University of Alberta

MIMD Parallel Approaches to Object Labeling

by

Robert Michael Gregorish ©

A thesis
submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree
of Master of Science

Department of Computing Science

Edmonton, Alberta
Spring 1990

ISBN 0-315-60198-1

UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Robert Michael Gregorish
TITLE OF THESIS: MIMD Parallel Approaches to Object Labeling

DEGREE: Master of Science
YEAR THIS DEGREE GRANTED: 1990

(Signed) [illegible]

Permanent Address:
241 Verden Place
511110 Range Road 214
Sherwood Park, Alberta,
Canada, T8E 1G7

Date: April 25, 1990

Honesty, courage, loyalty and duty are not only their own reward,
but the only reward a self-respecting person needs.
Robert A. Heinlein

UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled **MIMD Parallel Approaches to Object Labeling** submitted by **Robert Michael Gregorish** in partial fulfillment of the requirements for the degree of Master of Science.

Xiaobo Li
(Supervisor)

[illegible]

[illegible]

Jack [illegible]

Date: April 25, 1990

Abstract

Efficient solution of the consistent (unambiguous) labeling problem has applications in fields including image processing, computer vision, graph theory, cryptography, and other areas which can be posed as constraint satisfaction problems. It has been shown that the labeling problem is inherently suited to implementation on parallel architectures.

Many authors have investigated parallel approaches, but few researchers have looked at MIMD solutions to this problem. In particular, none have looked at those aspects of MIMD which are unique to this class of architecture and which can be exploited for efficient solution of the consistent labeling problem.

This thesis investigates the behavior of three parallel algorithms for object labeling which attempt to take advantage of the strengths of MIMD architectures. The algorithms are implemented on the Myrias SPS-2 multiprocessor architecture using the Myrias model of parallel computation and the "pardo" programming construct.

An analysis of the behavior of these algorithms and recommendations for their improvement are presented. In addition, a method of ordering the search using characteristics of the problem to eliminate inconsistent labelings early in the search is proposed. This ordering technique has a significant effect on the behavior of labeling algorithms by increasing the effective use of available parallel resources.

Acknowledgements

Many people deserve to be recognized as contributors to this thesis. I would like to gratefully acknowledge the assistance of my examining committee, Xiaobo Li, Jonathan Schaeffer, Tony Marsland, and Jack Mowchenko. Jonathan Schaeffer also provided insight into tree search strategies and advice on the interpretation of the experimental results. I would like to thank my supervisor, Xiaobo Li, who provided valuable counsel and who provided inspiration and direction to this thesis.

I am appreciative of the contribution made to this thesis by Myrias Research Corporation by their generous donation of computer time on the Myrias SPS-2 parallel computer. Personal thanks are extended to Martin Walker of Myrias Research Corporation for his effort in securing the time for me on the Myrias machine and to Franco Carlacci of Myrias Research Corporation for his considerable contribution of time and effort in getting my programs to run on the SPS-2. In addition, I would like to acknowledge the contribution of the Alberta Research Council for providing additional SPS-2 computer time.

I wish also to acknowledge the very personal contribution of my wife, Lisa Strang, and of my parents, James and Edna Gregorish. They provided me with their patience, support, and encouragement throughout.

Contents

List of Figures

Chapter 1

Introduction

Characteristics of the object labeling problem may be exploited by multiple instruction, multiple data (MIMD) parallel architectures to provide performance improvement in many cases. This thesis looks at the subject of object labeling and investigates the application of a MIMD architecture to the solution of this problem.

> Because solving consistent labeling problems is so closely allied to solving general combinational reasoning problems, parallel algorithms and associated computer architectures for their fast solution are important to have in our toolbox. Knowledge of them will be of definite help in creating the parallel algorithms and associated parallel computer architectures for efficiently solving the most general predicate calculus types of problems. The efficient solution of this kind of problem will be the hallmark of the next generation of smart computers. *Ullmann, Haralick, and Shapiro* [29].

1.1 The Labeling Problem

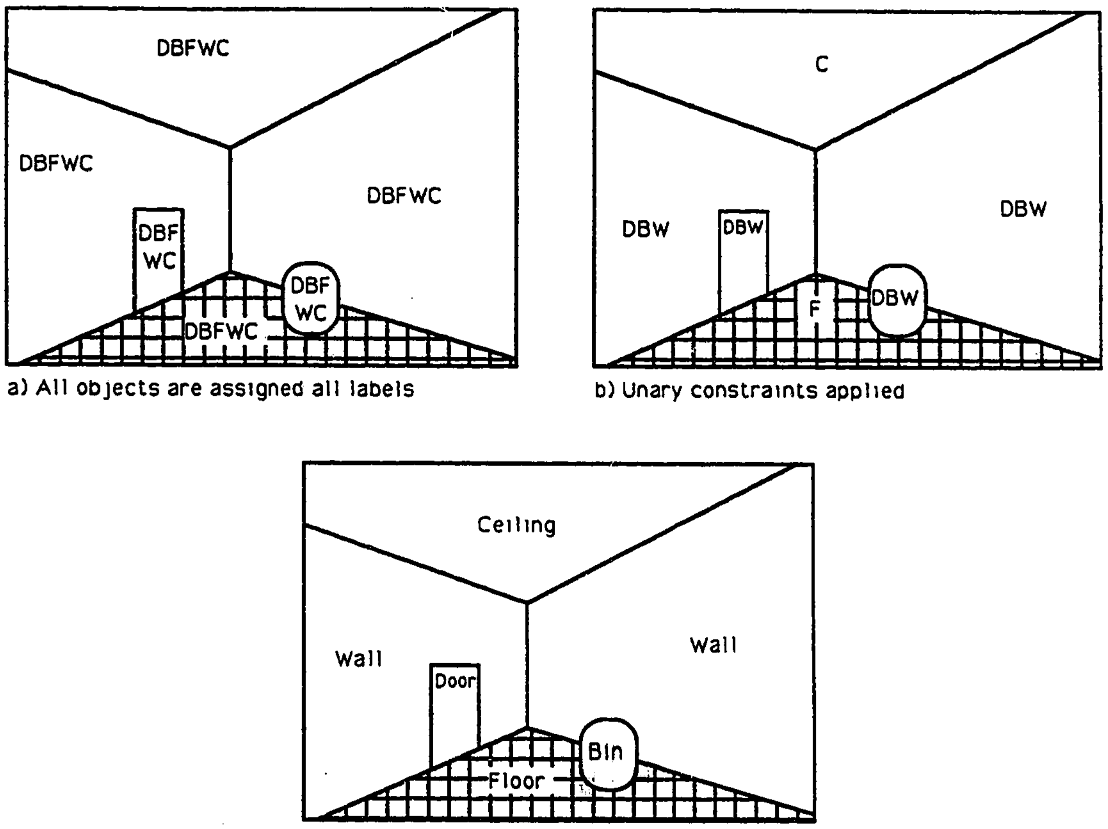
1.1.1 Basic Definitions and Terms

The general labeling problem consists of assigning *labels* to a set of *objects*, such that a set of *constraints* are not violated. A *labeling* is an assignment of a set of labels to each object. A labeling is said to be *consistent* if for each pair of objects and for every label of one of the objects, there exists a label for the other object which does not violate any constraints. A labeling is called *unambiguous* if it is consistent and assigns only a single label to each object.

In many applications, the problem of labeling is to find all unambiguous labelings for a given set of objects, labels, and constraints. In some applications, the problem may be stated as finding any single unambiguous labeling.

This thesis focuses primarily on the problem of labeling line drawings (planar projections of polyhedra), although object labeling has applications in many fields. Line drawings occur quite frequently in image processing as the result of edge detection or segmentation of real images. This would have applications in the fields of scene understanding and computer vision.

The constraints which have to be satisfied in a labeling problem occur because the objects in a given problem are related in some manner. These are often geometric or topological relations which limit the set of consistent labelings. Objects may also possess *properties* which are thought of as unary relations. These relationships are used to reduce or eliminate ambiguity in the problem.



a) All objects are assigned all labels

b) Unary constraints applied

c) Relational constraints applied

Figure 1: An Example of a Labeling Problem

1.1.1.1 An Example

At this point, an example may serve to illustrate these concepts. Consider the "stylized segmented office scene" [1] recreated in Figure 1. The problem is to assign the labels Door(D), Wall(W), Ceiling(C), Floor(F), and Bin(B) to the objects in the scene. Object properties constrain labels with unary relations and binary relations use pairs of objects to constrain labels. Some possible constraints for this scene can be informally stated.

