. For example, in image restoration, the labels represent gray levels and $D_p(f_p)$ is normally $(f_p - I_p)^2$, where I_p is the observed gray level intensity of p. In stereo problems, the labels are disparities and the data term $D_p(f_p)$ is some function of the intensity difference between the pixel p in the primary image and the pixel $p + f_p$ in the comparing image. $V_{p,q}(f_p, f_q)$ measures the cost of assigning labels f_p and f_q to two neighboring nodes p and q, and is often referred to as the discontinuity cost.

In vision problems, at the borders of objects, adjacent pixels should often have different labels and it is important that energy E does not over-penalize such labellings. Finding the labeling with a minimum energy in an appropriately defined MRF is the goal of our problem.

Generally, we can represent the energy function as the the sum of data energy E_{data} and smoothness energy E_{smooth} with

$$E(f) = E_{data}(f) + E_{smooth}(f). \tag{2.5}$$

Properties in early vision problems such as disparities or depth levels often vary smoothly almost everywhere, but change dramatically at object boundaries. Intuitively, data energy E_d measures the disagreement between the labeling assignment f and the observed data, and smoothness energy E_s coming from the negative log probability of the prior represents the extent to which our labeling assignment f is not piecewise smooth.

To find the labeling assignment for Equation 2.4, we first need to determine the forms and parameters for $V(f_p, f_q)$, $D_p(f_p)$ respectively. In stereo problems, for example, we use the following data cost function for a pixel $p = (x, y)$,

$$D_p(f_p) = min(|I_l(x, y) - I_r(x - f_p, y)|, t), \tag{2.6}$$

where t denotes a truncation value for data cost. For discontinuity cost, we can choose $V(f_p, f_q) = T(f_p \neq f_q)$ where $T(x)$ is defined to be 1 when x is true and 0 if x is false, which is often referred to as *Potts Model* [11]. The Potts Model captures the piece-wise smooth assumption of labels for neighboring node: we penalize two neighboring nodes choosing different labels. Another class of discontinuity cost functions are dependent on the difference between two labels. Since in some applications the label values may not smoothly changing everywhere, the cost function should become constant at some point as the difference of labels between neighboring nodes becomes large. By doing so, our energy function would not over-penalize such not smoothly changing labellings such as at the borders of objects. One example of such cost function is the truncated linear model, where the cost is proportional to the difference between two labels f_p and f_q up to the truncated value,

$$V(f_p, f_q) = min(s|f_p - f_q|, d), \tag{2.7}$$

where s represents a scale factor for the discontinuity cost, and d represents the truncation threshold value.

In the next section, after converting problems to the corresponding MRF, a powerful approximate inference algorithm, loopy belief propagation can be used to approximate the posterior probability.

2.3 Belief Propagation Algorithm

Loopy Belief Propagation applies to graph with loops. BP is an exact inference for graphs without loops. For cyclic graphs, such as the hidden variables organized in grid structure as shown in Fig. 2.1, loopy BP is not guaranteed to give the global optimal solution, but it serves as a good approximation inference which runs in a linear time proportional to the total number of hidden nodes.

Belief Propagation algorithms specific to Bayesian networks, pairwise MRF's and factor graphs (consisted of nodes representing variables and nodes representing functions of variables) have all been developed and these different versions are all mathematically equivalent [10]. The difference between them arises from the the fact that in factor graph there are two types of nodes and thus two types of messages, and in Bayesian networks arrows lead to different kinds of messages. We would focus our discussion on applying Belief Propagation to pairwise MRF without loss of generality.

The max-product BP algorithm is an efficient way used to find a low cost labeling for a specific energy function in MRF framework. This algorithm is defined to maximize a joint probability distribution for the unknown variables x_p shown in Equation 2.2. When we are working with negative log of probabilities, an equivalent computation can be performed, where the max-product BP becomes a min-sum BP. Energy function shown in equation 2.4 is in the negative log domain and the min-sum BP can be performed directly to find the labeling f which gives the minimum energy cost.

In the BP algorithm, we introduce variables such as $m_{pq}(f_q)$, which can intuitively be understood as a "message" information from a hidden node p to its neighboring hidden node q about what state (label) node q should be. A *message* in Belief Propagation is a reusable partial sum for the calculation of marginal probability. The message vector $m_{pq}(f_q)$ has the dimensionality of node q's label value set (number of possible states), with each element proportional to how probable node p thinks node q should choose the corresponding label. The max-product BP works by passing messages around graph. m^t_{pq} denotes the message from node p to node q at time t. In negative log domain, all $m^0_{pq}(f_q)$ are initialized to zero (all messages are initialized to uniform distributions). The message vector m^t_{pq} is defined over each label f_q by

$$m^t_{pq}(f_q) = \min_{f_p}\left(D_p(f_p) + V(f_p, f_q) + \sum_{s\in\mathcal{N}(p)\backslash q} m^{t-1}_{sp}(f_p)\right) \tag{2.8}$$

where $\mathcal{N}(p)\backslash q$ is all the neighboring nodes of node p except node q. All mes-

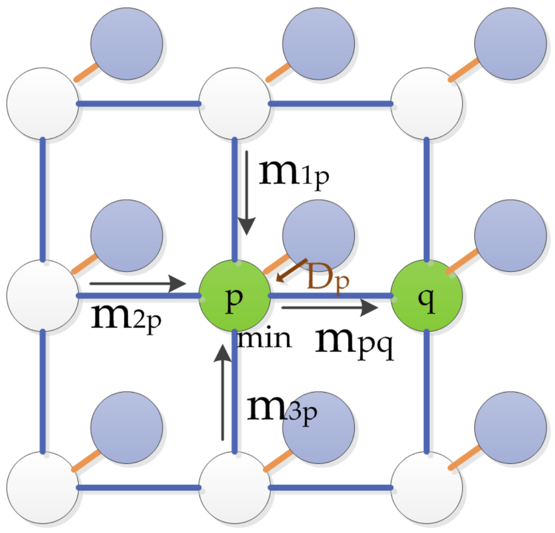

Figure 2.2: A diagrammatic representation of computation of message vector $m_{pq}(f_q)$ sending from node p to node q. The "min" showing next to source node p indicates that we are using the minimum component of RHS (over f_p) as the newly calculated message value.

sages are defined self-consistently by this update rule. A graphical depiction of message update rules in BP algorithm is shown in Fig.2.2.

In BP algorithm, estimated marginal probabilities are called *belief*s. The belief at node p is proportional to the sum of data cost $D_p(f_p)$ (a reflection of the local evidence at that node $\phi_p(x_p)$)and all messages coming into that node. After T iterations of message updating, a belief vector b_p for each node $p \in \mathcal{P}$ is computed through

$$b_p(f_p) = D_p(f_p) + \sum_{s \in \mathcal{N}(p)} m_{sp}^T(f_p) \tag{2.9}$$

Finally, the label f_p^* corresponding to the minimal element of vector b_p is selected as the minimum cost label for node p. A graphical depiction of the computation of belief vector $b_p(f_p)$ for node p is shown in Fig 2.3. $D_p(f_p)$ represents the cost of assigning label f_p to node p given its observed quantities at y_p. In this grid structure, each node p has four neighbors, corresponding to four messages

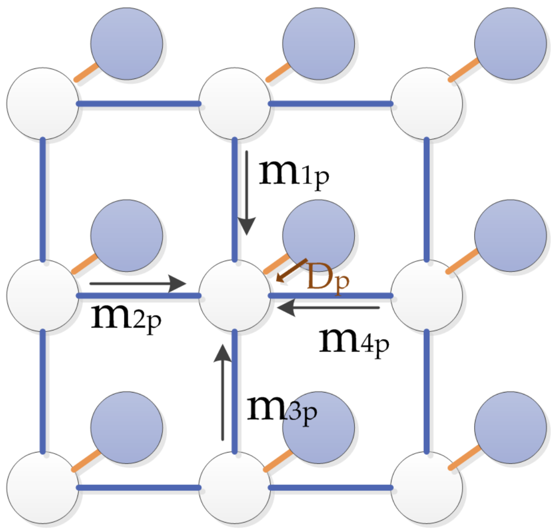

Figure 2.3: A diagrammatic representation of computation of belief vector $b_p(f_p)$ at node p: adding together all incoming messages $m_{sp}(f_p)$ and the data vector $D_p(f_p)$.

coming from 4 directions to node p.

In conclusion, BP algorithm works in two steps: (1) update messages until convergence (or stopping criteria) (2) calculate beliefs for each node.

As mentioned before, given observations in a certain problem, we are desired to find the state of some hidden variables, where we can choose from optimizing for maximizing the joint posterior $p(X|Y)$ (maximum a posteriori (MAP) estimator), or for the marginal posterior of each hidden variable $p(x_s|Y)$ respectively (minimum mean square error (MMSE) estimator), or for some other point estimator. The max-product BP algorithm we have shown here are trying to optimize for the MAP. As shown in [9], sub-pixel accuracy can be obtained through using sum-product BP (MMSE solution) in disparity or motion estimation problems. Both max-product BP and sum-product BP can be used in different contexts for different applications.

2.4 Energy Minimization Using Belief Propagation

Using the fact that for a wide range of models built for various applications, including Bayesian networks (directed graphical model), error-correcting codes and factor graphs can all be converted into a pairwise MRF framework to define an energy function, BP can be efficiently used to minimize the energy function. The success of BP is exciting because that it systematically and efficiently handles many different kinds of problems which seemed to be difficult, involving graphs with many nodes and loops [10]. The approximation to energy that BP is effectively implementing is more accurate and sophisticated than "mean-field" approximations, providing a principled framework for propagating information between nodes in a graph.

The standard implementation of max-product BP runs in $O(Nk^2T)$, where N is the number of nodes to be resolved, k is the number of possible label values for each node, and T is the number of iterations. During every iteration, it calculates $O(N)$ messages. In grid structure, for example, there would be $4N$ messages (up, down, left, right) per iteration. For every message, it takes $O(k^2)$ time to compute Equation 2.8, where we need to calculate the message component corresponding to every f_q while visiting k (number of possible f_p's) possible values. This is much better than the brute-force calculation of marginal probability, which has a complexity of $O(k^N)$.

For some specific discontinuity functions we are able to decrease the computational complexities shown in $O(k^2)$. For example, for discontinuity cost function using truncated linear model as shown in Equation 2.7, Felzenszwalb and Huttenlocher [1] proposed that every message can be computed in $O(k)$

time instead of $O(k^2)$ time through computing messages in two passes over the full set of labels. This optimization is applicable because for most early vision problems, the discontinuity cost is only relevant to the difference between two labels f_p and f_q rather than the particular values of (f_p, f_q).

CHAPTER 3

RELATED WORK

In this chapter, we are going to review past research efforts towards the goal of speeding up belief propagation algorithms and mapping BP to hardware for real time performance. Some research have been conducted to decrease the computation complexity (or data redundancies) in BP algorithm thus decrease iteration times while some works are trying to minimize the memory and bandwidth requirement which would be promising for further adaption to real-time hardware implementations. Ideally we would like to incorporate all possible optimizations and improvements into our final parallel implementation. As mentioned in the previous chapter, the overall complexity of BP algorithm is $O(Nk^2T)$, where N is the number of nodes to be resolved, k is the number of possible label values for each node, and T is the number of iterations. Therefore, from the computationally complexity view, we can work from three aspects (N, k, and T) to decrease the running time. By distributing the computation work to several processors, we are actually trying to decrease the number of nodes N for each processor. We should always bear in mind that in order to get the most speedup from this parallelization implementation, our distributed calculation framework should not hurt iteration time T too much and should produce a reasonable output (have a convergence). In order to get a better understanding of how different optimizations would affect both the performance and quality of the algorithm, we are going to review recent research efforts towards speeding up Belief Propagation algorithm.

3.1 Reduce iteration time T: message update schedules lead to fast convergence.

From the time dimension perspective, we can try to decrease the iteration time T term in the complexity given by $O(Nk^2T)$. This can be achieved through optimizing the message update schedule. Ideally, we are looking for a message update schedule which leads to convergence quickly for a certain graph. The message update schedule determines when a message sent to a node would be used by that node to compute messages for the node's neighbors.

In synchronous message updating (BP-S), all messages are updated in parallel. Once every node has computed the messages for all its neighbors(4 messages for a 4-neighbor grid structure) in one iteration, all the messages produced in this iteration are used in next iteration to generate new messages sending from each node. For iteration times, it is noted in [9] that setting the number of iterations to be equal to half of the largest dimension is suitable. Synchronous message updating BP would be denoted as BP-S in the successive discussions.

Sequential asynchronous message updating (BP-A), which is also referred to as accelerated message updating because of its fast convergence in chain and grid structure, is an efficient schedule for a serial processor. The most important property of BP-A is that a newly computed message can be used to compute another message immediately (or inside current iteration) and do not need to wait until next iteration. In grid structure, for example, message updating can work in this manner: (1) The first node in a row, p would send a message to node at its right, $p + 1$. (2) Node $p + 1$ would use this message immediately, along with the messages it had previously received from above and below, to compute a new message to node $p + 2$. (3) Once this has been completed for every row, the same procedure occurs in the right, up, down direction.We can see in this update scheme each constructed message is immediately consumed in building the next message in the same direction. Here we can define that one iteration is consisted of four phases: update all right messages, update all left messages, update all down messages, update all up messages.

The advantage of BP-A is that information (messages) is quickly propagated across long distances in graphs. For a synchronous update schedule on an image with width W (pixels), theoretically it would take W iterations for information from one side of the image to reach the other. However, the asynchronous schedule would only require one iteration to converge very quickly.

Another advantage of BP-A is that the we can break the whole graph into pieces and process pieces sequentially since intuitively messages are propagating sequentially. BP-A enables the exploration of the possibility of exploiting data reuse in hardware implementations. In tile based BP proposed in [12], they only keep boundary messages in-between processing tiles to reduce memory and bandwidth cost (originally, memory requirement for BP is $O(Nk)$, bandwidth complexity is $O(NkT)$). Because boundary messages in one direction would always be reused by the successive tile, we can keep these messages in on-chip memory and save 25$ bandwidth when switching tiles. Besides, a sequential order of message updating is also perfect for pipelining in architecture designs[12] and can reduce the total running time of BP.

One last thing to note is that strict synchronous message updating means that we need to double the storage space for messages because we need to hold both message values at iteration t and at iteration $t+1$, which is too expensive for hardware implementations. On the opposite, in asynchronous message updating scheme, every newly calculated message value would be written to a certain memory location immediately and message values from last iteration would always be overwritten. The checkerboard alternative updating shown in Fig.3.1 is not a pure synchronous message updating since not all messages get updated during every iteration. Due to the fact that we are always updating different message sets at iteration t and the successive iteration $t+1$, we do not need to worry about the overwritten problem. We just need allocate storage space accommodating all messages (no need to double the space). However, even we can manage the storage problem using checkerboard updating, we still need an iteration number in the order of the size of image to reach convergence.

3.2 Reduce Number of Message Updates Per Iteration

In order to reduce the number of message updates per iteration (which is reflected in N in $O(Nk^2T)$), we are trying to decrease the number of messages essential to drive the information flow. The idea behind this is that we are expecting to drive the flow of information in BP when discarding some messages.

It was noted by Felzenszwalb and Huttenlocher [1] that for a bipartite graph BP can be efficiently performed through updating two sets of messages alternatively. In a bipartite graph, all nodes can be split into two sets so that every edge is connecting two nodes belonging to different sets. If we denote two sets of nodes as A and B, any messages is either sent from $x_p \in A$ to $x_q \in B$ or from $x_q \in B$ to $x_p \in A$. Given messages sent from $x_p \in A$ at iteration t, we can calculate all messages sent from $x_q \in B$ at iteration $t+1$, and then again we can calculate

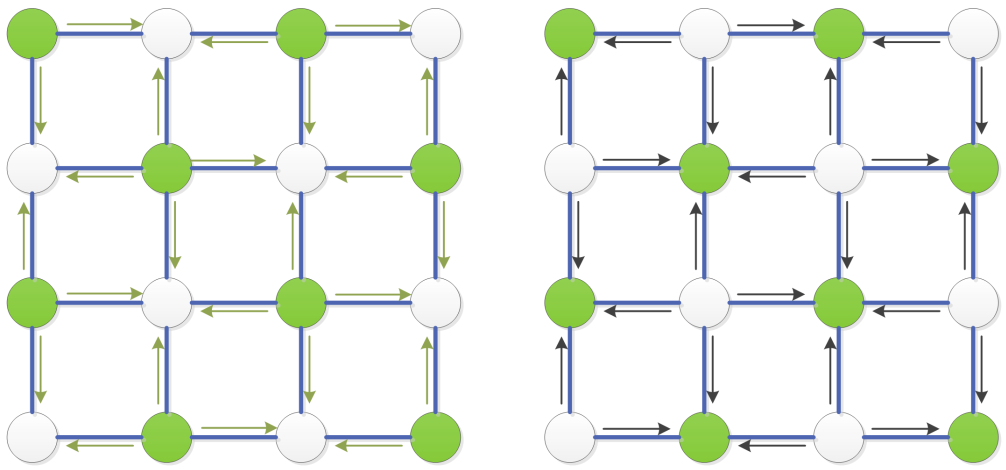

Figure 3.1: A diagrammatic representation of updating two sets of messages alternatively at iteration t (shown in left) and at iterative $t + 1$ (shown in right) for a bipartite graph.

all messages sent from $x_p \in A$ at iteration $t + 2$ without calculating them at iteration $t + 1$. This "checkerboard" message updating scheme changes the total number of messages needed to be updated from $O(N)$ to $O(N/2)$ without hurting the total iteration number T. Two plots in Fig. 3.1 are showing how we are alternatively updating messages from iteration to iteration. One thing to notice is that the bipartite property can also be expanded to block/cluster based MRF, which means that we can if we have break the large graph into blocks (clusters of nodes), we can process in a checkerboard manner on block level, while inside each block (cluster), we can still use sequential message updating or other message updating schemes.

Larsen et al. [7] proposed to store beliefs (which has been compressed to 2 values at each node) at each node rather than message vectors on the edges between pairs of nodes. In this work targeting for handling strong occlusion effects in multiple-view reconstruction, the belief probability density functions has been compressed into a single pair of values to represent the current best estimate label value for that node and confidence in that label value. Though this simplified approximate formulation of BP is targeting for occlusion, it also illustrates the idea of compressing computation work.

Ogawara[13] proposed to average outgoing messages from each node into one message vector to reduce both computation time and the required amount for memory. Both computation time and memory had been reduced at the price of loss of accuracy when evaluated against the Middlebury stereo benchmark.

3.3 Is there an optimal sequential message updating order which benefits both convergence speed and number of necessary message updates?

As we can see from previous discussions, these two types of approaches focusing on convergence speed and amount of computation respectively always have effects on both properties. Or in other words, these optimizations or estimations which try to change (reduce) the number of message updates (computation) every iteration in BP usually also have some influence on the total number of the iterations (convergence speed), and vice versa. This means that we should look at how to benefit both convergence speed and computation (or definition) of messages. One natural thought would be observing how messages from nodes in different regions of the image are changing and evolving from initial values to converged values and how different scheduling would affect the convergence speed while giving different amount of computation.

For example, in order to pursue the ″optimal″ sequential message updating order, the residual splash belief propagation[6] is giving higher updating priority to a node which has produced larger amount of new information (sending out meaningful messages) during last update (vertex residual) and schedule message updates in the whole graph according to the vertex residual order. They first show that there exists a lower bound of convergence for chain graphical models. Then they generalizes it to general cyclic graphical models, where they proposed splash operation on a tree centered at one node to mimic the optimal sequential ordering for chain structure. This generalization aims at improving convergence speed through pursuing an ″optimal″ sequential ordering of message updating for cyclic graphs. As is shown in the original paper of residual BP [5], when compared with round robin asynchronous message updating and sequential tree-reweighted message updating which was first proposed by Wainwright in [14], residual message updating, where message update order is determined as the algorithm process, lead to a decrease in the number of message updates (calculations) by 40 60 percent in their crafted models in different sizes. This means that residual based message updating (dynamic asynchronous message schedules) can approximately cut the calculation work by half when compared with static (round robin and tree-reweighted) asynchronous message updating scheme. However, there are several reasons why residual based dynamic message updating (including residual splash BP) may not be favored for inference tasks in embedded device where both memory requirement and speed are important concerns. First, we could not ignore the fact that this dynamic message schedule would lead to an increase in both calculation work of residuals and memory requirement, also the maintaining of priority queen is probably also linear to the scale of graph. If this is unfortunately the case, the advantage of ambitiously doing message updating for ″important″

nodes may not be that apparent. Second, as we would show in experiment results in late chapters, despite the fact that we can cut down the total number of message updates by around half (still the same order), the contribution to speeding up convergence is not comparable with (in the same order as) what we could achieve through using hierarchical BP. Plus, the exploration of parallelism for dynamic message updating in the work parallel residual splash belief propagation [6] shows that the maintaining of a priority queue is necessary for multiple processors working together, which is apparently much more complicated and expensive than an implementation which makes use of the natural parallelism lying in round robin asynchronous message updating scheme. We would illustrate more details in respect to this comparison in Section 4.2.2.

Lastly, one noted difficulty for implementing BP (especially in hardware such as GPU, FPGA) is the large memory requirement for handling a large number of message vectors. Quantitatively, the memory size needed for storing all messages would be $O(s * N * k*)$, where s is the size of neighborhood (which would be 4 in the case of a grid structure), N is the number of nodes (which would be number of pixels for early vision applications) and k is the number of labels (size of the label set). There are also some research conducted on simplifying or altering each message vector. For instance, in order to reduce the intra-message redundancies, Yu et al. [8] applied compression techniques to the messages in BP algorithm to both save memory space and reduce read/write bandwidth. It is noted [8] that through applying Envelope Point Transform (EPT) and predictive code to messages in BP algorithm, 8 times of compression can be achieved without much loss of depth accuracy for dense stereo reconstruction applications. The advantage of doing EPT is that we have flexible control of compression ratio, however, EPT itself is a nonlinear operation so linear operations such as message addition cannot be carried out directly in the compressed domain. Another example is simplifying message updating computation through reducing the number of labels in some of message vectors (targeting for k shown in $O(Nk^2T)$) , which is referred to as constant space belief propagation [15]. In their proposed algorithm, they hierarchically reduce the number of labels (size of label sets) as the spatial resolution increases.

3.4 Hierarchical Belief Propagation

Different from previous discussions focusing on optimizing for computation on the original graphical model built from problem directly, Felzenszwalb and Huttenlocher proposed to build a set of coarser graphs based on the problem-level graph and do computation and estimation in a coarse to fine manner. This hierarchical (multi-scale) BP scheme updates messages in coarse levels (smaller graphs), then pass message values down to finer levels (larger graphs) and use computational results from coarse levels as initialized values for the finer levels. This hierarchical BP scheme dramatically decreases the total number of iterations T thus contributes to the running speeding of BP. Intuitively, all messages are updated in parallel at every iteration. This implies that it would take many iterations for information to travel large distances in the grid structure since it can only propagate one "step" (edge) every iteration.

In hierarchical BP scheme, we started with computing and propagating messages in the coarsest level I. After T iterations the resulting messages calculated at level i are always used to initialize messages at level $i-1$ (finer level) until we reached convergence at level 0, which is the original node-based graph.

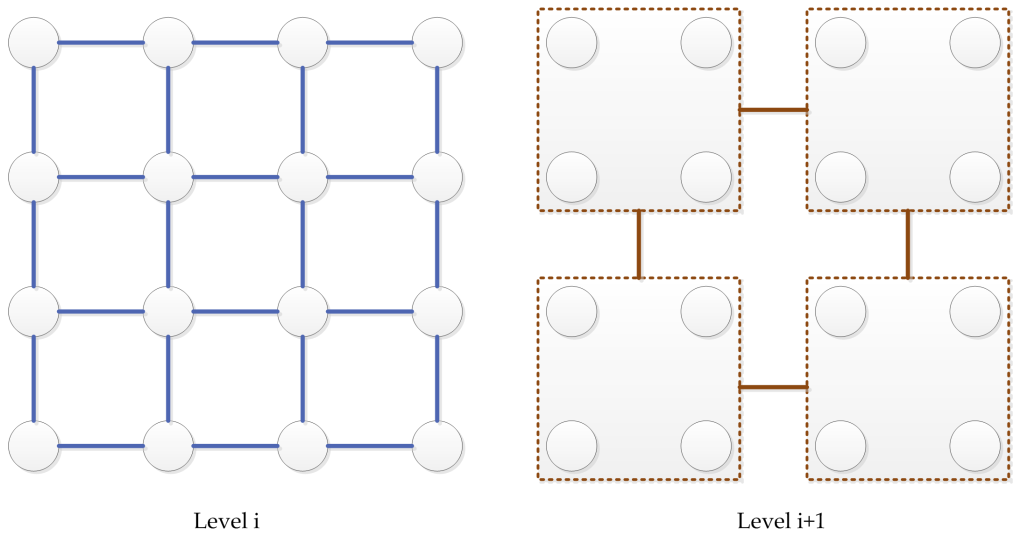

Figure 3.2: An illustration of grid structure graphs at level i (shown in left) and at level $i + 1$ (shown in right). Every super-node at level $i + 1$ is a group of 2×2 nodes from level i.

This would make messages converge faster since messages in finer levels are initialized closer to the fixed point they are expected to converge to. Intuitively, this construction helps us to achieve convergence much more rapidly because long range interactions inside the original grid graph can now be replaced by short distance propagation in coarser levels, where messages are propagated in smaller scale graphs consisted of groups of nodes. Fig. 3.2 illustrates two levels of grid structure graphs when grouping $2^i \times 2^i$ nodes into one super-node in each level i. We would revisit this hierarchical BP scheme give a more detailed description of it in 4.

3.5 Parallelism Opportunities in Belief Propagation

The convergence speed of inference tasks for probabilistic graphical models is still a practical challenge in large domains, especially for mapping certain applications to mobile devices where real time performance is often desired. As computer architecture transition towards ever increasing parallelism, we would like to explore the possibility of parallelism at fundamental level in the inference algorithms.

Ideally we would like to expose parallelism throughout different applications using graphical models by developing a few core parallel algorithms. Graphical models can also be viewed as a common language for representing statistical models in a various application domains. Inference, the process of calculating probability distributions in the model, is the primary computationally intense portion in reasoning and learning in graphical models. Thus through providing an efficient parallel implementation for inference algorithms, we could expose parallelism to a wide range of different applications such as machine learning, computer vision and image processing. As discussed before, while there are many popular inference algorithms (including graph-cut and belief propagation), belief propagation is one of the most popular one. Therefore, an efficient parallel algorithm for the message-passing belief propagation would provide a solid basis for various inference tasks in graphical model.

Ideally, a good combination of approaches we have discussed in previous sections could be applied to pursue the best convergence speed while maintaining a relatively small memory and bandwidth requirement. Different from all the optimizations which focus on either a specific type of message schedule or an optimized/estimated message expression and representation, the focus of our work is to find an optimal parallel framework which nicely describes a general graphical model for a given problem, leads to the best performance of BP running on graphical models while also suitably accommodating all these optimization efforts we have discussed here.

Based on all the discussion, hierarchical computational model of BP which reduces the complexity of the problems and leads to a dramatical performance improvement for different inference tasks from real world is an ideal candidate for serving as the baseline algorithm for a parallel implementation. This hierarchical computation model is general enough for different graphical models, especially when the size of problems goes up, doing multi-scale computation can always makes a larger problem faster to solve. Also we can always choose to apply different message updating scheduling and message representation/computation optimizations inside the computation of graphs from each level. Thus, we would keep the focus of this work to be providing a general parallel computational model supporting the hierarchical BP for general graphical models.

In next chapter, we would talk about how how a graphical model can be built in our framework, how we are building a parallel framework of hierarchical BP and how different considerations of optimization have been incorporated.

CHAPTER 4

IMPLEMENTATION AND RESULTS

We propose a modification and generalization of the hierarchical BP algorithm presented by Felzenszwalb and Huttenlocher [1] which targets for four-connected grid structure graphs (early vision problems). First, our implementation of hierarchical belief propagation is based on adjacency list representation of graphs, which (1) allows for arbitrary grouping of nodes in multi-scale computation through linking father node (in a higher level) with its children nodes (in a lower level), (2) would work for graphs in any general topologies (including non grid structure graphs) since the graph in each level is a direct description of nodes (**node** structure) and its neighbor nodes (**adjacent node** structure) and does not require every node to have a same number of neighbors (which is true for grid structures). Second, we adopt sequential asynchronous message updating scheme (accelerated message updating) which has been shown to be good for further expansion to hardware implementations. Third, a fully parallel framework of hierarchical BP is developed. This achieves (1) parallelization of pre-computation parts node initialization, edge initialization, message initialization in every level, (2) parallelization of message passing scheme in different message update schedules. Lastly, we would like to do a comparison between our proposed implementation and other implementations (specifically, we would look at residual based Parallel Splash BP).

4.1 Hierarchical BP Built on Adjacency List

An algorithm is essentially manipulation of data in a suitably chosen data structure. The underlying data structure is a vital foundation in the design and analysis of algorithm implementations. Making good use of structural properties of certain data structures would yield efficient implementations for various problems. BP is performed on MRF in our context, thus an efficient representation for nodes (which are associated with data costs) and its neighborhood (which are associated with the messages sent out from this node) in undirected graph is needed.

In this section we would first briefly review the hierarchical max-product BP implementation [1] which originally targets for four-connected grid structure graphs (early vision problems) was implemented in their work. Then we would demonstrate how adjacency list structures would provide a solid foundation and generalization ((1) irregular node grouping and (2) general graph structures) for this hierarchical BP scheme.

In hierarchal BP, when we group several nodes p_i at the finer level i into a group of nodes b at the coarser level $i+1$ (an example grouping of four nodes into one super-node on a grid structure is shown in Figure 3.2) the data cost of assigning a label f_b to super node b at level $i+1$ can be expressed as

$$D_b(f_b) = \sum_{p_i \in b} D_{p_i}(f_b). \tag{4.1}$$

We would refer this cost computation for every super node b at coarser levels as **node initialization** in our context. A summation of costs is equivalent to a product of probabilities. Thus the interpretation for D_b can be the probabilities of all children nodes p_i choosing the label f_b. Let's denote the finest level, which is the pixel based graph as level 0 and the coarsest level as I. After node initialization from level 1 to level I, the hierarchical BP would start the message computation shown in Equation 2.8 in level I where all messages have been initialized to zero. After T_I iterations of message updating in levels I, all resulting message values at level I would be used to initialize message values at the next finer level $I-1$. The same **message initialization** and message updating computation would be performed in every level below I until we reach convergence at level 0. For four-connected grid graph, there are four types (directions) of messages sent from each node: $down, up, left, right$. When we denote these messages sent to four directions as d, u, l, r, messages sent from node p_i at level i are inherited from messages sent from super node b at level $i + 1$ following

$$\{ \text{l, r, d, u} \}^{0}_{p_i}(f) = \{ \text{l, r, d, u} \}^{T_{i+1}}_{b}(f). \tag{4.2}$$

where l^t_p represents a message vector that node p sends to its left at iteration t, and similarly r^t_p, d^t_p, u^t_p represents the message vector sent right, down and up, respectively.