Unary Constraints:

1. The Ceiling is the single highest region in the image.
2. The Floor must be checkered.

Binary Constraints:

1. A Door is adjacent to the Floor and a Wall.
2. A Wall is adjacent to the Floor and Ceiling.
3. A Bin is smaller than a Door.
4. A Bin is adjacent to a Floor.

The set of constraints depends on the problem being considered as well as the type and quality of information available and its value in constraining the labeling. In this artificial example, the application of these constraints results in the generation of a unique unambiguous labeling as indicated in Figure 1(c).

1.1.2 Notation

Unfortunately, there is no generally accepted standard notation regarding the labeling problem. Many researchers have developed their own notation and often where different researchers use the same notation to mean different things, this leads to confusion. Leung [18] provides a thorough summary of the various notation schemes used by the leading researchers in this field.

The main definitions used in this thesis for the labeling problem are taken from Rosenfeld, *et al.* [24], which is a major publication in the field of relaxation labeling, often cited by later authors. They provide a rigorous definition of the terms *unambiguous* and *consistent.* Their definition of the discrete model for labeling is presented in Section 1.1.2.1.

Haralick and Shapiro [8] introduce the concept of a *compatibility model.* This model is used in the development of the algorithms presented later in this thesis, so the definition of this model is presented in Section 1.1.2.2.

For a discussion of the relationship between these two models and models proposed by other researchers refer to Leung [18].

1.1.2.1 The Discrete Model

(Rosenfeld *et al.* [24])

Let $A = \{a_1, \ldots, a_n\}$ be the set of n objects to be labeled and let $\Lambda = \{\lambda_1, \ldots, \lambda_m\}$ be the set of m possible labels. For a specific object a_i, not every label in Λ may be suitable. Let $\Lambda_i \subseteq \Lambda$ be the set of labels which are compatible with object $a_i, 1 \leq i \leq n$. The set Λ_i depends on the properties of the object a_i.

For each pair of objects (a_i, a_j), $i \neq j$, some pairs of labels may be compatible and some others may not. Let $\Lambda_{i,j} \subseteq \Lambda_i \times \Lambda_j$ be the set of compatible pairs of labels. $(\lambda, \lambda') \in \Lambda_{i,j}$ means that it is possible for object a_i to have label λ and object a_j to have label λ'.

The set $\Lambda_{i,j}$ depends on the relationship between objects a_i and a_j in the scene. If a_i and a_j are not related to each other, then there are no restrictions on the pairs of labels which are compatible with these two objects, so we get $\Lambda_{i,j} = \Lambda_i \times \Lambda_j$. In addition, the special cases $\Lambda_{i,i} = \{(\lambda, \lambda) | \lambda \in \Lambda_i\}$ for all i and $\Lambda_{i,j} = \Lambda_{j,i}$ are defined.

A *labeling* $\mathcal{L} = (L_1, \ldots, L_n)$ of A is an assignment of a set of labels $L_i \subseteq \Lambda$ to each $a_i \in A$. $\mathcal{L}$ is *consistent* if, for all i, j,

$$(\{\lambda\} \times L_j) \cap \Lambda_{i,j} \neq \phi, \text{ for all } \lambda \in L_i$$

For $i \neq j$ this means that for every pair of objects (a_i, a_j) and each label λ in L_i there exists a label λ' in L_j that is compatible with λ.

The null labeling $\mathcal{L}_0 \equiv (\phi, \ldots, \phi)$ is trivially consistent and shows that there always exists at least one consistent labeling for a set of objects. However, if a non-null labeling $\mathcal{L}$ is consistent, then *every* L_i of $\mathcal{L}$ must have at least one label. For instance, if there exists some $L_i \neq \phi$ and $L_j = \phi$ for some i, j, then the definition of consistent is violated. There is also a special labeling, $\mathcal{L}^{(\infty)}$, called the *greatest consistent labeling* which is

1. consistent, and

2. if $\mathcal{L}$ is consistent then $\mathcal{L} \subseteq \mathcal{L}^{(\infty)}$.

A labeling $\mathcal{L}$ is *unambiguous* if it is consistent and assigns a single label to each object. Note that it is possible for there to exist a non-null consistent labeling, but for there to be no labeling which assigns a single label to each object. For example, consider the following [24]:

$$A = \{a_1, a_2, a_3\}$$

$$\Lambda = \Lambda_1 = \Lambda_2 = \Lambda_3 = \{\lambda, \mu\}$$

$$\Lambda_{1,2} = \Lambda_{2,3} = \{(\lambda, \lambda), (\mu, \mu)\}$$

$$\Lambda_{1,3} = \{(\lambda, \mu), (\mu, \lambda)\}.$$

In this example, the labeling $\mathcal{L}^{(\infty)} = (\Lambda, \Lambda, \Lambda)$ is consistent, but there does not exist a consistent labeling which assigns a single label to each object.

1.1.2.2 The Compatibility Model

(Haralick and Shapiro [8])

The *compatibility model* introduced by Haralick and Shapiro [8] is sometimes referred to as a *world model.* The basic concepts of the model are as follows:

1. A set of units $U = \{u_1, \ldots, u_M\}$, analogous to the set A in Section 1.1.2.1.

2. A set of labels, $\Lambda = \{\lambda_1, \ldots, \lambda_K\}$ as in Section 1.1.2.1. This is a slight deviation from the notation in [8], which uses L as the set of labels.

3. A set $T \subseteq U^N$ which is the set of all N-tuples of units which mutually constrain one another. Units may constrain each other N at a time, where N is any number $\leq M$. T is called the *unit constraint relation.*

4. A set $R \subseteq (U \times L)^N$ which is the set of all $2N$-tuples $(u_1, \lambda_1, \ldots, u_N, \lambda_N)$ where $(\lambda_1, \ldots, \lambda_N)$ is a legal labeling of units $(u_1, \ldots, u_N)$. R is called the *unit-label constraint relation.*

As a simple example [8], consider the following:

$$
\begin{aligned}
U &= \{1,2,3,4,5\} \\
\Lambda &= \{a,b\} \\
T &= \{(1,2,3),(1,2,4),(1,2,5),(2,3,4),(2,3,5),(3,4,5)\} \\
R &= \{(1,a,2,a,3,a), \\
&\quad (1,a,2,a,4,a), \\
&\quad (1,a,2,a,5,a),
\end{aligned}
$$

$(1, a, 2, b, 3, a)$,

$(1, a, 2, b, 4, b)$,

$(1, b, 2, b, 5, b)$,

$(2, a, 3, a, 4, a)$,

$(2, a, 3, a, 5, a)$,

$(2, b, 3, a, 4, b)$,

$(3, a, 4, a, 5, a)\}$

Given this compatibility model, (U, Λ, T, R), only one unambiguous labeling is possible; (a, a, a, a, a) for objects $(1, 2, 3, 4, 5)$.

In a compatibility model, units may constrain each other N at a time, where N is any number $\leq M$. There may be unary, binary, or N-ary constraints within a model. This means that the set T may have tuples of various length, and for each tuple in T, there are corresponding tuples of twice the length in set R indicating how the units constrain each other.

Haralick and Shapiro unfortunately use the term *consistent* labeling to refer to what Rosenfeld *et al.* [24] call an *unambiguous* labeling. This thesis will also adopt this convention and refer to the *consistent labeling problem* as the determination of *unambiguous* labelings.

1.1.3 Applications of the Labeling Problem

The labeling problem is a generalization of several problems from varying fields of specialization. Haralick and Shapiro [8] cite several examples: the subgraph isomorphism problem [6], the graph homomorphism problem, the automata homomorphism problem, the graph coloring problem, the relational homomorphism problem [7], the packing problem, the scene labeling problem, the shape matching problem, the Latin square puzzle, constraint satisfaction problems [4], and theorem proving. Rosenfeld *et al.* [24] also mention the solution of cryptanalytic problems, region merging in a scene analysis system, and a variety of scene processing problems including curve detection, cluster detection, noise cleaning, and template matching.