In the original implementation, at each level, five 2-dimension arrays in the same size ($cost, down, up, left, right$) are used to represent data cost vectors corresponding to every node and four message vectors sending from every node. For example, $cost[p]$ represents the data cost vector of node p, $cost[p][f_p]$ represents the data cost value when node p chooses the label f_p, $down[p]$ represents the message vector sent from node p to the node sitting below it (let's denote this node as q) and $down[p][f_q]$ represents the message value sending from p to the node q when the label value for q is fq. This implementation is valid as long as the grouping or correspondence between every father node and its children nodes are following the same rule in a 4-connected grid structure graph. For example, Figure 3.2 is showing a grouping where every super node is consisted of the same number (four) children nodes.

In order to provide wide support for more general graphs in different topologies (not just grid structure graphs we have used for computer vision) and the flexibility for irregular grouping, we propose an adjacency list based construction of graphs where the graph at each level is consisted of **node**s and **adjacent node**s and every father node is linked with its children nodes crossing successive levels of graphs (used in hierarchical BP). Data cost information is stored in every node structure since every node is associated with a range of possible label values and their corresponding costs. Message information is stored in every adjacent node. The illustration of message updating in hierarchical BP and the data structure supporting this operation is shown in Figure 4.1 and Figure 4.2. As shown in Figure 4.2, in level *i*, adjacent node 2 (circle shape) in node 1's (square shape) list would store the message vector sent from node 1 to node 2 (the "right" direction message vector sent from node 1).

To better understand the reason why adjacency list is chosen to be used to describe every node (represented with **node** structures) the collection of its neighboring nodes/edges (represented with **adjacent node** structures) in our implementation, let's first look at the common operations that one can perform on a list, which are:

1. FIND(member): check if the member is in the list and if so give its location
2. INSERT(new): inset "new" member into a certain location in the list
3. DELETE(old): delete "old" member from a given location in the list

These operations make linked list a good candidate for describing a node's neighborhood since (1) it would be convenient for us to access a certain node's neighbor node(s) and it would be performed in a constant time per neighbor node; (2) It is straightforward to add or delete a node or edge in a certain graph; (3) Through the use of array of pointers, the upper bound on the list length can

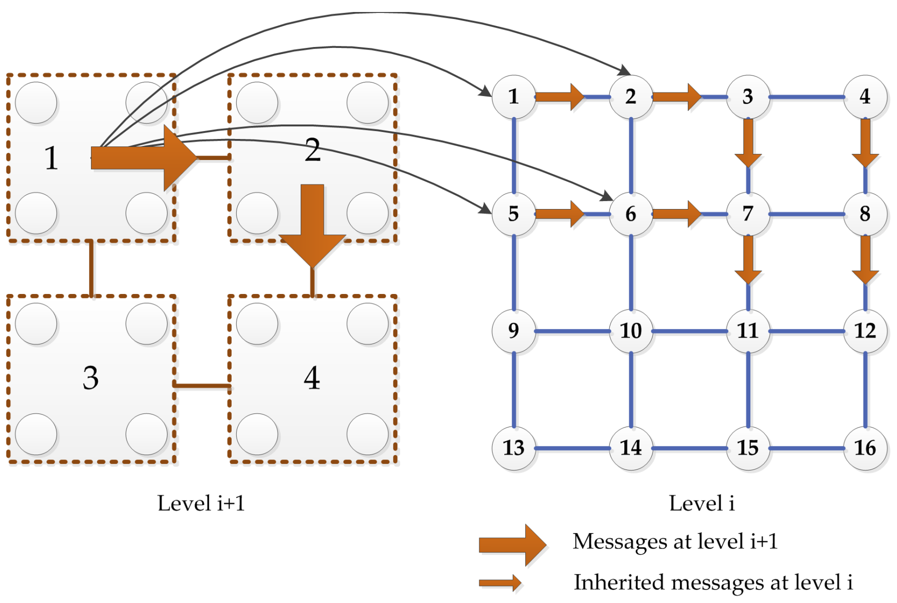

Figure 4.1: An example of initializing message vectors at level i from coarser level $i+1$. Every father node at level $i+1$ has 2×2 children nodes from level i. The correspondence between father node 1_{i+1} and children nodes $1_i, 2_i, 5_i, 6_i$ are shown with arrowed pointers. The arrows which represent messages are showing how the message vectors ("right" and "down") at level i are inherited from coarser level $i+1$. For example, message vectors from 1_i to 2_i, from 2_i to 3_i, from 5_i to 6_i and from 6_i to 7_i are initilized with the message vector values from 1_{i+1} to 2_{i+1} calculated at level $i+1$.

be removed, which means that every node can have different numbers of neighboring nodes (represented with adjacent nodes) and we can always add a new node to the original graph through inserting a new member in the corresponding linked list.

Recall from previous discussion in Section2.4 which explains that the motivation behind hierarchical BP is to facilitate long range flow of information across the MRF and thus to lead to faster convergence, a rigid grouping such as always grouping 4 nodes into one super node shown in Figure 3.2 may not be a good choice in many specific situations. For example, in an image restoration

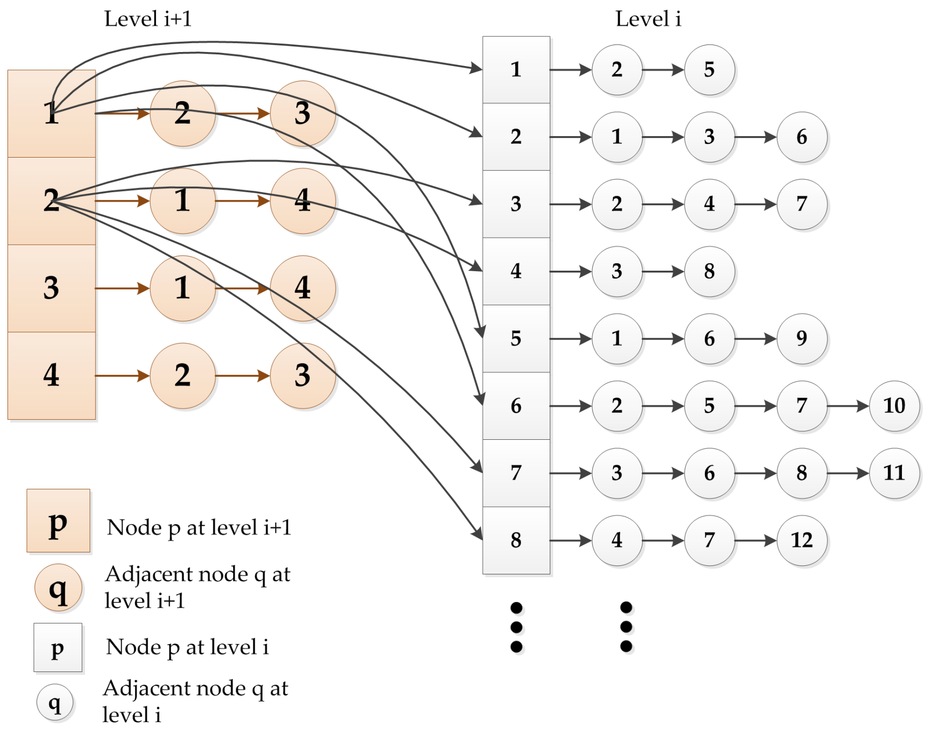

Figure 4.2: An illustration of a 16-node graph's linked list representation at level i (shown in left) and at level $i+1$ (shown in right). Every super-node at level $i + 1$ is a group of 2×2 nodes from level i.

task for the image shown in Figure 4.3, we probably would like to group one half of the pixels together into a large super-node and increase the convergence speed. Another example would be that when we have different emphasis on different parts of the graph, we might need to group nodes into super-nodes in different sizes. For an MRF built on adjacency lists which supports irregular grouping of nodes such as the example shown in Figure 4.4, when perform BP on this graph representation we are granted the flexibility of laying different emphasis on different portions of the graph model. This flexibility in grouping makes adjacency list structures a very good support for general hierarchical BP computation.

Another advantage we can obtain from using adjacency list is that it would

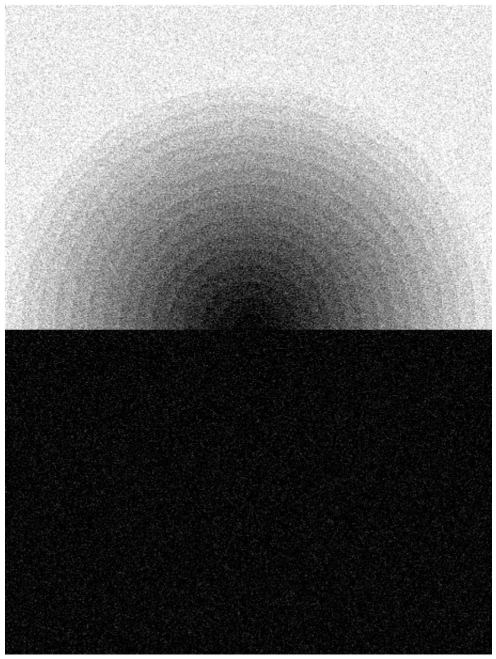

Figure 4.3: A noisy image where a large portion are expected to have similar label values.

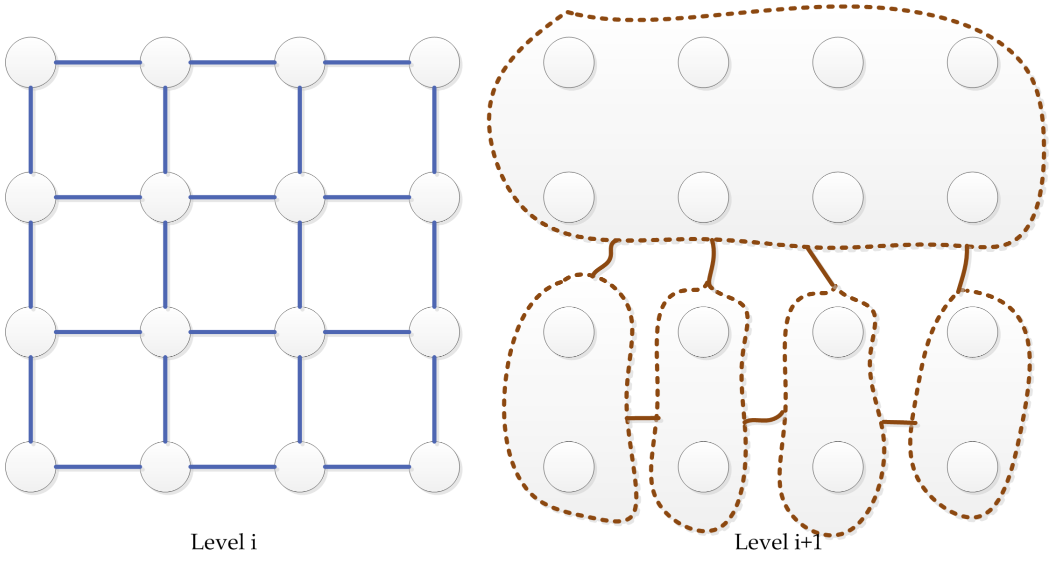

Figure 4.4: An example of aribitrary grouping of a grid structure graph from level i (16-node, shown in left) to level $i+1$ (5 super nodes, shown in right). The nodes $p_i \in \mathcal{P}_i$ at level i are shown as white circles. The nodes $p_{i+1} \in \mathcal{P}_{i+1}$ at level $i+1$ are shown as dashed circles. These super nodes at level $i+1$ have different numbers of children nodes.

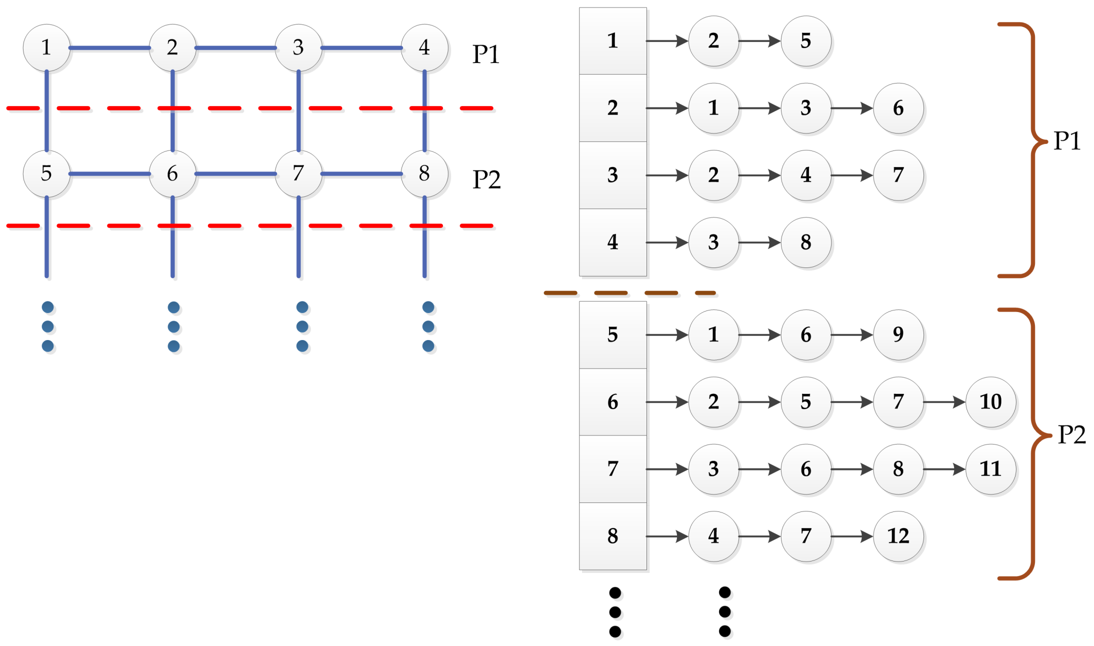

Figure 4.5: An example of computation work partition in a grid structure graph. Left image is showing two processors responsible for processing one row in the original graph respectively. Right image is showing a certain part of the adjacency list which is going to be written to by a specific processor. $p1$ and $p2$ denote processor 1 and processor 2 respectively.

be very easy for computation (workload) partition. This makes adjacency list data structures a good foundation for message updating/passing in belief propagation a parallel style. Figure 4.5 is showing how different processor (computation unit) is in charge of accessing and writing to different portions of the adjacency list structure.

4.2 Parallel Hierarchical Belief Propagation Implementation

4.2.1 Message Updating and Graph Partitioning

In this section, we are going to give a detailed description of our parallel hierarchical belief propagation implemented in C for shared memory architecture, where all communication between processors is accomplished through accessing shared memory space. First, we would present why asynchronous message updating would be a good choice for parallelized BP. Then we would provide a detailed description of the design of our parallel hierarchical belief propagation. Specifically, our algorithm is mainly consisted of two parts: (1) parallelization of pre-computation parts including node initialization, edge initialization, message initialization in every level, (2) parallelization of message passing (it is possible to use different message updating schedules).

As discussed in Chapter 3, many optimization methods, especially different message-update schedules have been proposed to accelerate the convergence speed, and some optimizations are also trying to reduce or control the memory or bandwidth requirement while speeding up the convergence to a fixed point. **Sequentially asynchronous message updating (BP-A)** scheme, where each updated message is immediately consumed for the calculation of next message, usually leads to a faster information flow and has been shown to converge to a unique fixed point under conditions similar to those that guarantee the convergence of BP using synchronous message updating [5]. Many results have shown that BP-A is much faster than the synchronous message updating (BP-S) (BP-S in combination with checkerboard style mention in Section 2.4). One intuitive way of understanding BP-A is that any new computation is triggered when new information arrives. Our research has shown that once applying BP-A to hierarchical BP, a stable results can be obtained at a very promising speed, which makes hierarchical belief propagation algorithm an ideal candidate for further adaption to realize real-time inference applications in embedded devices. Therefore, we use it one of the baseline designs for the performance comparison.

Any ordinary message update schedule can be used for a parallel implementation of BP, but we find that round robin BP-A (asynchronous message update schedule) best for adaption into a parallel algorithm considering both performance and implementation cost such as memory storage requirement. The graph level parallel abstraction relies on the partitioning, so we would like to (1) minimize the communication/dependency, (2) balance computation and storage. One example of performing our parallel message computation with round-robin asynchronous belief propagation using two processors is shown in Figure 4.6. Now we would explain the rationale behind our parallel scheme in three aspects.

First, we can distribute the computation work between different processors in a certain way thus the information flow would be not cut, which guarantees the ”sequential” property of asynchronous message schedule. The reason why the ”sequential” property is important is because while messages may be computed in any order (in synchronous message updating, for example, every message is computed once in every iteration), information is alway propagated sequentially. As we can see from Figure 4.6, updating of messages in the same direction in different rows (or columns) are independent to each other. Thus we can alway partition the graph in the same direction as the messages (information flow) to make sure that every processor is doing meaning work contributing to the final convergence to a fixed point.

Second, the synchronization problems in round-robin BP-A are easy to solve. As shown in Figure 4.6, all processors are simultaneously updating messages in the same direction in the same phase. For the first two phases (i.e, ″right″ and ″left″), we do not need to force any orders for synchronization purpose on these two processors since each processor only reads from memory space which is not written by any processors in these two phases (″up″ and ″down″ message values) and only writes to its own memory district. Similarly, there is no load-store memory consistency problem for the last two phases (i.e, ″up″ and ″down″).

Third, cache locality has been explored through exposing parallelism to round-robin BP-A where messages newly updated by one processor would be consumed by this processor immediately. This efficient memory accessing comes from the fact that we choose to use asynchronous message updating schedule where the calculation is performed in a sequential order. Also, since we are always updating messages sent from nodes in a specific order here, we might be able to make more use of a specific memory access pattern.

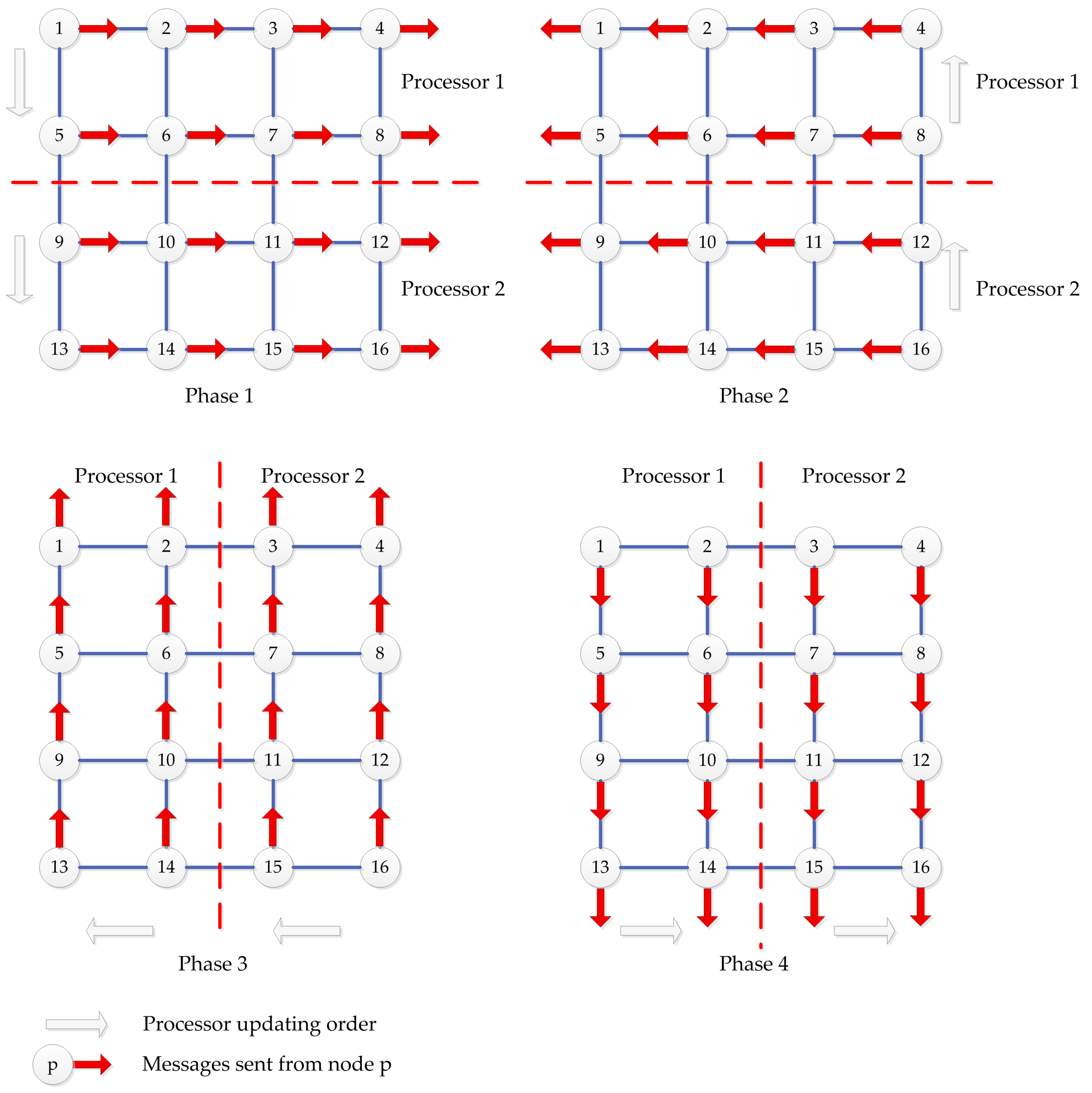

Figure 4.6: One iteration for two processors using round robin asynchronous message updating belief propagation (BP-A) in grid structure. Every iteration is consisted of four phases, where messages in ”right”, ”left”, ”up”, ”down” directions are updated respectively. For first two phases where messages in ”right” and ”left” direction get updated, we cut the grid structure graph in x direction so that the distributed work for two processors are independent. Similarly we cut the grid structure graph in *y* direction for ”up” and ”down” phases.

4.2.2 Parallel Algorithm

Algorithm 1 is showing how parallel hierarchical belief propagation is working in a big view. The computation can be divided into two phases. The first one would be node initialization which has been shown in details in Algorithm2 and the second phase is edge/message initialization (shown in Algorithm 3 and Algorithm 4) and iterative message passing calculation which is the the most computation intensive part (shown in Algorithm 5).

Message initialization in hierarchical BP

As mentioned in previous discussions, our implementation provides with the flexibility for passing messages (initializing messages) from coarser levels to finer levels in arbitrary graphs. In order to identify the correspondence between messages in different levels, **DirectionKey** is used to tag every message vector (stored in Adjacent Nodes) with a direction property. Direction Keys are attached to adjacent nodes in paraGraphInitAdj( ) and checked later in paraGraphInitMessg( ) for message initialization from coarser levels to finer levels. Figure 4.7 is showing how we are tagging messages in the same direction (or defined to be in the same direction) with a Direction Key. Figure 4.8 is showing how messages (stored in Adjacent Nodes) in different levels are defined using a common Direction Key set. In grid structure, for example, the value of DirectionKey would be 0(up) or 1(down) or 2(left) or 3(right).

Workload distribution

The balance of workload distribution is very straightforward in grid structures. Since there is a same number (four) of messages emanating from each node, we can always partition the graph into subgraphs with very balanced total number of nodes and thus guarantee that every processor has the same amount of message updates. However, the graph partitioning (workload distribution) may not be that straightforward in other cases, especially for irregular graph structures. Also, for a general graph, DirectionKey needs a specific definition from the algorithm designer. For irregular graphs, we should always try to partition the graph in a manner which avoids cut down the sequential information flow (carried by messages) just like what we have done in Figure 4.6 for a regular graph (grid structure). With a well defined DirectionKey, it would be easy to do message inherited from coarser levels to finer levels in any general graphs in our framework provided in Algorithm 3 and Algorithm 4.

Synchronization: memory locks and barriers

In order to guarantee the correctness and efficiency of our message passing computation conducted by several processors, we need to force some ordering constraint on memory operations in execution. Such memory fences are necessary because from a programmer's perspective, we have some exceptions or assumptions on the order to read and write operations of some shared variables (in a shared memory model, those variables are stored in shared memory). Changes in these ordering would either cause race condition (where locks and unlocks can be used to grant access to only one processor) or an unexpected result.

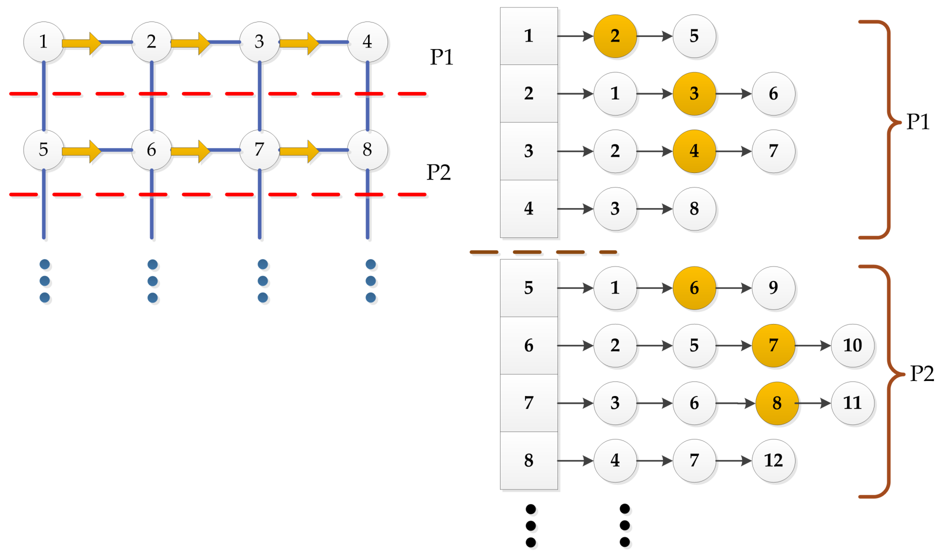

Figure 4.7: An illustration showing how adjacent nodes are tagged with DirectionKey. Adjacent nodes shown in yellow are tagged with a same Direction Key signifying that the directions of messages sent from their source nodes to them are the same (or we "define" them to be the same).

In our implementation, a pair of lock and unlock might be needed in paraGraphInitNode() depends on the graph partitioning. In line 17 out of Algorithm 2, we are writing to every super node in level k and the computation is partitioned on nodes in level k-1. For example, if we happen to have both node $q1$ and $q2$ grouped into one super node g_k, and node $q1$ is distributed to processor 1 while node $q2$ has been distributed to processor 2, we need to guarantee that the situation of both processor 1 and processor 2 are writing to g_k structure's memory space simultaneously would never happen.

As shown in algorithm 1, several barriers are used to synchronize between threads. The first barrier $Barrier : init_{node}$ is used to make sure that we would not proceed to the next level's node initialization until the initialization of all nodes in current level has finished. Similarly, $Barrier : init_{adj}$ is used to wait until the initialization of all messages are finished before start the message updating in

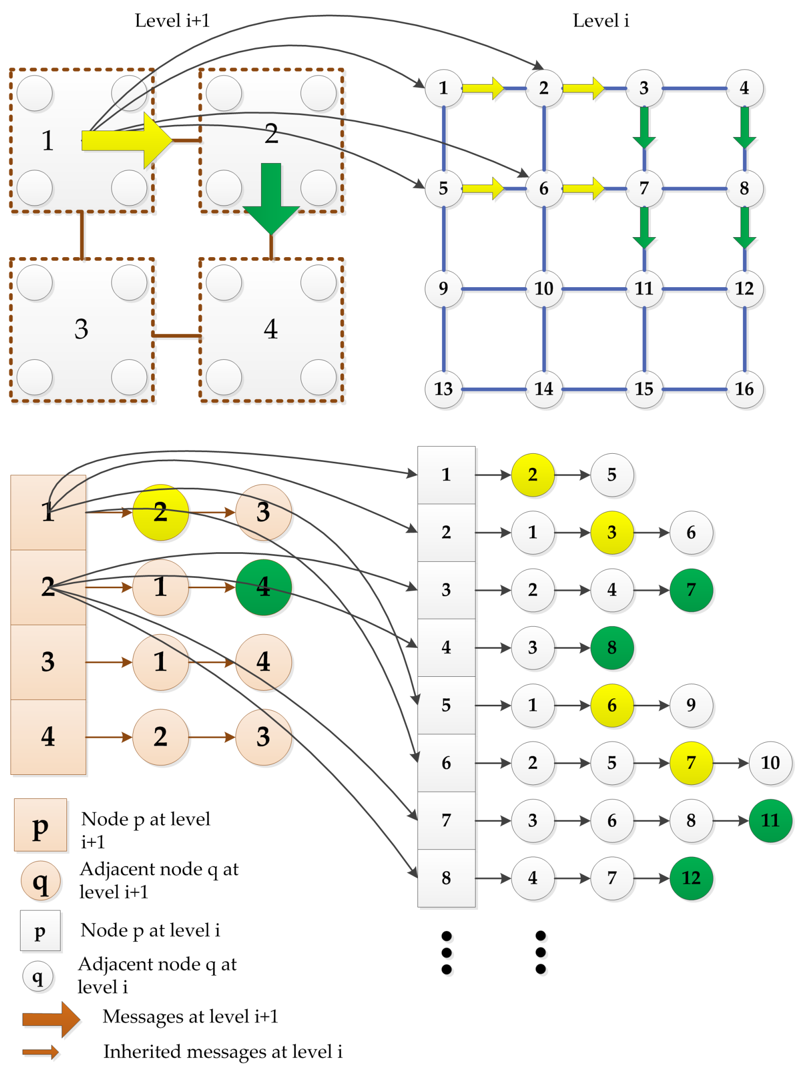

Figure 4.8: An illustration showing how adjacent nodes in different levels are tagged with Direction Key out of a direction key set. Adjacent nodes shown in yellow are tagged with a Direction Key signifying that the messages sent from their source nodes to them are in ”right” while green stands for ”down”.

paraBPA(). We do not need a barrier between the last iteration of *paraBPA()* and the successive *paraGraphInitAdj()* because the graph partitions we have done in *paraGraphInitAdj()* and *paraBPA()* in the same way so that processor *i* is responsible for the same subset of messages calculated (written) in *paraBPA()* and used (read from) in *paraGraphInitAdj()*.