1.2 Motivation for this Research

The solution of the labeling problem has potential benefit to many areas as indicated in Section 1.1.3. While the primary focus of this thesis is in the domain of image processing, there is much interest in applications wherever the problem can be posed as a constraint satisfaction problem.

Research efforts are motivated along several paths. Some researchers have investigated the theoretical analysis of the labeling problem, with the hope that a better understanding of the foundation of labeling will provide insight into better solutions [9, 24, 12].

Others are developing various algorithms and approaches with the primary focus being practical results. Research along this line considers such things as strategies to reduce the search space to improve performance [8, 32, 29, 11, 27, 20, 21, 28].

Other approaches look at exploiting the parallelism inherent in the labeling problem. Most of the research in this area has involved looking at algorithms for

SIMD computer architectures, with the focus on reducing either the communication complexity, the memory requirement, or the computation complexity of the solution [17, 18, 5, 14, 19].

There has not been much research done in the field of MIMD approaches to the labeling problem[26, 29]. A MIMD computer architecture encourages a higher level approach to the design of a parallel algorithm than does the SIMD class of parallel computers. The SIMD model requires that a problem be broken up into homogeneous pieces that can be handled identically on several processing elements. The MIMD approach allows the problem to be broken up, perhaps unevenly, over the set of processing elements to take advantage of non-uniform or non-symmetric characteristics of the problem.

Since this class of parallel computer architecture is becoming more commonplace, it is important to investigate how the features unique to MIMD parallel computers can be exploited to increase the performance of solutions to the labeling problem. This thesis focuses on finding unambiguous consistent labelings of line drawings.

1.3 Outline of this Thesis

Preliminary investigation into the field of object labeling involved a review of past and present research efforts. Chapter 2 provides an overview of research in the field of labeling. The first MIMD approach, "Recursive Descent with Deepening", is presented in Chapter 3. Two alternative approaches, "Split-Label-Merge" and "Eureka Jump" are presented in Chapter 4. Finally, the conclusions of this thesis and suggestions for future work are presented in Chapter 5.

The Appendices contain some background information. The sample labeling problems consisting of 9, 20, 33, and 54 objects are presented in Appendix A. Appendix B contains the input format for the test line drawings. A discussion of parallel architectures is presented in Appendix C. An overview of the Myrias SPS-2 parallel computer and the PAMS operating system is presented in Appendix D.

Chapter 2

Research in Labeling

2.1 Overview

An important research area in image processing and image understanding is how to incorporate contextual knowledge into the interpretation of objects. Labeling techniques have primarily developed from Waltz's early work in unambiguous labeling of line drawings [32]. Waltz's technique was discrete in nature, in that it allowed only unambiguous interpretations of line segments.

Relaxation labeling was formalized and made popular by Rosenfeld *et al.* [24] in what is considered a classic paper in relaxation labeling. Their paper presents a formal definition of labeling and develops discrete, fuzzy, and probabilistic models for the solution of this problem. They also show that Waltz's filtering algorithm is inherently parallel and can be implemented on a network of processors.

The discrete model concepts of consistent and unambiguous are generalized to include weighted "fuzzy" and probabilistic interpretations. They [24] prove that assigning every possible interpretation to every object and then discarding incompatible interpretations until no more can be discarded yields a greatest set of interpretations. This set is then used as the starting point for a probabilistic relaxation process which converges towards the most probable labeling.

This probabilistic relaxation process can be represented by a series of matrix multiplications. Repeated iterations of this multiplication process have the effect of updating the probability estimate. Rosenfeld *et al.* [24] did not demonstrate that this will converge in all cases. Furthermore, it was observed by them that the convergence can be quite slow.

The theoretical basis for relaxation labeling is discussed by Haralick and Shapiro [8, 9]. They present a different notation from Rosenfeld *et al.* and a look-ahead operator which makes use of the set T, the unit-unit constraint relation presented in Section 1.1.2.2. They show that the set T has consequences in terms of implementation in that only units which constrain each other need to be stored and manipulated.

Other research in labeling involves the development of different operators for probabilistic relaxation [21, 28]. As well, researchers have looked at other image processing applications which utilize relaxation labeling techniques[27, 11, 20]. Several authors have done surveys of the literature on relaxation labeling or comparisons of the various techniques [31, 16, 22, 12, 1].

Kittler and Illingworth [16] note that twice as many papers in the field of labeling were published in the period 1981 to 1985 than all papers published before that period. They also note that although relaxation has long been recognized as highly parallel, there has been little literature regarding practical hardware implementation.

Since this thesis is concerned with MIMD parallel implementations of the consistent labeling problem, the next few sections will examine in more detail some papers published recently concerning parallel architectures or algorithms for the labeling problem.

2.2 Overview of SIMD Techniques

Gu *et al.* [5] describe several VLSI architectures for the Discrete Relaxation Algorithm (DRA). They state that, owing to the high computational cost including time complexity and data communication cost, a conventional hardware architecture implementation is not feasible.

They reformulate DRA as a parallel computational tree and use a multiple tree-root pipelining scheme to reduce the time complexity of this problem. Their architecture, DRA3, uses a dynamically re-configurable highly parallel routing scheme. With it, they are able to claim $O(nm)$ time complexity (n objects, m labels).

Kamada *et al.* [14] propose a parallel architecture for relaxation which can be applied to general use. They remark that previous parallel implementations are specialized to specific applications, such as labeling images at the pixel level. They propose a processor with a simplified control structure to minimize data communication between objects and their neighbors. The processor implements this simplified control structure with a round-robin communication architecture.

McCall *et al.* [19] compare various parallel computer architectures and problem solving strategies for the consistent labeling problem. They compare the various approaches by simulating the various multiprocessor architectures using a Pascal program. The simulation is set up so that the parameters affecting the problem can be varied and the results compared.

Some factors they considered were: the type of intercommunication network and the passing order of the objects. Architectural factors which could influence performance were:

1. The number of processors. This is the total number of processing elements in the system.

2. The diameter of the architecture. This is defined as the maximum distance between any two processors in the system.

3. The average distance between the processors. This is the distance between processing elements, on average, throughout the entire system.

4. The average distance vector. A distance vector is an n-dimensional vector, where n is the number of processors in the system. A processor with a distance vector (dv1, dv2, ..., dvn) has dv1 processors reachable by a path of length 1, dv2 processors reachable by a path of length 2, and so on. The average vector is composed of the distance vectors of all processors in the system.

Their conclusions were that an interrupt system is better than a polling system in terms of overall performance. They also show that it does not matter to which processor the problem is initially sent. Within a particular architecture class, the diameter is a good predictor of performance, with a larger diameter giving poorer performance.

When they compared different architecture classes they found that, using the *best* architectures from the architecture class, few architecture classes perform significantly worse than the class with the best performance. The architectures which did not perform well had either a large diameter or few buses.

This implies that, for *reasonable* performance, the choice of parallel architecture for the consistent labeling problem is not critical, providing that the chosen architecture performs well within its class. That is to say, the consistent labeling problem should perform comparably on any suitably "good" MIMD parallel architecture.

Yalamanchili and Aggarwal [33] discuss the organization of multiprocessor architectures. Their focus is on how to interconnect multiple processing units such that the special communication requirements of image processing are efficiently handled. They propose a system organization centered around a class of interconnection networks and a global bus. Control schemes for realizing the intertask communication typical of image processing problems are developed which are claimed to be simple, distributed, and efficient.

Leung *et al.* [17] develop a general computational model for relaxation labeling and uses this model to develop parallel algorithms for two SIMD computer architectures. Also, Leung [18] has a good comparison and summary of the different notation schemes used by various leading researchers in the field of relaxation labeling.

2.3 Overview of MIMD Techniques

Not much literature has addressed the application of MIMD techniques to the problem of labeling. Siegel *et al.* [26] describe a large scale multiprocessor system designed at Purdue University which can be dynamically reconfigured to operate as one or more independent SIMD and/or MIMD machines. The objective of this design was to implement a system which was a compromise between flexibility and cost-effectiveness. The authors demonstrate how this architecture can show significant improvement over other conventional systems. Such a system may provide advantages in the field of object labeling by combining techniques of both SIMD and MIMD algorithms.

Ullmann *et al.* [29] present a theoretical paper that shows that tree pruning techniques can be implemented using various forms of parallelism to reduce the elapsed time for solving the consistent labeling problem.