Algorithm 1: A pseudocode for parallel hierarchical belief propagation: paraHBP()

```
1:  Every processor executes in parallel
2:  for level k=0 to $LEVELS$ (fine to coarse) do
3:      if it is level 0 then
4:          paraGraphInitNode(level=0);
5:                              ▷ Initialize nodes and load precomputed data costs
6:      else
7:          paraGraphInitNode(level=k);
8:                    ▷ Initialize nodes and calculate data costs from the finer level
9:          $Barrier(init_{node}, procs)$
10:     end if
11: end for
12:
13: for level k=$LEVELS$ to 0 (coarse to fine) do
14:     if it is level $LEVELS$ then
15:         paraGraphInitAdj( );                      ▷ Initialize adjacent nodes
16:     else
17:         paraGraphInitAdj( );                      ▷ Initialize adjacent nodes
18:         paraGraphInitMessg(level=k);
19:             ▷ Initialize messages using coarser $level = k - 1$ calculatoin results
20:     end if
21:     $Barrier(init_{adj}, procs)$
22:
23:     for iteration from 0 to $ITER_k$ do
24:         paraBPA();
25:     end for
26:
27: end for
28:
29: $Barrier(bp, procs)$
30: paraChooseLabel(level=0);
```

Algorithm 2: A pseudocode for parallel node initialization: function paraGraphInitNode(*level* = *k*)

```
1: Every processor executes in parallel
2: if it is level 0 then
3:     Graph node partition: define blocks of nodes in $level = k$ for processor $P_i$
4:     for every node in level k in my block do
5:         Initialize/allocate node structure
6:         load precomputed data costs
7:     end for
8:
9: else                                                    ▷ coarser levels
10:    Graph node partition: define blocks of nodes in $level = k$ for processor $P_i$
11:    Graph node partition: define blocks of nodes in $level = k-1$ for processor
       $P_i$
12:    for every node in $level = k$ in my block do
13:        Initialize/allocate node structure
14:    end for
15:    for every node $q$ in $level = k - 1$ in my block do
16:        Specify $q$'s father-node $g_k$ in $level = k$
17:        Update node $g_k$'s data cost, children node list
18:                ▷ A lock implementation might be needed for node $g_k$'s update
19:    end for
20: end if
```

Algorithm 3: A pseudocode for parallel message initialization: function paraGraphInitAdj( )

```
1: Every processor executes in parallel
2: Graph edge partition: define the block of node pairs for processor $P_i$
3: for every edge in my block do
4:     GraphCreatePair(DirectionKey)
5: ▷ For node pair $(p, q)$, create adjacent node q(p) in the neighbor list of node
   p(q) and link it with node q(p).
6: end for
```

Algorithm 4: A pseudo-code for parallel edge initialization: function paraGraphInitMessg( )

```
1: Every processor executes in parallel
2: Graph node partition: define blocks of nodes in $level = k$ for processor $P_i$
3: for every node $g$ in my block do
4:     for every child node $p$ (in level k-1) of node $g$ do
5:         compare and match $p$'s and $g$'s AdjNodes usingDirectionKey
6:         Initialize messages sent from $p$ using calculated message values sent
   from $g$ (for matched DirectionKey cases)
7:     end for
8: end for
```

Algorithm 5: A pseudo-code for parallel belief propagation using round robin asynchronous message updating in grid structure: function paraBPA( )

```
1: Every processor executes in parallel
2: Graph node partition in x direction: define the $block_x$ (rows of nodes) for processor $P_i$
3: Graph node partition in y direction: define the $block_y$ (columns of nodes) for processor $P_i$
4: for every row in my $block_x$ do
5:     for every node in current row (left to right) do                ▹ right
6:         Update message vector
7:     end for
8: end for
9: for every row in my $block_x$ do
10:    for every node in current row (right to left) do                ▹ left
11:        Update message vector
12:    end for
13: end for
14:
15: $BARRIER(bpDirection, procs)$
16:
17: for every row in my $block_y$ do
18:    for every node in current column (down to up) do                ▹ up
19:        Update message vector
20:    end for
21: end for
22: for every row in my $block_y$ do
23:    for every node in current column (up to down) do                ▹ down
24:        Update message vector
25:    end for
26: end for
```

4.3 Experiment Setting

We evaluate our hierarchical belief propagation implementation in sequential and shared-memory multi-core settings on a variety of graphical models. Specifically, we tested our code on 4*10 core Intel Xeon Processors (2.1GHz). Comparisons against other popular parallel implementations are provided.

For all experiments for stereo vision problems, we use truncated linear model for discontinuity costs, $V(f_p, f_q) = min(s|f_p - f_q|, d)$, where d denotes a truncation value for discontinuity cost. We use the following data cost function for a pixel $p = (x, y)$,

$$D_p(f_p) = \lambda \, min(|I_l(x, y) - I_r(x - f_p, y)|, t) \tag{4.3}$$

where t denotes a truncation value for data cost and λ denotes the relative weight of data costs. These three parameters d, t, λ are provided accordingly.

For all experiments for image restoration problems, discontinuity cost is given by $V(f_p, f_q) = min(|f_p - f_q|^2, d)$ and data cost fuction for one pixel variable(node) p is represented with $D_p(f_p) = \lambda min((I(p) - f_p)^2, t)$.

One thing to notice here is that we can always change our data cost, discontinuity cost models and parameter settings which would lead to a better results for a specific problem later and the same computation procedure described here would still fit nicely in applying belief propagation to solving a specific MRF. There are some other more complicated models for computing data cost terms ($D_p(f_p)$), like window-based normalized cross-correlation (NCC) would probably work better than this single pixel matching cost we are using here, especially for applications where the quality of stereo image pairs are not as good as Middlebury images due to intensity changes.

However, the point here is that programmers are granted the freedom of choosing their own definition of data cost terms (and/or smoothness terms) based on the kind of problem they are trying to solve. And we are focusing on the energy function optimization part once these parameters have been defined.

4.4 Results

4.4.1 Speed Analysis

$$speedup(n, p) = \frac{Time\ to\ solve\ using\ the\ best\ serial\ algorithm}{Time\ to\ solve\ using\ p\ processors} \tag{4.4}$$

In real speedup measure, the parallel execution time is compared against the execution time needed by the fastest serial algorithm for the application. Since for many applications ,we may not know the fastest one, or no one algorithm is the fastest for all examples, the run time of the sequential algorithm that is used in ″practice″ is chosen to be the comparison baseline. In Figure 4.9, hierarchical belief propagation using round robin asynchronous message updating is used as a sequential baseline for comparison. An almost linear speed-up is achieved here due to the natural graph partitioning we have achieved with round robin asynchronous message updating.

4.4.2 Computation Time Breakdown

We have also tested our parallel hierarchical BP code with image segmentation tasks. The results are represented in Figure 4.15. Specifically, we compared the computation time breakdown for different phases in parallel hierarchical

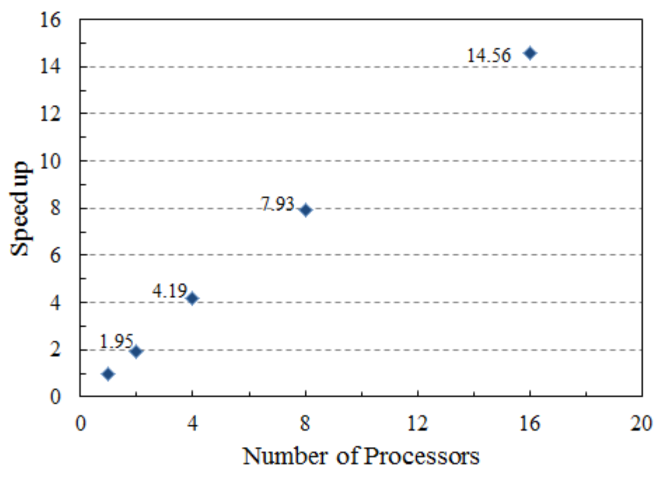

Figure 4.9: Speed-up for parallelized hierarchical belief propagation with round robin (sequential) asynchronous message updating (PHBP-A). Asynchronous message updating is performed in an up-down-left-right manner. HBP-A is used as a sequential baseline here.

BP scheme when using different numbers of iterations inside every level. Table 4.4.2 is showing the numbers and percentage execution time breakdown for one image using 8 processors and Figure 4.4.2 provides a visualization of this time breakdown. As we can see from these numbers and figures here, the most time consuming part is level 0 (task-level) iterative belief propagation, which is more than half of the total computation time for L3I3 (three levels and three iterations of message updating per level). Level 0 iterative BP is still around 40 percent of the total execution time even when only one iteration of message updating is performed (for L3I1). The reason why level 0 (task level) iterative BP is time consuming is because that at level 0 the number of nodes is exactly the same as the name of variables we are solving, which is probably a huge number. For example, the number of nodes (pixels) would be 307200 for a typical 640*480 image. For the optimal/ideal case, we might just need to visit every node once or twice (I1 or I2) and then obtain and finalize the marginal information for every variable in the task level. We should bear in mind that this visiting-every-variable at least once is inevitable because no matter how better our initial guesses have been made through computation in higher levels, we always have to come down to level 0 to compute for each node and finalize information for each node.

As shown in Figure 4.11, for the image segmentation task of one frame from a video input, we could achieve a pretty reasonable output with L3I1 whose total execution time is around the half of the total execution time for L3I3. The reason why we did not lower it down to one third (from L3I3 to L3I1) is because we always need to spend around the same amount of time for initial computation of costs and for each level's edge construction and message initialization. This also hints that once a hierarchical framework has been built for a relatively complex graphical model (a large number of nodes), it is not very expensive to perform more message updating inside each level.

Image Segmentation	L3I3		L3I2		L3I1	
PHBP-P8	Time (us)	Percent	Time (us)	Percent	Time (us)	Percent
Total node/cost Init	11210	5.1 %	10259	6.3 %	10470	8.9 %
Level 2 Graph Init Edge	1413	0.6 %	1079	0.7 %	1291	1.1 %
Level 2 BP	7734	3.5 %	5494	3.4 %	3389	2.9 %
Level 1 Graph Init Edge	4276	2.0 %	4148	2.6 %	3988	3.4 %
Level 1 Init Message	2610	1.2 %	2602	1.6 %	2734	2.3 %
Level 1 BP	29495	13.5 %	20889	12.9 %	10395	8.8 %
Level 0 Graph Init Edge	15342	7.0 %	16997	10.5 %	14517	12.3 %
Level 0 Init Message	11320	5.2%	11077	6.8 %	11072	9.4%
Level 0 BP	125960	57.8 %	85612	52.8 %	44887	38.0 %
Final Label Choose	7857	3.6 %	7805	4.8%	6911	5.8 %
Total calculation	217964		162250		118157	

Table 4.1: Computation time breakdown for processing one frame of video input using 8 processors. First two columns are the execution results from parallel hierarchical BP using asynchronous message updating (PHBP-A) with three levels and three iterations of message updating per level (L3I3). The next two columns are for PHBP-A L3I2. The last two columns are results from PHBP-A L3I1.

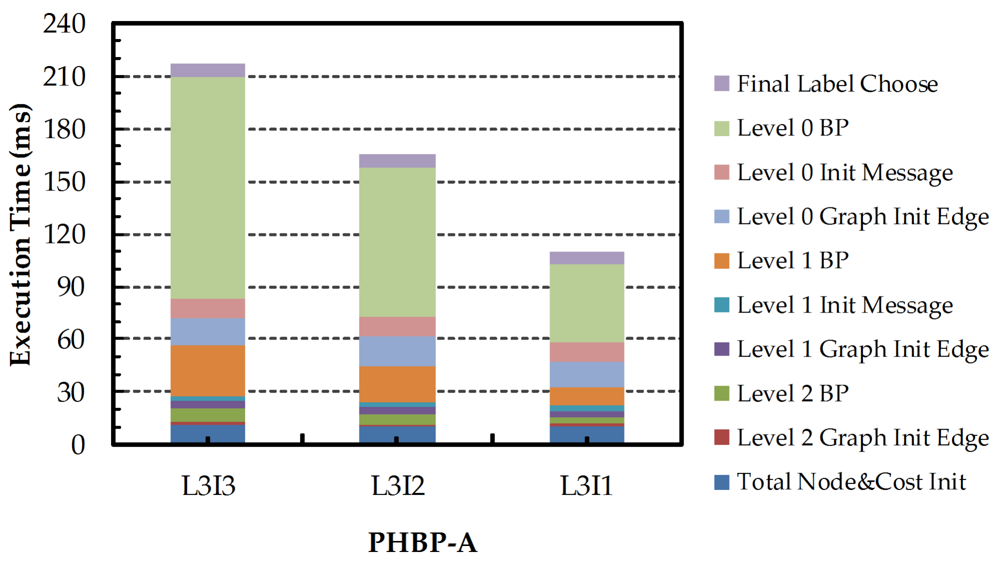

Figure 4.10: Computation time breakdown for processing one frame of video input using 8 processors. All results are obtained from PHBP-A.

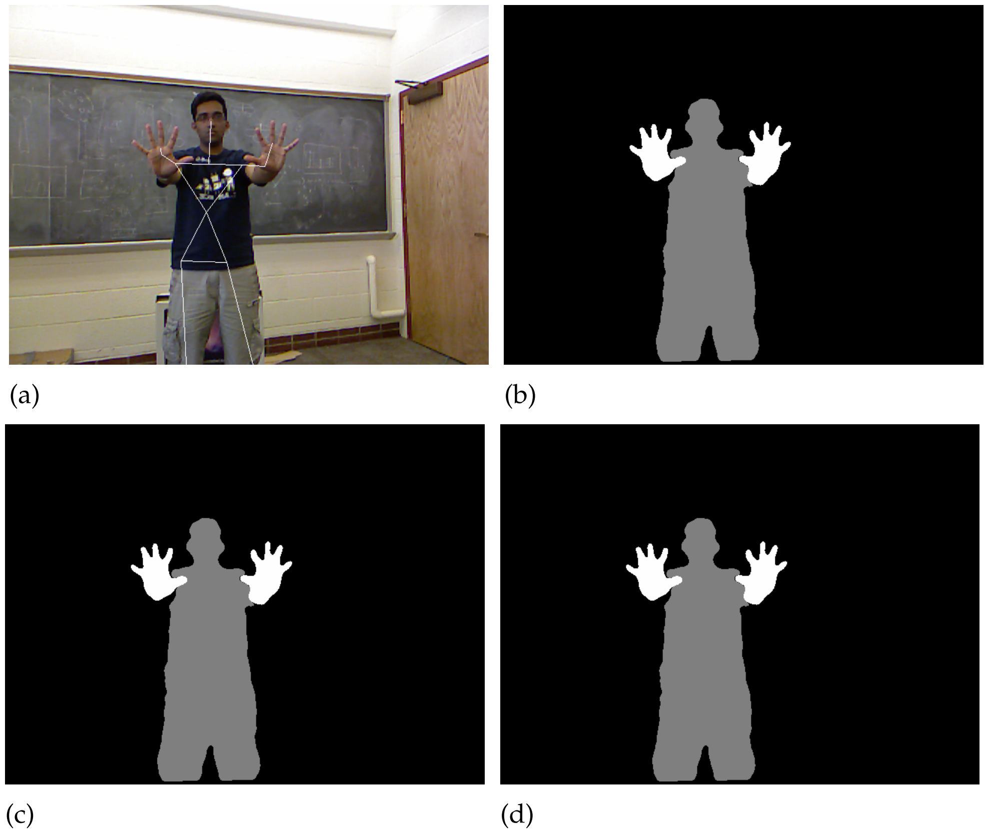

Figure 4.11: Image segmentation task with different number of iterations. First one is the input. Results from PHBP-A with L3I3, L3I2, L3I1 are shown from (b) to (d).