They state that the practical usefulness of a consistent labeling formulation depends on the actual constraints that are employed. They cite as an example, the subgraph isomorphism problem, where examination of graph-theoretic factors has experimentally yielded greater efficiency than can be obtained using the simplest consistent labeling formulation. Kirousis [15] demonstrates how application dependent knowledge in the field of labeling line drawings can be used to reduce computational complexity.

The approach that Ullmann *et al.* [29] use to develop their parallel formulations is that they do not wish to deal with application-specific refinements that may improve efficiency, but rather with introducing parallelism to reduce the elapsed time for *all* consistent labeling problems.

This thesis also takes this approach and attempts to develop general parallel algorithms for the consistent labeling problem. Refinements particular to a specific application or targeted to a specific architecture are possible, but are not developed in this thesis.

The parallel algorithms presented by Ullmann *et al.* [29] use a ***constraint propagation algorithm*** which causes the deletion of inconsistent labelings to cause the deletion of other inconsistent labelings, which causes the deletion of others, and so on. They use the unit-unit constraint relation, T, and the unit-label constraint relation, R, as described in Section 1.1.2.2 as the basis for their algorithms. The algorithms presented in this thesis also use the sets T and R for constraint satisfaction.

One algorithm they present requires the design of a specialized combinational constraint network which exists outside of the CPU of the processor and performs logic operations for constraint satisfaction in parallel, functioning as a co-processor. Another solution they propose is to subdivide the labeling problem into M subproblems and then solve these separately on M independent processors.

The method of subdividing the problem in their paper is quite simple but does not make optimal use of the M processors since not all processors take equal time to complete their subtree search. In answer to this, they propose a computer interconnection network which would allow idle processors to request additional work from other processors which have not completed their subtree.

This thesis expands on these ideas by attempting to structure the tree search in such a way as to reduce the elapsed time to find a solution. The task subdivision used by this thesis is also more complex in that it attempts to use information in the problem itself to determine how to partition the problem.

Chapter 3

A MIMD Approach: Recursive Descent with Deepening

3.1 Introduction

One way that MIMD architectures differ from SIMD architectures is in the approach a programmer takes to develop a parallel algorithm for the specific class of machine. A SIMD algorithm requires that the processing elements execute the identical instruction on their piece of data and then transfer the result to the appropriate processing element in a "lockstep" fashion. Designing a SIMD algorithm involves analyzing or defining the problem to identify which aspects can be expressed in a regular, structured manner. The program then consists of the control instructions and control mask which tell the processing elements when to execute an instruction and when to pass data. The control program is dependent on the target architecture.

MIMD algorithms, on the other hand, make use of the "multiple instruction" nature of the architecture. What this means is that not all processing elements must behave in an identical manner. A problem is broken up into smaller subproblems, which are solved in parallel to produce a final solution. These smaller tasks need not be similar in size or in complexity. The design of the algorithm consists of deciding how to break up the problem domain to distribute it effectively.

3.1.1 The Myrias Model of Parallel Computation

Myrias Research Corporation has developed a high speed MIMD parallel computer architecture called the *SPS-2*, which can provide supercomputer performance on various problem domains. The underlying hardware is a "card" consisting of four Motorola MC68020 microprocessors, each with 4 megabytes of local memory and a MC68881 floating point co-processor. The processors communicate with each other through a bus hierarchy. Processors on the same card communicate locally over a bus. Cards are in turn connected together to form a cage, and cages are connected together to expand the system to the desired number of processing elements in multiples of 64 processors.

One design goal of the Myrias machine was to isolate the programmer from the details of the operating system. Aspects such as task allocation, load leveling, communication and synchronization between processing elements, and memory management are all handled transparently in that the programmer is not aware of how the system accomplishes these tasks.

To incorporate parallel programming, Myrias has added an extension to the C programming language. This extension is the inclusion of a *pardo* instruction which specifies that the code contained in its scope is to be executed in parallel. This introduces the concept of parallel *tasks*. It is possible to request that more things be done in parallel than there are processors available to handle it. This permits arbitrarily large problems to be handled on a limited domain of processors.

Another aspect to the Myrias architecture is that there is no shared memory between tasks. When a pardo is executed, the task executing the pardo is called the *parent* task and the sub-tasks it generates are called its *children*. Each child task inherits an identical copy of the memory space from its parent. It is then free to read or write to this memory as it pleases. When all child tasks have completed, the memory is merged back into a single memory space in the parent and execution continues. The merging rules are such that if only one child modified a memory location, that is the new value of that location in the parent's memory. If two or more children modified a specific location, then the value of that location in the parent is undefined, unless they both modified it with the same value.

Since the execution of a pardo involves communication between processors to distribute the program to local memory and to merge memory images on completion, it is desirable to only do a pardo if there is sufficient work for the tasks to justify the parallel task overhead. With these considerations in mind, the approaches presented in this thesis were developed.

Refer to Appendix D for further information on the Myrias SPS-2 and the PAMS operating system.

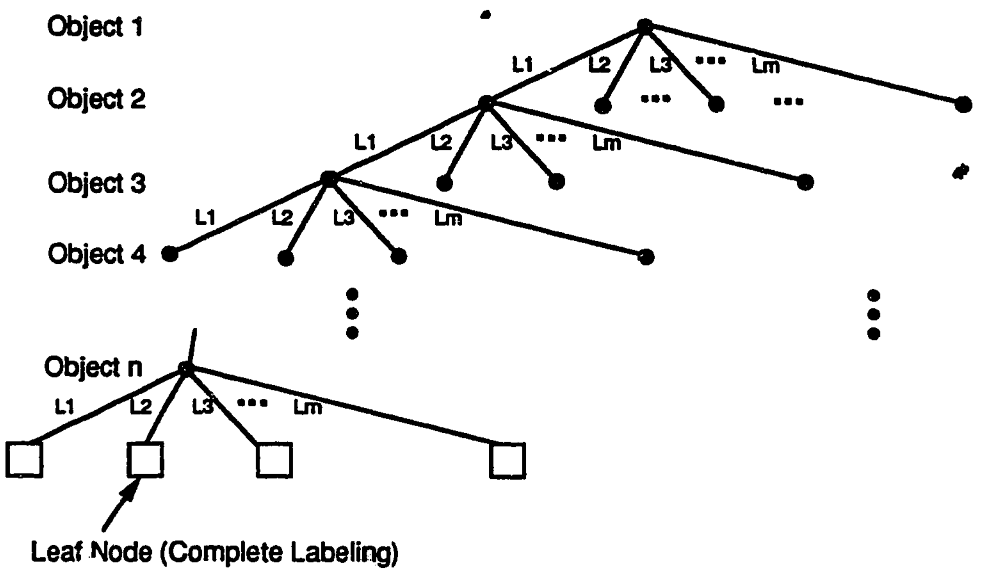


Figure 2: Example of a search tree for n objects and m labels

3.1.2 Choice of Approach

In general, the labeling problem has a complexity which is exponential in nature. A given problem with n objects and m labels has a search tree with n levels and a branching factor of m. This means that there are potentially m^n terminal nodes, each corresponding to a specific labeling. This is illustrated in Figure 2.

This "worst case" for a problem occurs when a set of objects is not constrained in any way, and each label is a valid label for each object. In this case it is not possible to reduce the search tree. However, this is not meaningful in a practical sense.

A parallel approach to this type of problem involves partitioning the search space in some way and then using several processing elements to simultaneously calculate a portion of the solution. If a problem would take time t_s on a sequential machine, and it takes t_p on a parallel machine, then the speedup is defined as t_s/t_p. The speedup is ideally k, the number of processors. That is, if you use 10 processors, you desire a 10-fold improvement in performance. This k-fold improvement is called *linear* speedup. In some cases it is possible to achieve *super-linear* speedup [13], where the actual performance improvement is greater than k. Such super-linear speedups are possible because they are measured relative to the performance of the same algorithm on one processor. This does not consider the behavior of the *best* sequential algorithm. In general, k is an upper bound on the expected speedup. Statistically, super-linear speedup is possible, but it cannot be the expected result.

Theoretically, linear speedup of an exponential problem is not considered interesting [30] in that the resulting complexity of the problem is still exponential. In practice, however, any speedup is beneficial. Furthermore, in real labeling problems, the search space can be reduced. Objects mutually constrain each other and not all labelings of a set of objects are equally valid. The problem then becomes how to best structure the search algorithm to reduce the search tree *most* of the time, given a realistic problem.

This chapter looks at an approach based on a "branch-and-bound" tree search technique. This method searches the tree to a certain depth using a sequential recursive descent algorithm and then recursively launches parallel tasks on the branches which have not been eliminated. The behavior of this algorithm is illustrated in Figure 3.

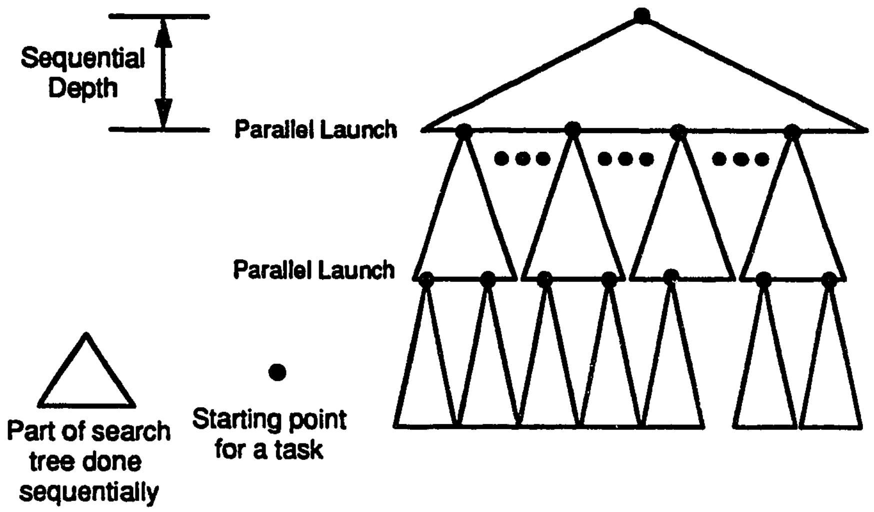

Sequential Depth
Parallel Launch
Parallel Launch
Part of search tree done sequentially
Starting point for a task

Figure 3: Behavior of the recursive deepening algorithm

3.2 Recursive Descent with Deepening

3.2.1 Description

One standard method of traversing a tree is known as recursive descent or depth-first search. As applied to the labeling problem the algorithm for this method follows:

```
descend(L,u)
labeling L;
object u;
{
     for (each label for object u) {
          assign label to object u in labeling L;
          if (label violates any constraints)
               continue; /* next label */
          else if (u is the last object)
               print labeling L;
          else
               descend(L,u+1);
          }
          return;
}
```

The deepening technique searches only to a fixed depth, which is determined beforehand. At that point, in a sequential implementation, a decision is made as to the order in which to search the branches which were not eliminated.

3.2.2 Parallel Approach

A straightforward parallel implementation of the recursive-descent with deepening approach is to search the tree using a sequential depth-first algorithm to a specified depth and then launch parallel tasks on the branches which were not eliminated. Each of these tasks then recursively searches its piece of the tree to a specified depth and launches more parallel tasks on the remaining branches. The number of parallel tasks this can generate is unknown prior to execution. At the terminal nodes, any remaining labelings are printed out. The algorithm for this follows:

```
descend(prime_task, partial_labeling, u, depth)
boolean prime_task;
labeling partial_labeling;
object u;
integer depth;
{
     if (u is deeper than the sequential search depth) {
          add partial_labeling to list L;
          return;
     }

     for (each label for object u) {
          assign label to object u in partial_labeling;
          if (label violates any constraints)
```

```
                continue; /* next label */
            else if (u is the last object) {
                print labeling;
                return;
            }
            else
                descend(FALSE,partial_labeling,u+1,depth);
        }
        if (not prime_task)
            return;
        /* At this point we have a list of partial labelings, L */
        pardo (i = 1 :  number of partial labelings in L) {
            /* apply ''descend'' to each labeling in L */
            descend(TRUE,L(i),u+depth,depth);
        }
        return;
    }
```

There is a performance penalty when launching parallel tasks due to the overhead necessary to start and stop the processing elements. This approach only launches parallel tasks on those branches of the tree which have not been eliminated in the sequential recursive descent phase. By avoiding the startup of unnecessary tasks, this penalty is reduced. Also, it is not desirable to have each processor do too small a task. In this case, the parallel overhead outweighs the computation and it would be faster to use a sequential machine. Therefore, it is desirable to determine the proper "granularity" for the problem.

It is not clear how to best choose the depth to search sequentially. This would depend on the specific architecture and the overhead to launch a parallel task versus the speed of its processing elements. At one extreme, if the sequential search depth is n where n is the number of objects, then the execution of the program is totally sequential (coarse granularity). That is, there is no parallel launch phase.

At the other extreme is a sequential search depth of 1 (fine granularity). This means that at each node of the tree, a parallel task is launched on any labeling which has not been eliminated. This has the potential to generate many parallel tasks and so the startup overhead becomes more significant.

The Myrias architecture permits the multi-tasking of several parallel tasks on a single processing element. A task must wait for its children to complete before it can continue. This means that it is in an idle state until it can proceed. This idle time can be used by other tasks and efficient use of the processors can be realized. Fortunately, this "load leveling" is performed by the operating system. The programmer has no direct control over how tasks are allocated to processors.

3.2.3 Object Ordering

Another consideration for this approach is the order in which labels are assigned to objects. Since the objects mutually constrain each other, the set of possible labels for an object depends on the labels already assigned to objects higher in the search tree. That is, the assignment of a specific label to an object affects those objects which are constrained by it and which have not yet been assigned labels.

One approach is to use a random or arbitrary ordering without consideration of how the objects constrain each other. This has the advantage of being simple, but also has the potential to generate large search trees.

A second approach is to look at which objects constrain each other and [illegible] this information to attempt to order the objects such that the highly constrain[illegible] objects are labeled first. These "highly constrained" objects affect the gre[illegible] number of other objects and thus labeling them early should result in a reduce

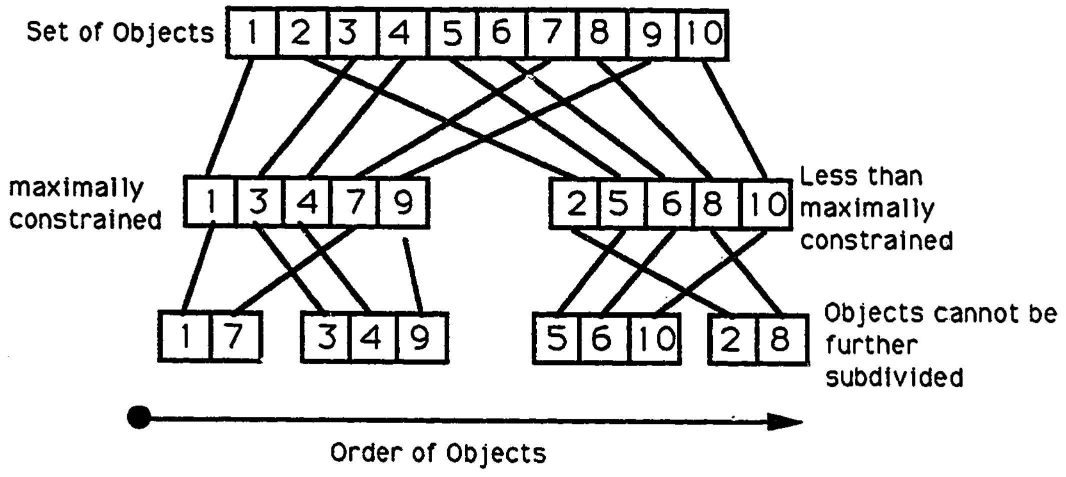


Figure 4: Behavior of the unit-constraint ordering algorithm

search tree. The information about which objects constrain each other is already given in the unit-constraint relation T, described in Section 1.1.2.2. This set is used as the basis for the *Unit Constraint Ordering Algorithm.*

This approach looks at how many other objects each object constrains. The objects are split into two groups: those objects which constrain the most other objects (maximally constrained) and those which are less than maximally constrained. Then, recursively, these two groups are broken down by looking at which objects in the set constrain the most other objects in the set, again producing two groups. The recursion stops when the set cannot be broken down farther. Figure 4 illustrates the behavior of this technique.