4.4.3 Comparison of Parallel Hierarchial BP with Other Parallel BP

Figure 4.12 shows how the energy function is minimized (optimized) in two versions of parallel belief propagation using round robin asynchronous message schedule. As we can see from this plot, hierarchical version (PHBP-A) can actually achieve a decent energy value using less than 1/5 of the time than the flat version (BP-A). And this is not just the case for Tsukuba images.

We have tested for four stereo matching problems provided in Middlebury

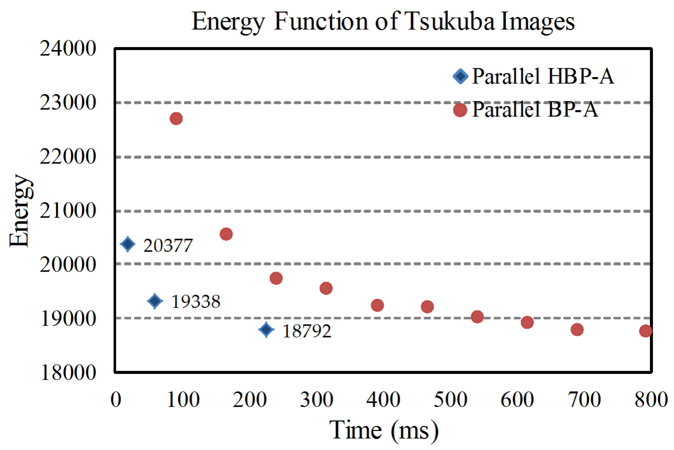

Figure 4.12: Energy of stereo solutions as a function of running time. Both PHBP-A and PBP-A are parallel versions working on 8 shared-memory processors.

stereo datasets. The solutions of using L3I1 (3 levels, 1 iteration per level) and L3I2 (3 levels, 2 iterations per level) are shown in Figure 4.14. Based on both the energy function values we have achieved and the observed output images, we could tell whether the chosen iteration time is enough for a certain application. For our PHBP-A scheme, even with L3I1, a pretty reasonable output is derived for all four Middlebury benchmarks. This result leads to an interesting comparison against Parallel Splash framework which is based on the theory that once we have visited and updated every message (or node) once a residual (updating priority) could be calculated and used to tell whether and when we need to update this message again in the future. However, if we only need one iteration, which is one visit to every message vector in level 0 (which is the ”variable node” or ”pixel” based real problem level) to reach convergence, there would be no need for computing residuals and schedule a revisit for the future at all. In order to get a quantitative sense about the convergence speed comparison of our hierarchy based implementation versus dynamic (residual based) asynchronous message updating based implementations, we choose to do an experiment on image restoration of a ”sun” image using both our implementation and Parallel Splash implementation [6].

For both parallel hierarchical BP (PHBP) and parallel splash BP (PSBP) implementations, we tested them on a same shared memory 8 core system. It took around 950 ms to solve an image segmentation task using PHBP and the a comparable results took about 45s using PSBP. The corresponding outputs of the image restoration tasks using both our parallel hierarchial belief propagation and parallel splash implementations are shown in Figure 4.13.

Since the execution time might be dependent on the efficiency of specific implementations, it may not be very intuitive to compare the execution time directly. In order to do a more direct and fair comparison with the Splash Parallel framework proposed in Residual Splash for Optimally Parallel Belief Propagation [6], we choose to compare the total number of updates needed for one image restoration task instead of the execution time. Table 4.4.3 illustrates that for a noisy image (608*456), PHBP-A needs to update 90k nodes while Parallel Splash needs to update 636k ”splashes”, where each splash is consisted of several nodes. Since the size of splash is changing dynamically in Parallel Splash BP, it is hard to estimate the actual size of average splash here. For a best case, the size of each splash would be one (depth of tree is one), which means that every splash is consisted of 5 nodes in a grid structure graph. We would see a 90k VS 3180k comparison between PHBP-A and Parallel Splash BP.

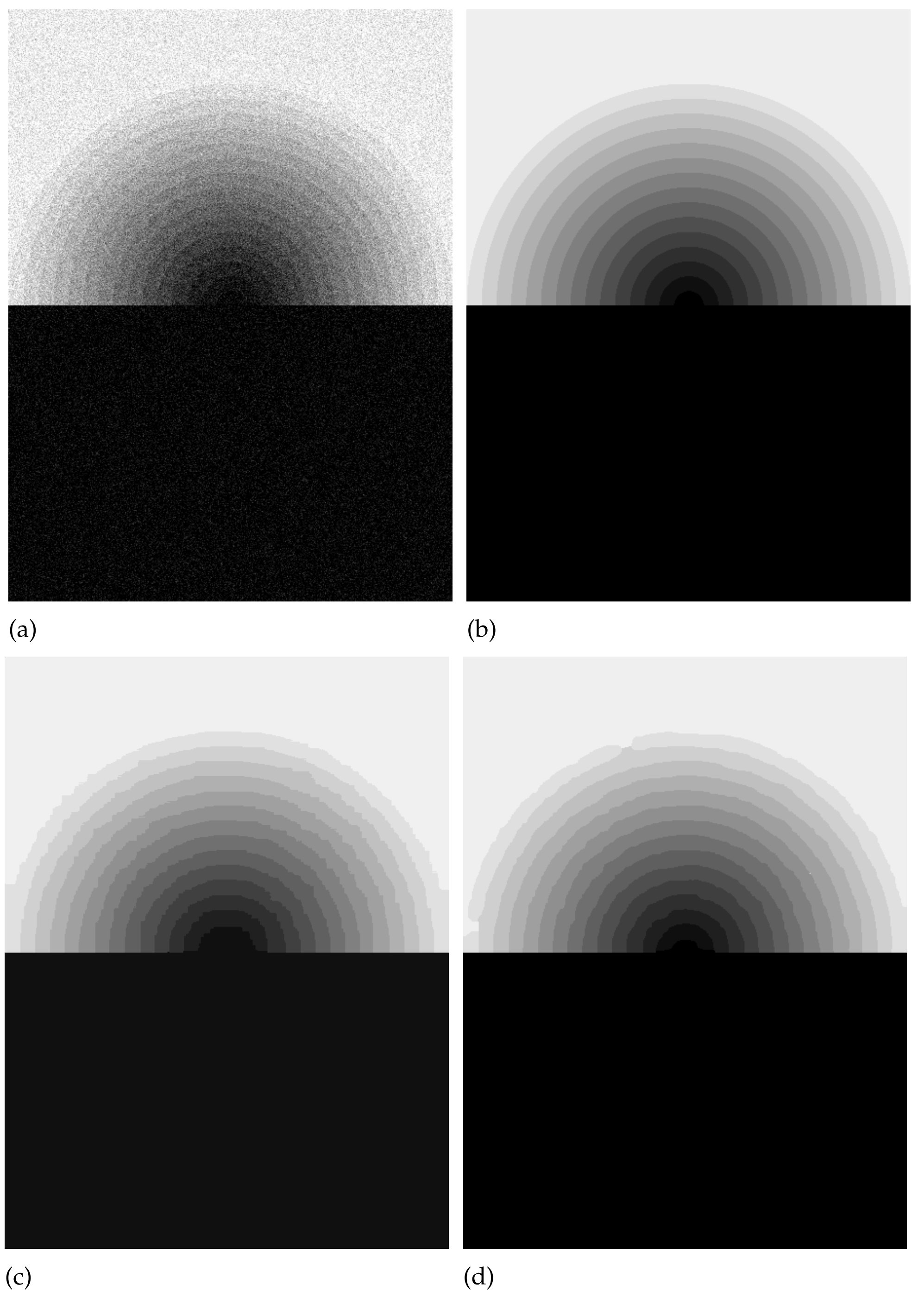

Figure 4.13: Image restoration task with "sun". (a) noisy picture. (b) clean picture. (c) Our outputs with PHBP-L3I2-P8. Average message update counts per processor is 363888. (d) Outputs with Parallel Splash when set residual to 0.02. Average message update counts per processor is 636780.

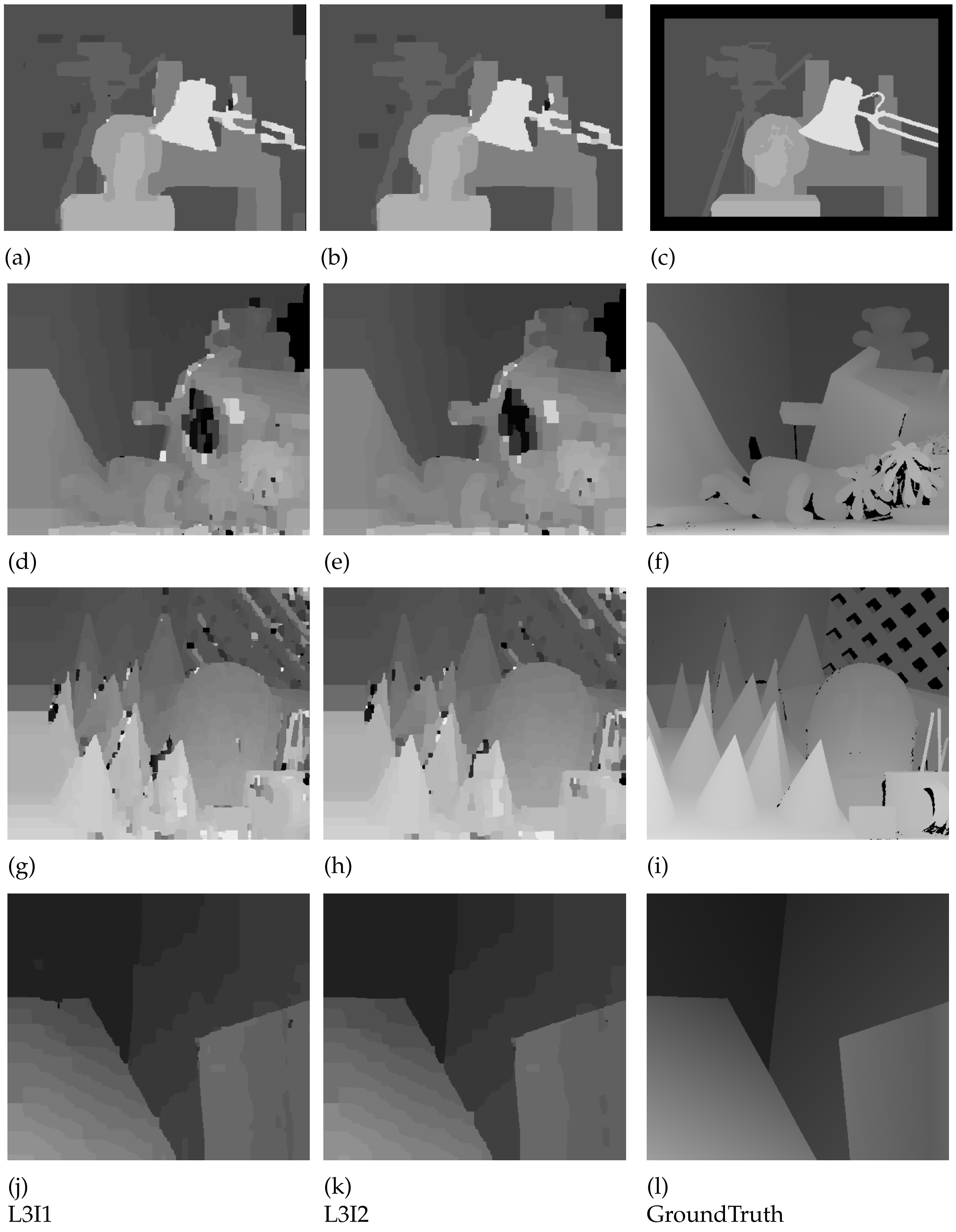

Figure 4.14: Outputs of four stereo pairs from Middlebury datasets. From top to bottom: Tsukuba, Teddy, Cones, Venus. Three columns are outputs of parallel HBP-L3I1-P8 (hierarchial belief propagation with 3 levels and1 iteration per level, 8 processors), outputs of parallel HBP-L3I2-P8, and corresponding ground truth respectively.

total number of updates	PHBP-A	Parallel Splash BP
image size (608*456)	90k	636k*splash size
image size (608*456), if splash size=1	90k	3180k

Table 4.2: Comparison of number of updates for image restoration task of ″sun″ image.

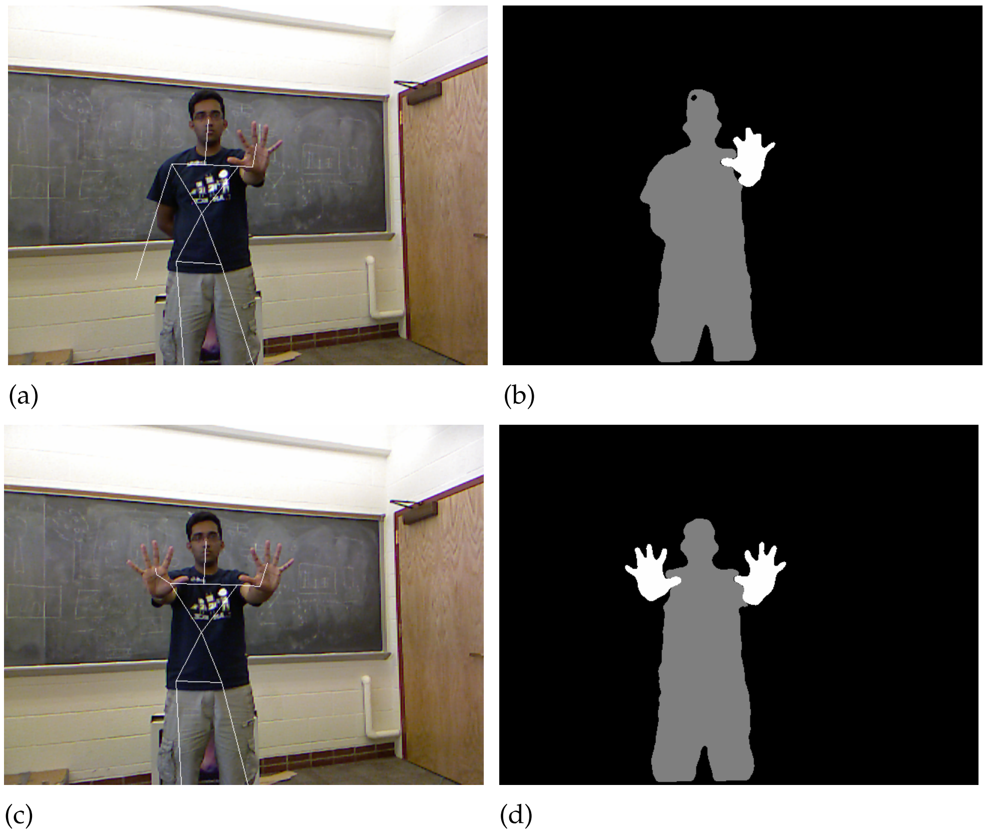

Figure 4.15: Image segmentation experiment for frame29 and frame35 images. Column one is original scene captured by Kinect. Column two is the output using PHBP-A (Parallelized HBP with Asynchronous Message Updating) output performed with 3 levels, 3 iterations/level (L3I3) on 8 processors. Number of labels is set to 3: 0 stands for hands(target), 1stands for body, 2 stands for background.