The third approach is to use application dependent knowledge to order the objects in a way that will reduce the search tree. For the domain of labeling line drawings, Rich [23] provides a simple algorithm which orders the objects (lines) based on the spatial relationship of objects in the scene.

1. Start with the line segments found by moving clockwise around the boundary of the scene, starting at any vertex.
2. Add line segments which share a common vertex with any boundary edge.
3. Working towards the centre of the scene, keep adding objects. Ensure that each line segment is connected to one that has been previously labeled.

By always labeling a line which is connected to one which has already been labeled, this algorithm makes good use of mutual constraints. This technique is referred to in this thesis as the *Spatial Relation Ordering Heuristic.* For the purposes of this experiment, the ordering was done manually. This ordering technique could be automated, but the development of the program to do this was not considered within the scope of this thesis.

3.3 Experimental Results

3.3.1 Description

To investigate the effect of the three object orderings (random, unit-constraint heuristic, and spatial-relation heuristic) on the recursive descent with deepening algorithm, the following experiment was devised:

1. Use four test cases of increasing complexity—9, 20, 33, and 54 objects. This would show the behavior of the algorithm on problems of different sizes. The 9, 20, and 33 object cases are combinations of a simple cube. The 54 object test case is an attempt to demonstrate a possible "real" scene after

processing to extract edges. Refer to Appendix A for an illustration of these test cases.

2. For each test case, vary the depth of the sequential search phase and the number of processors available. The depth was varied from 1 to n. Runs were done on a domain size of 1, 2, 4, 8, 16, 32, 48, and 64 processors.

3. Ensure that the same program is run in all cases. The only difference between runs is in the ordering of the objects.

4. The performance measure of interest is elapsed execution time. This is sometimes referred to as "wall clock time". Efficient use of the parallel resources is not measured. This takes the view that processors are a resource to be used, similar in principle to memory. There should not be a penalty associated with not using a resource. Here, the interesting measure is "how fast is it".

5. The results are to be compared to each other, not to a benchmark or similar program on another machine. There was no data available in the literature regarding labeling benchmarks. Also, it was decided to investigate the relative performance of the various orderings within a MIMD framework. It was not considered important exactly which architecture was used. The Myrias SPS-2 met the criteria of being available and providing an environment for general purpose MIMD parallel computing.

3.3.2 Results and Analysis

The graphs starting on page 38 contain the results for the 4 test cases. Note that the graphs do not all use the same scale for the "y" (elapsed time) axis. Figures 5, 7, 9, and 11 show the elapsed execution time as the number of PE's is varied. These graphs illustrate the behavior of the algorithms as more processors are added to the problem. Figures 6, 8, 10, and 12 show the elapsed time for the 4 cases as the sequential search depth is varied. The effect of increasing the task granularity by increasing the depth can be studied from these graphs.

The elapsed time results for the 9 object case (Figures 5 and 6) show that for a problem this small, it is best to do the processing with a straight sequential algorithm (depth=9, number of PE's=1). This is evident from the unit constraint and spatial relation cases where the performance is made worse if parallelism is introduced. Thus in these cases, the parallel task overhead adds a significant amount to the overall execution time. It is interesting to note that in these two cases, the performance actually decreases for more than two processors (Figure 6(b,c)).

The random case does benefit by the addition of up to about 16 processors, with some slight improvement as more are added. This illustrates that the rapid exponential growth of the search tree is already evident for cases as small as 9 objects. The improvement in this case is due to the "bushiness" of the search tree providing sufficient work for the processors.

The results for the 20 object case (Figures 7 and 8), show the dramatic difference between the random and heuristic orderings. All three orderings benefit initially from the addition of more processors, as is evidenced by the steep initial slope in Figure 7 for all three cases.

From Figure 8, a general conclusion seems apparent from the greater depth required before any decrease in performance is observed. As the quality of the sequential search algorithm improves, it is desirable to search deeper into the tree sequentially. This is expected since the better ordering algorithms produce a reduced search tree, so it is possible to search deeper with the same amount of work. The implication of this is that fewer tasks are generated resulting in better performance on a smaller processor domain size and reduced parallel overhead.

As the problem size is increased to 33 and 54 objects, it becomes difficult to gather results for the random ordering case. For example, with 33 objects and random ordering, each run was taking approximately 44 (!) hours to complete. The 54 object case required even more time. This demonstrates the importance of ordering the objects. It was not possible to gather sufficient data points to produce performance graphs for these cases. Figures 9 and 10 show results for the unit constraint and spatial relation heuristics.

From Figure 9 it is interesting to note that even with a small amount of parallelism (e.g. depth=28) there is an improvement over the sequential case (depth=33). It is apparent that a smaller depth is desirable, but it is worth noting that for larger problem sizes *any* parallelism is useful.

Figure 10 shows that the choice of sequential search depth is becoming less important. In both cases, performance is similar for depths up to about 5 for the unit constraint heuristic (a) and up to approximately 10 for the spatial relation heuristic (b). This figure also demonstrates the earlier observation that as the quality of the ordering algorithm is improved, the sequential search depth can be increased.

The 54 object case (Figures 11 and 12) behaves much like the 33 object case, with the main difference being total execution time. The shape of the graphs is similar between the two test cases, indicating that the algorithm's behavior is becoming stable. This means that it is reasonable to expect larger problems to exhibit similar performance characteristics, but with increased elapsed time results.

For 54 objects the same observation about the choice of depth can be made. It would appear that the key factor in the choice of search depth is that there be enough tasks generated to utilize the available parallel resources. As the problem size increases, so does the size of the search tree. The available processors "saturate" much earlier since it is not possible to eliminate many branches of the tree early in the search. Once this saturation has been reached, performance is not greatly affected by slight changes in depth until the tasks begin to become sufficiently large. At this point, not enough tasks are generated to utilize the processors effectively.

Figures 13, 14, 15, and 16 show a speedup graph and a relative performance graph for each of the test cases.

The "quality" or relative performance is measured by comparing the execution time for a test case (t_p) to the "best" execution time (t_{best}). This is the run with the lowest elapsed time over all runs for that test case. For example, if the best execution time was 1.4 seconds, a point on the graph with a value of 0.5 would be half as fast as the best case and so would have taken 2.8 seconds. This sequence of graphs is meant to compare the algorithms using elapsed time as the metric. The speedup curves show impressive results for the randomly ordered cases. When absolute performance is compared, a better indication of the behavior of the random ordering is obtained.

The 9 object case (Figure 13) shows that the random case benefits by up to 16 processors after which the speedup begins to decrease. Even so, the speedup for 16 processors is poor (1.8). The relative performance graph (b) shows that the random case performs worse than the other two orderings.

Figure 14(a) shows that the speedup for the random 20 object case is quite impressive. The other two orderings show some slight speedup initially, but then do not improve as more processors are added. The relative performance graph (b) shows that in terms of total execution time, the random case is only about 0.15 times as good as the best time. The unit constraint ordering performs almost half as good, or requires approximately twice as long as the spatial relation heuristic.

As the problem size is increased to 33 and 54 objects (Figures 15 and 16), the speedup graph for the two heuristic orderings improves. This indicates that larger problems are able to utilize the parallel resources more effectively. Again, random ordering gives the best speedup, but when elapsed time is compared, the random case does not perform well. This is seen by either comparing the relative performance graphs (part (b) of Figures 13, 14, 15, and 16) or by noting the difference in the elapsed time axis of the results graphs (Figures 5, 6, 7, 8, 9, 10, 11, and 12).

Figure 17 is a comparison of speedup for all three orderings on each of the test cases. This demonstrates how the orderings behave as problem size and number of processors is increased. Here it is evident that increasing the problem size with any of the orderings will result in an improved speedup response. The significance of this figure is that it demonstrates that the algorithms are able to use more parallel resources as the problem size is increased. The speedup curves for the 33 and 54 object cases do not appear to be leveling off, which indicates that those cases could benefit from the addition of more processors. This shows the viability of this approach and demonstrates a potential for its application to larger problem sizes.

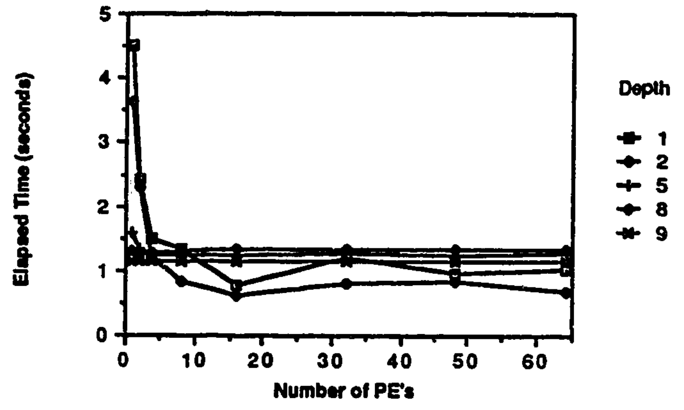
3.3.2.1 General Observations

In general, it is possible to conclude that for sufficiently large problems, the choice of sequential search depth is not critical. It is only necessary to ensure that the parallel tasks have enough work (suitable granularity) and that there are enough tasks generated to use the processors available. The determination of this depends on the specific problem and MIMD architecture being used.

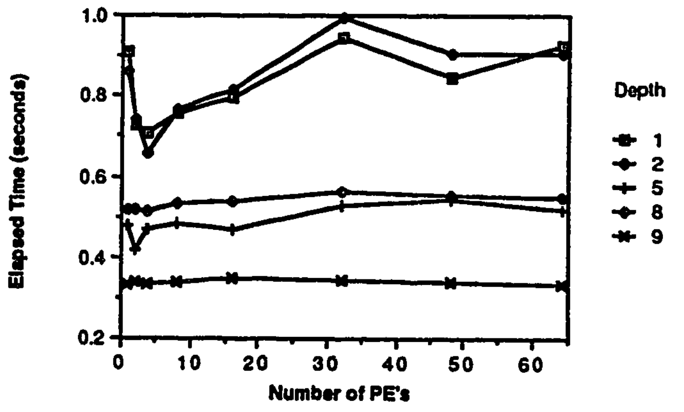
The ordering of objects is important with this approach. Random ordering is poor. However, in some cases, random ordering may be all that is possible. This occurs if the objects either constrain each other equally or if the set of constraints is of limited value in reducing the search tree, perhaps due to the difficulty of defining the constraint set for a particular problem. In this case, it is significant that the random case makes good use of parallelism, which is evident in the speedup curves for this case (Figure 17(a)). Any speedup of a real problem is useful.

The unit constraint set T is available as part of the problem specification and it can be used in instances where specific knowledge about the problem is not available to help order the objects. In general, however, application dependent knowledge is required to further improve the algorithm's performance.

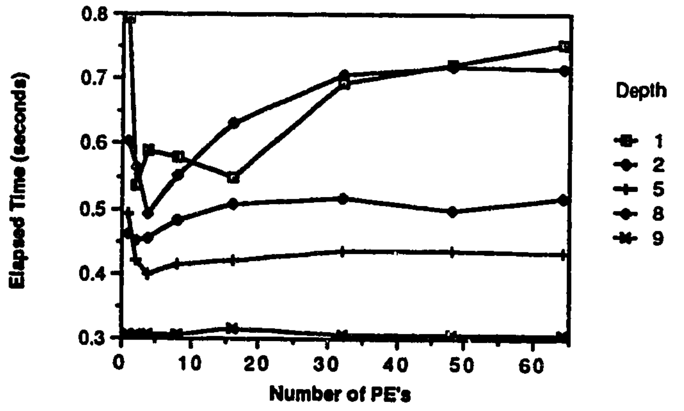
The results demonstrate that object labeling is possible using recursive descent with deepening. Significant speedup is possible for sufficiently large problems. As the problem size increases, it becomes necessary to improve the quality of the techniques used to reduce the search tree for the algorithms. This is important to reduce the effects of the exponential growth of the search tree which drastically affects performance and parallel speedup. This experiment varied the object ordering, but results indicate that this approach could also benefit from other techniques to prune the search tree [8, 29].



(a) Random Ordering

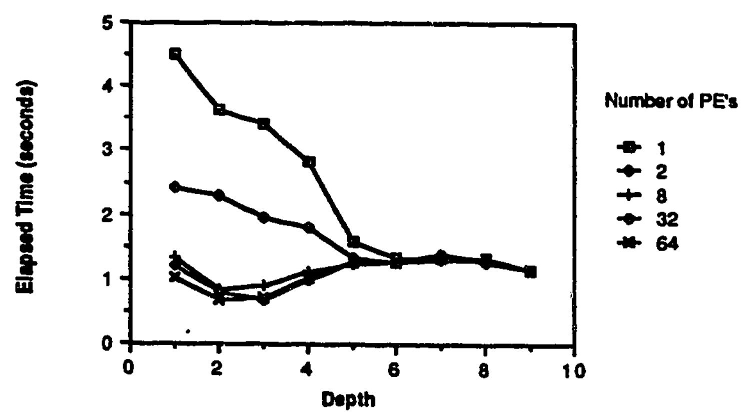


(b) Unit Constraint Heuristic Ordering

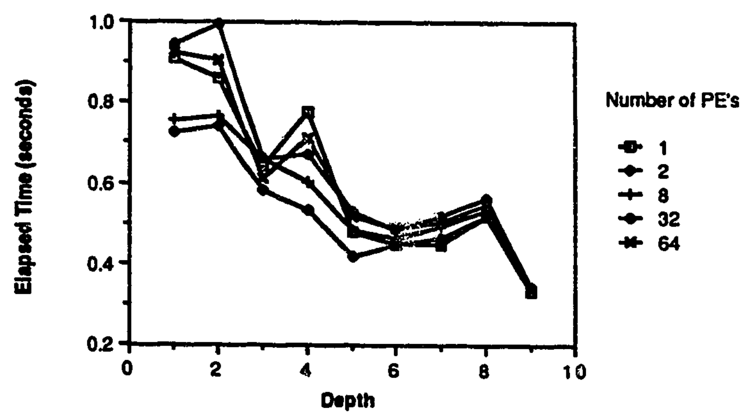


(c) Spatial Relation Heuristic Ordering

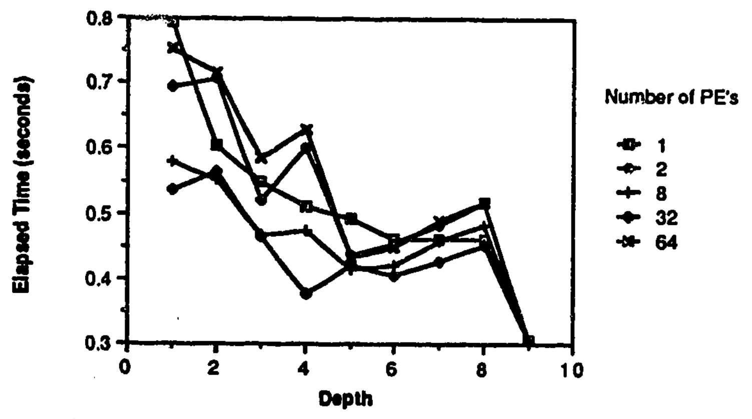
Figure 5: Timing results for 9 objects as number of PE's is varied