Stereo Problem	Tsukuba		Venus		Teddy		Cones	
	Nonocc	All	Nonocc	All	Nonocc	All	Nonocc	All
PHBP-P8-L3I2	1.99	3.80	1.33	2.42	22.0	30.0	17.8	24.8
PHBP-P8-L3I1	1.94	3.88	1.39	2.49	23.0	30.9	20.2	26.9

Table 4.3: Error in disparity estimation for Middlebury dataset.

4.4.4 Quality Analysis

Table 4.4.4 shows the percentage of bad pixels (errors) in disparity estimation. If the difference between the estimated disparity and the ground truth is more than one, it is marked as an error, or bad pixel. "Nonocc" stands for the ratio of the erroneous pixels in non-occluded regions, while "all" stands for this error ratio for all pixels. Figure 4.16 is showing the definition of non-occluded region for a "Venus" image and one example of the distribution of bad pixels in an disparity output of "Venus" image pairs.

As shown in Table 4.4.4, we achieved pretty promising results for Tsukuba and Venus, however our results for Teddy and Cones are not very ideal. One thing to note here is that the focus of our work is to demonstrate how our parallel hierarchical computational structure of BP is supporting the energy minimization problems in MRF. In order to show this, we just choose to use the simplest data cost function and discontinuity cost functions.

4.5 Discussion

Belief propagation offers parallelism opportunities on both graph level, where messages from different nodes can be computed in parallel, and message vector

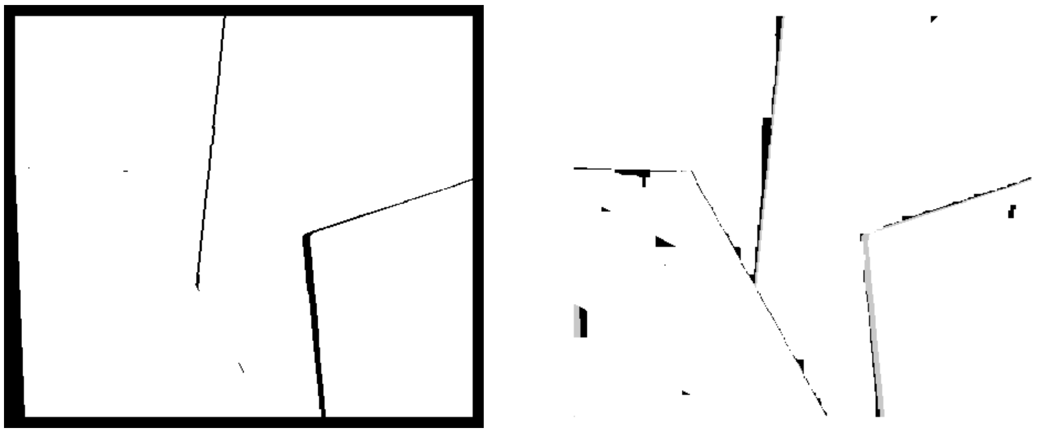

Figure 4.16: (a) is showing the Non-occluded regions (white) and boarder regions (black) for the Venus image. (b) is showing the distribution of bad pixels (with an absolute disparity error larger than 1) in the PHBP-P8-L3I1 output of Venus.

level, where the computation for each message vector over different label values can be expressed as matrix operations and then parallelized easily. In our discussion, we would focus on graph level parallelism since the message vector level trick is straightforward and we can always add it later.

Synchronous belief propagation is inherently parallel, where given all messages from last iteration, all messages in current iteration can be computed independently and simultaneously without any communication. As shown in[6], natural parallel synchronous BP is highly inefficient since in most cases, on every iteration, only a few newly computed messages increase awareness (information propagation) while the rest are wasted. This can also be intuitively understood through thinking about the fact that while messages may be computed in any order (in synchronous message updating, every message is computed once in every iteration), information is alway propagated sequentially. Many computations in synchronous message updating in earlier iterations would be obsolete and useless towards convergence.

In Parallel Splash framework proposed in [6], the local forward backward scheduling in a chain structure has been generalized to a forward backward scheduling on artificially constructed spanning trees rooted at nodes. For the problem level graph (eg. pixel-based graph for computer vision problems), splash algorithm has decreased the amount of message updating calculation at the expense of residual calculation and extra storage. However, when adopting a hierarchical calculation style using our data structure in a parallel framework, the simplest round-robin asynchronous message updating gives us an very ideal convergence speed and decent outputs with a straightforward graph partitioning, little extra calculation (no maintaining priority queue) and small memory requirement (no residual calculation/storage is needed). All these benefits provided by using round-robin asynchronous message schedule in a hierarchical BP framework prove it to be a better parallel algorithm than Parallel Splash in both performance and memory storage for some tasks. Or in other words, the advantage of using a dynamically aggressive (informed) message updating schedule as Splash is no longer alluring when the powerful hierarchical computation is adopted in a parallel framework.

Similarly in Yang's fast-converging BP [16], they updated only the messages of the non-converging pixels to get a computational complexity sub-linear in T. They showed that the GPU implementation of HBP can achieve near video rate (16 FPS) with low resolution images (320*240) ans small number of disparities (16). Several methods have been proposed to reduce the memory requirement of BP but mostly at the cost of increasing either the running time of message computation or the execution time for data cost computation.

Another high speed hierarchical work we have talked about in previous chapters is constant-space BP proposed by Yang [16]. In this work, unlike previous memory reduction methods focusing on the original spatial resolution, they hierarchically reduce the disparity range to be searched. This reduction in disparity range basically reduces the computational complexity through decrease the$N$ in $O(Nk^2T)$ (which would be $O(NkT)$ for truncated linear discontinuity cost model) . As we have shown from our time breakdown, BP computation in level 0 (original spatial resolution) graph is most computational intense since we have a large N in level 0. This hierarchical disparity range reduction is very effective when k is also very big. Thus ideally we would like to incorporate this hierarchical disparity range reduction in our PHBP when k is very big for some applications with high resolution (large k).

CHAPTER 5

CONCLUSION

In this work, we identified and addressed the challenges in the design and implementation of high speed efficient parallel belief propagation solving inference tasks in undirected graphical models in shared memory architecture. Especially, we propose a modification and generalization of the hierarchical BP algorithm presented by Felzenszwalb and Huttenlocher [1] which targets for four-connected grid structure graphs (early vision problems). Our implementation of hierarchical belief propagation is based on adjacency list representation of graphs, which (1) allows for arbitrary grouping of nodes in multi-scale computation through linking father node (in a higher level) with its children nodes (in a lower level), (2) would work for graphs in any general topologies (including non grid structure graphs) since the graph in each level is a direct description of nodes and its neighbor nodes and does not require graph to have a regular pattern. Second, we adopt sequential asynchronous message updating scheme (accelerated message updating) which has been shown to be good for further expansion to hardware implementations. And we have shown that the round robin asynchronous is easy to implement and lead to balanced workload distribution in parallel framework. Third, a fully parallel framework of hierarchical BP is developed. We achieved parallelization of both pre-computation parts and computational intense message passing portion.

Through comparisons with other parallel frameworks, out implementation outperforms residual based Parallel Splash in both performance and memory cost. Also, the flexibility and extension to general graphs and the support for irregular grouping in multi-scale computation makes our parallel hierarchical belief propagation very promising for a variety of applications in the future. One possible improvement in the future would be incorporating dynamical grouping of nodes in the multi-scale computation thus we could adaptively decide the ″scale″ of computation we would like to perform in each level. Another possible direction would be instead of using the same number of iterations and same scales (searching space) of messages for computation in different levels, we could do more computation/exploration in higher levels where the computational cost is lower while lay less computational cost in lower levels where the computation is very expensive.

APPENDIX A
CO-PROCESSOR DESIGN FOR BELIEF PROPAGATION

In my course project for Complex ASIC Design, I explored a co-processor design focusing on the message computation in Belief Propagation. Especially, I implemented a data-flow style *message engine* co-processor for the generation of each message in Belief Propagation. Further future research exploration would be using several message engine in parallel using a specifically designed memory (node) accessing pattern for achieving better performance.

A.1 Introduction

While transistor counts continue to double every generation, the power dissipation per transistor is only improving slightly. Given the limited chip-level power budget, the breakdown of voltage scaling is calling for a limitation of the percentage of transistor that can switch at full frequency. Therefore, while the transistor density continues to increase, the power efficiency has become the dominant limiting factor in processor design. Recent research efforts such as Quasi-Specific Cores, Conservation Cores (synthesized from application source code), Dynamically Specialized Execution [17] all demonstrate that customized hardware specification provides higher energy efficiency compared with general-purpose computer architectures.

With probabilistic graphical models serving as a powerful visual representation of statistical dependencies between variables, problems in different fields such as computer vision, artificial intelligence, speech and image process can all be reformulated as the computation of marginal probabilities on graphical models. And this computation process is often referred to as inference. In this project, we choose to focus on belief propagation (BP) algorithm which has been shown in Algorithm 6, which is one of the most powerful inference algorithms for solving problems built on probabilistic graphical models. Specifically, we would aim at BP performed on a grid structure graph, which means every node has four neighbors. Inside belief propagation, the most time-consuming computation part is called *message passing* (shown in Algorithm 7), which is required to be executed for each edge in the graph for many iterations. The computation complexity can be expressed as $O(Nk^2T)$, where N is the number of nodes to be resolved, k is the number of possible label values for each node, and T is the number of iterations. We would talk about this in details in Section A.3.

This project aims to carry out a study of both the benefits and trade-offs of employing application specific circuits in terms of performance, power and area as compared to a general-purpose PARCv2 processor. In order to do this, I started from the belief propagation algorithm written in C and then partitioned this application such that some portions would be compiled for use on the general purpose processors (GPPs) and some other portion (eg. message updating operation represented with function textbfmsg) will be implemented on coprocessor. The coprocessor is optimized to execute that particular segment of code in companion with processor. Specifically, since this message updating function (which is going to be built as a message engine in our design) is called in a way that memory addresses are accessed in a specific pattern, we could explore different possibilities in a architecture level to achieve higher performance and energy efficiency through instantiating different number of message engines and organise them in different ways.

A.2 Baseline Design

The baseline Deisgn would be the bypassed PARCv2 processor. After rewriting and adapting the belief propagation C code into a suitable version for further compilcation and evaluation, the complied assembly code in PARC ISA could be executed using the bypassed PARCv2 processor. This means that this PARC procesor is capable of executing our application benchmark (BP) and serving as a suitable baseline for comparison with the alternative design.

Specifically, as shown in Algorithm 6, function **msg** are called by four times consecutively. In order to do further comparison with our higher level architecture design with several message engines, we would generate two numbers in baseline design: (1) the cycle counts for one message updating function (2) the cycle counts for four message updating function as a whole.

A.3 Alternative Designs

After making analysis on the results of running the assembly version of BP algorithm on PARC processor (baseline design) with respect to cycle time, cycle counts, we identified the segment of code (execution) which should go to co-processor. Now we are going to show how we are building up the whole design incrementally and hierarchically.

A.3.1 Overview and breakdown of belief propagation algorithm

Upon analysis on belief propagation algorithm code, we can find that the computationally intensive portion (function *bp*) lies inside the outermost loop which would be called from iteration to iteration (let't denote the number of iterations as T). A pseudo-code of belief propagation function is shown in the Algorithm6 below. With a closer look into the computation of function **bp**, a natural thought would be dividing this computation into two parts: the first one is the generation of memory addresses for accessing four message arrays (u,d,l,r) and one cost arrays (cost); the second part is accessing those memory locations in a specific order and then perform a series of computation (function *msg*). The pseudo-code of message updating function msg is shown in Algorithm 7. In next section, we would look at this *msg* algorithm in details and analyze the data flow graph (DFG) of our loop kernel.

Algorithm 6: Belief propagation function $bp(u, d, l, r, cost)$

```
1: for every $node$ do
2:     //Address generation
3:     $node_u = node - W; //node\ sitting\ above$
4:     $node_d = node + W; //node\ sitting\ below$
5:     $node_r = node + 1; //node\ sitting\ in\ the\ right$
6:     $node_l = node - 1; //node\ sitting\ in\ the\ left$
7:
8:     //Message updating in four directions
9:     $msg(u[node_d], l[node_r], r[node_l], cost[node], u[node]);$
10:    $msg(d[node_u], l[node_r], r[node_l], cost[node], d[node]);$
11:    $msg(l[node_r], u[node_d], d[node_u], cost[node], l[node]);$
12:    $msg(r[node_l], u[node_d], d[node_u], cost[node], r[node]);$
13: end for
```

Algorithm 7: Message updating function $msg(s1, s2, s3, s4, dst, L, DISC_K)$

```
1: for $value = 0$ to $L(every\ label)$ do
2:     $dst[value] = s1[value] + s2[value] + s3[value] + s4[value];$
3:     if $dst[value] < min$ then
4:         $min = dst[value];$
5:     end if
6: end for
7: $min = min + DISC_K;$
8: for $value = 0$ to $L$ do
9:     if $min < dst[value]$ then
10:        $dst[value] = min;$
11:    end if
12: end for
```

A.3.2 Architecture design of Co-processor

The processor would communicate with co-processor via a val-rdy interface. There would be a command queue between processor and each co-processor. Inside co-processor, configuration registers would be set up to hold all the values the co-processor needs during its computation. For setting up each configuration register, one **mtvps** instruction would be called to move a specific value (corresponding to one argument in the C code of *msg*) from processor to co-processor before launching the computation in co-processor. Specifically, the **mtvps** instruction creates a message packet containing the co-pocessor register address and data which get enqueued onto the command queue. The **go** instruction sets the co-processor busy bit in processor and it is also sent through this command queue. This busy bit gets cleared when the co-processor execution is finished.

In our message engine, we need to use 7 configuration registers (serving as constants values during one complete execution of message engine) : $\{s1 \cdots s4, dst\}_{BaseAddress}$, constant $DISC_K$ and the length of array *length*. All above are about one message engine's architecture. After successfully realizing this message updating unit, I would move forward to pipeline this design.

A.3.3 processor/coprocessor interface

In order to answer the question of when and how would the co-processor know it should start computing(Leave IDLE state and enter CAL state), we are going to use a similar interface as last year's application specific loop accelerator (ALSA) project's interface between ALSA and control processor. Basically, we would have one instruction which sets up all co-processor registers by moving those values from general purpose register spaces. All our message engine argument including base addresses for arrays, data constants and array lengths would be passed from processor using this moving instruction.

After all arguments are moved to coprocessor registers, the processor would execute a go instruction to start this coprocessor (message engine for now). We might need a bit in processor signifying that the coprocessor is in use through the usage of **go** instruction. Upon completion of execution of the coprocessor, program flows would be redirected to the instruction after this go instruction.

Since we are going to have more than one message engines, we need to expand and add some extra controlling logic to this coprocessor configuration manager. Specifically, we would test and compare the trade-offs of having several message engines working simultaneously versus pipelining the message engine. For having several message engines, we only clear the busy bit when all message engines have completed calculation.

A.3.4 processor and coprocessor/memory interface

Here we are trying to answer questions like (1) How would storing/loading to/from data memory be performed? (2) Will there be a consistency problem if both of them having access to shared memory resource? Or in other words, should we allow the processor to work while the coprocessor is in execution (busy)?

For question (1), an architecture of arbitrating access to memory resources shared by processor and co-processor (two requesters) is developed. Upon receiving a request to access the shared memory resource from a first requester, we grant and lock access to memory to the first requester such that no other requesters may be granted the access. Fig A.2 is showing the interface between processor, co-processor and memory. A memory manager shown with dashed rectangle needs to be implemented in RTL.

For questions (2), in my BP application, once the memory accessing addresses have been calculated, the message engine would be launched for four times continuously, and then it proceed to the next group of address generation, and message updating for 4 directions again. Basically, it is repeating this process. For simplicity, I would choose to stall the processor when the co-processor is busy. If everything is working fine, I would proceed to a more aggressive design choice of having processor and co-processor working concurrently.

A.3.5 Data Flow driven Message Updating Engine

We chose to design a data driven architecture for our message updating engine. Our message updating engine is going to be built from a set of modules. As proposed in [18], self loop pipelining can be naturally achieved through replicating cyclic hardware structures (which are responsible for the control of loop iterations) and then get them autonomously executed in order, with synchronization being achieved by the data flow. Their scheme can also be applied to nested loops requiring less aggressive pipeline balancing efforts than usual software pipelining techniques. We would first use a similar technique to our inner loop: message updating engine. In order to do this, we are going to use a similar duplicating cyclic hardware structure scheme. The data flow graph is shown in Figure A.3.

As shown in Algorithm 7, the computation in one message engine is consisted of two loops: the first loop is calculating and storing the temporary results for every element in array *dst* while trying to find the minimum value of an array; The second loop is comparing every temporary result against a certain threshold value which was derived from minimum value found in loop1. For the second loop, one input (the temporary values of *dst*'s elements) comes from a large queue whose size is decided at design time, and another input (threshold value for comparison) comes from one value *min* calculated at the end of loop one. Thus we can construct the whole data flow diagram with two paths: one for the calculation of *min*, another for the temporary values of elements in *dst* array. The data flow graph is shown in Figure A.3.

For the first loop, first 5 address generators are used for loading $s1, s2, s3, s4$ from memory and writing calculation results *dst* back to memory. In order to find the minimum value in the temporary *dst* array, we use one module **LoopInit** to hold the current minimum value. Every time when there is a new value popped out from **ADD4** , current minimum value is used to compare against it. **LoopInit** is designed in the way that for the very first comparison it is initialized to be INF, and for the sucessive comparisons, it is always providing the last comparison round's output, which is held by a state variable inside this operator. A **Split** operator is controlled by a **Loopdone** operator which counts and redirects(controls) **Split** when it counts up to LoopCount. Through connecting in this way, the **Split** would always feed the current minimum back to the **Compare** operator when the first loop is not finished yet and sends down the final minimum value when the first loop is finished.

Now we have temporary *dst* elements computed from line 2 in Algorithm 7 held in the big queue. The size of this queue needs to be decided at design time and this limits the maximum length of the array that this message engine can deal with. We also have one threashold value coming from adding the minimum value together with constant $DISC_k$ as shown in line 7 in Algorithm 7. For the comparison operation(line 9 in Algorithm 7) in second loop, we need to generate the *min* threshold token for *LoopCount* times in order to consume the *LoopCount dst* elements provided from the big queue. Thus a **Generate** operator is used here to generate the *min* threshold token for *LoopCount* times.

We have several **AddrGen** operators in this message engine, which shows the idea presented in [18] which said that through duplicating the hardware structures responsible for the control of loop iterations, a natural loop pipelining can be achieved.

A.4 Evaluation: Initial Results

Table A.4 shows a performance comparison between co-processor and processor for a specific message updating function benchmark. The baseline design is evaluated with the BP algorithm implemented in C with msg function also declared in C. The co-processor design is is evaluated with the BP algorithm implemented in C with msg function declared with assembly instructions which

	length=2	
	Processor	Coprocessor
NumCycles	104	31
NumInstructions	86	27
bpmsgCycles		12
IPC	0.83	0.87

Table A.1: Comparison of cycle counts and number of instructions for baseline bypass processor design and alternative coprocessor design in different array lengths (which is L shown in Algorithm 7). This shows how coprocessor design is scaling for applications of different complexities.

sets up configuration registers in co-processor and then evokes the computation in co-processor.

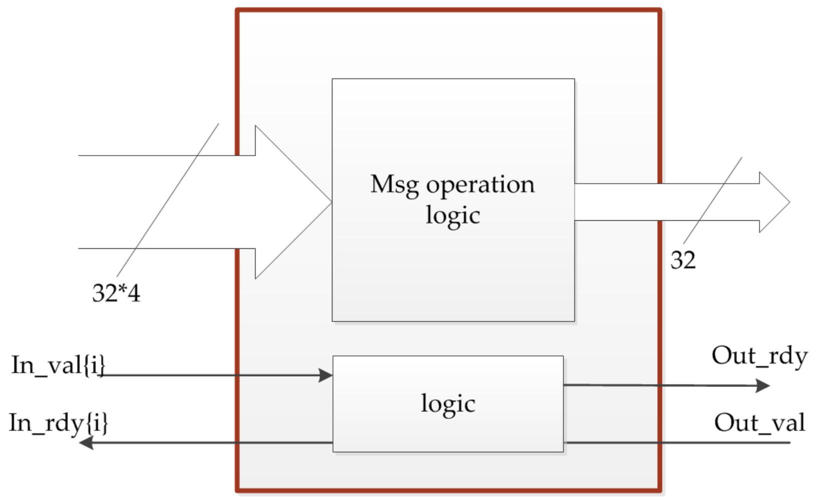

Figure A.1: A diagram representation of message engine with val-rdy interface.

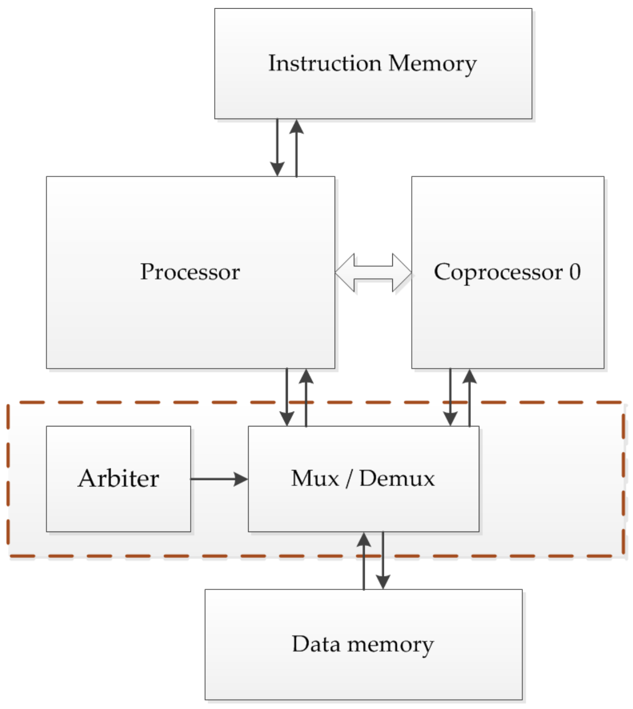

Figure A.2: A diagram representation of interface between processor, coprocessor and memory.

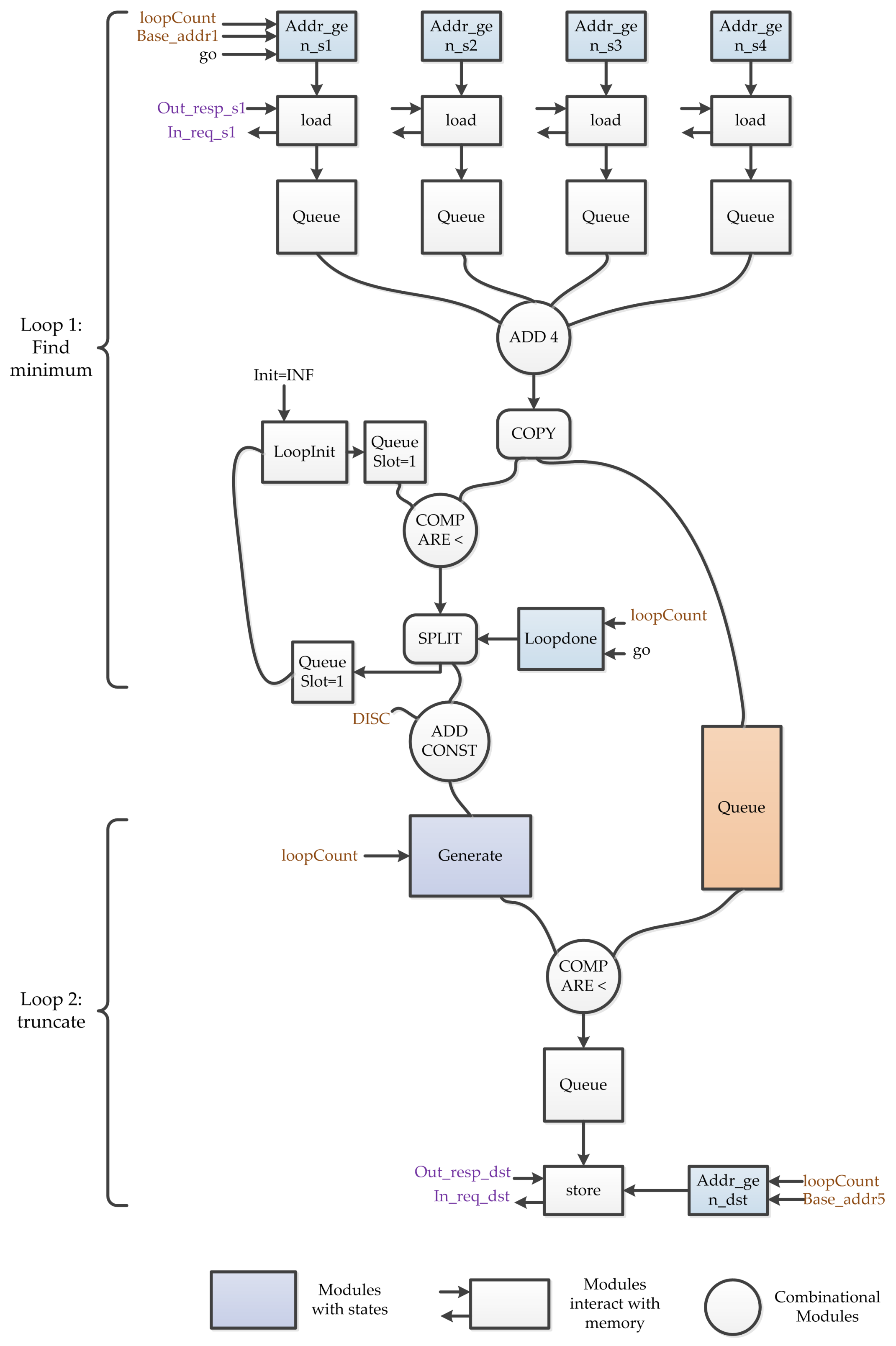

Figure A.3: Data flow graph for message engine. Circles represent logic or arithmetic operators. Rectangles represent operators with states. Rounded rectangles represent token(rdy,val) flow operators.

BIBLIOGRAPHY

[1] P. Felzenszwalb and D. Huttenlocher, "Efficient belief propagation for early vision," *Proceedings of the 2004 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2004. CVPR 2004.*, vol. 1, pp. 261–268, 2006.

[2] R. Szeliski, R. Zabih, D. Scharstein, O. Veksler, V. Kolmogorov, A. Agarwala, M. Tappen, and C. Rother, "A comparative study of energy minimization methods for Markov random fields with smoothness-based priors." *IEEE transactions on pattern analysis and machine intelligence*, vol. 30, no. 6, pp. 1068–80, Jun. 2008.

[3] F. Kschischang, "Factor graphs and the sum-product algorithm," *IEEE transactions on Information Theory*, vol. 47, no. 2, pp. 498–519, 2001.

[4] Y. Weiss and W. Freeman, "On the optimality of solutions of the max-product belief-propagation algorithm in arbitrary graphs," *IEEE Transactions on Information Theory*, vol. 47, no. 2, pp. 736–744, 2001.

[5] G. Elidan, I. McGraw, and D. Koller, "Residual belief propagation: Informed scheduling for asynchronous message passing," *Proceedings of the Twenty-second Conference on Uncertainty in AI (UAI)*, 2006.

[6] J. Gonzalez, Y. Low, and C. Guestrin, "Residual splash for optimally parallelizing belief propagation," *International Conference on Artificial Intelligence and Statistics*, vol. 5, 2009.

[7] E. Larsen and P. Mordohai, "Simplified belief propagation for multiple view reconstruction," *3D Data Processing*, 2006.

[8] T. Yu, R.-S. Lin, B. Super, and B. Tang, "Efficient Message Representations for Belief Propagation," *2007 IEEE 11th International Conference on Computer Vision*, pp. 1–8, 2007.

[9] M. F. Tappen and W. T. Freeman, "Comparison of graph cuts with belief propagation for stereo, using identical MRF parameters," *Proceedings Ninth IEEE International Conference on Computer Vision*, pp. 900–906 vol.2, 2003.

[10] J. Yedidia, W. Freeman, and Y. Weiss, "Understanding belief propagation and its generalizations," *International Conference on artificial intelligence*, 2003.

[11] Y. Boykov, O. Veksler, and R. Zabih, "Fast approximate energy minimization via graph cuts," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 23, no. 11, pp. 1222–1239, 2001.

[12] C.-C. Cheng, C.-T. Li, C.-K. Liang, Y.-C. Lai, and L.-G. Chen, "Architecture design of stereo matching using belief propagation," *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*, pp. 4109–4112, May 2010.

[13] K. Ogawara, "Approximate Belief Propagation by Hierarchical Averaging of Outgoing Messages," *2010 20th International Conference on Pattern Recognition*, no. i, pp. 1368–1372, Aug. 2010.

[14] M. Wainwright, T. Jaakkola, and A. Willsky, "Tree consistency and bounds on the performance of the max-product algorithm and its generalizations," *Statistics and Computing*, no. Bertsekas 1995, pp. 143–166, 2004.

[15] Q. Yang, L. Wang, and N. Ahuja, "A constant-space belief propagation algorithm for stereo matching," *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 1458–1465, Jun. 2010.

[16] Q. Yang, L. Wang, R. Yang, and S. Wang, "Real-time global stereo matching using hierarchical belief propagation," *3DPVT*, pp. 798–805, 2006.

[17] E. Magdaleno, J. P. Lüke, M. Rodríguez, and J. M. Rodríguez-Ramos, "Design of belief propagation based on FPGA for the multistereo CAFADIS camera." *Sensors (Basel, Switzerland)*, vol. 10, no. 10, pp. 9194–210, Jan. 2010.

[18] J. Cardoso, "Dynamic loop pipelining in data-driven architectures," *Proceedings of the 2nd conference on Computing*, pp. 106–115, 2005.