(a) Random Ordering

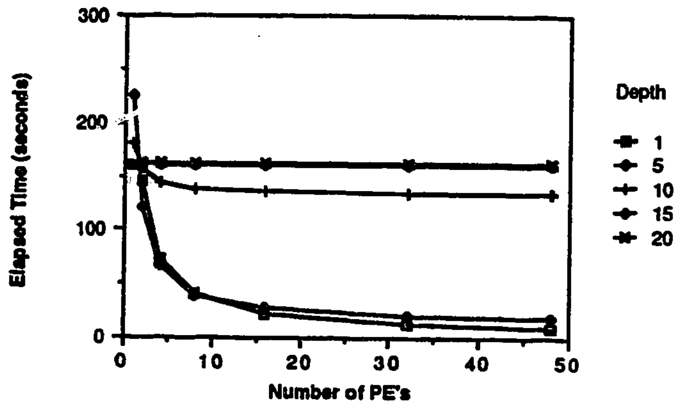


(b) Unit Constraint Heuristic Ordering

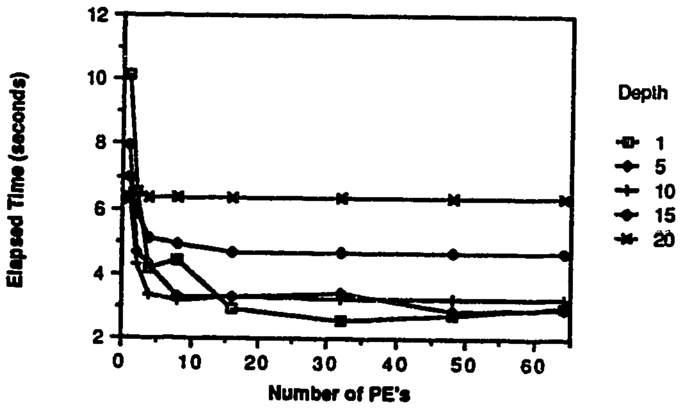


(c) Spatial Relation Heuristic Ordering

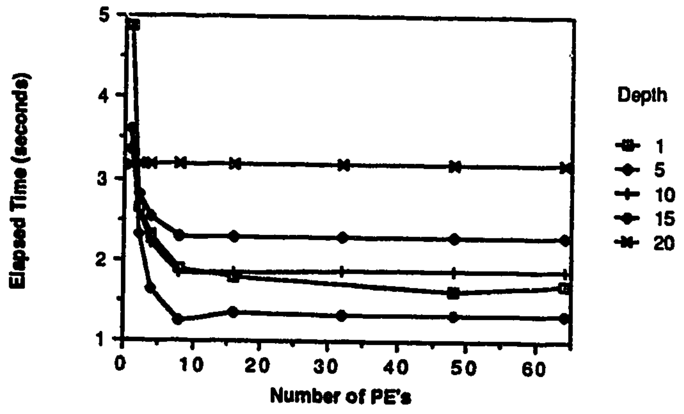
Figure 6: Timing results for 9 objects as sequential search depth is varied



(a) Random Ordering

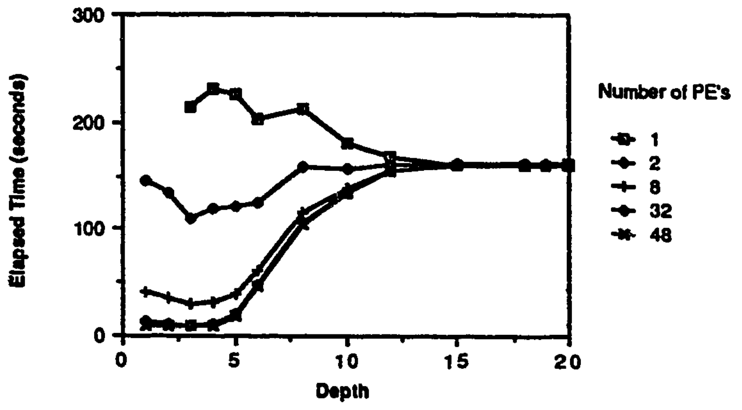


(b) Unit Constraint Heuristic Ordering

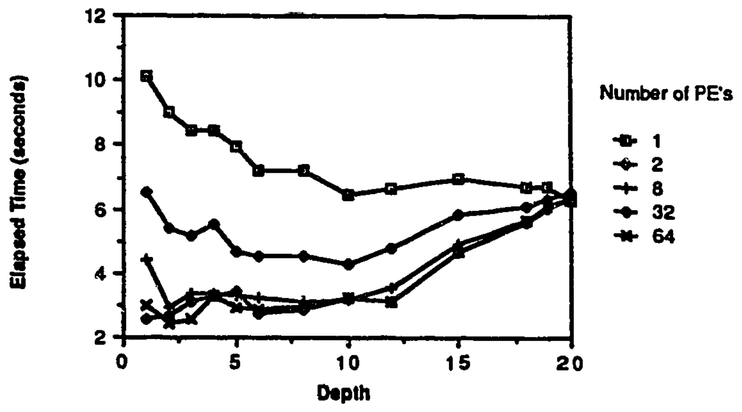


(c) Spatial Relation Heuristic Ordering

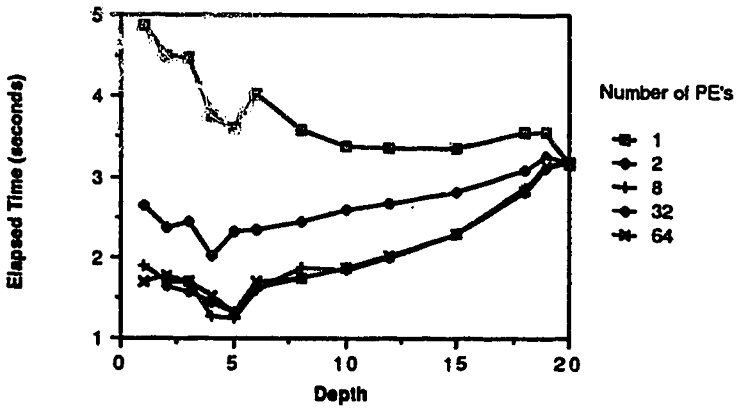
Figure 7: Timing results for 20 objects as number of PE's is varied



(a) Random Ordering

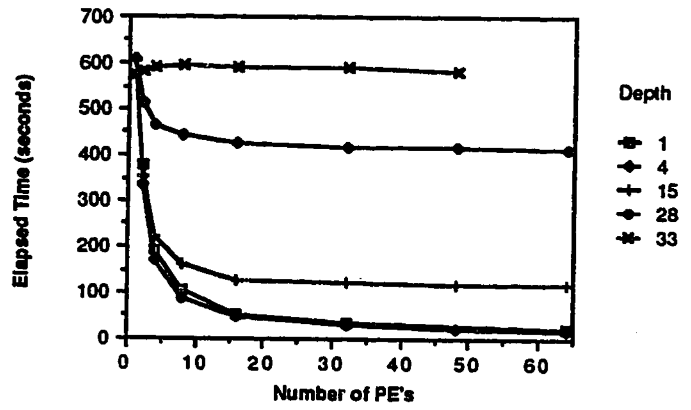


(b) Unit Constraint Heuristic Ordering

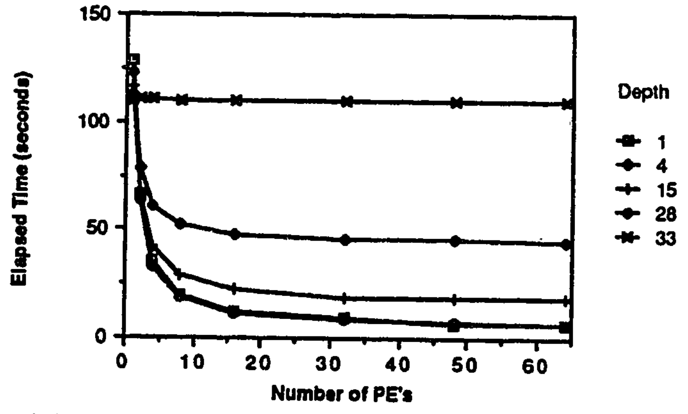


(c) Spatial Relation Heuristic Ordering

Figure 8: Timing results for 20 objects as sequential search depth is varied

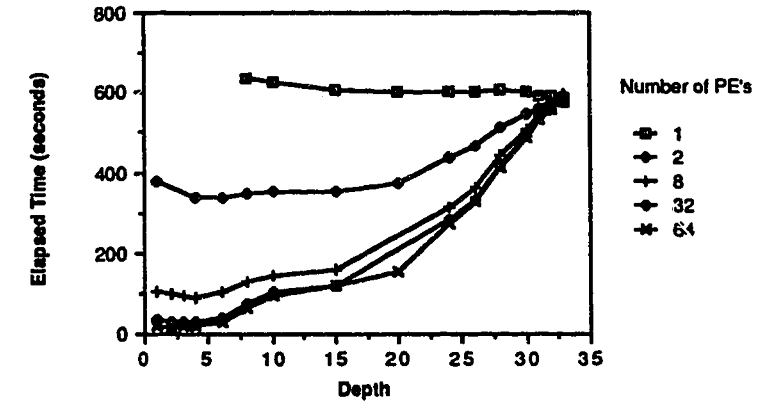


(a) Unit Constraint Heuristic Ordering

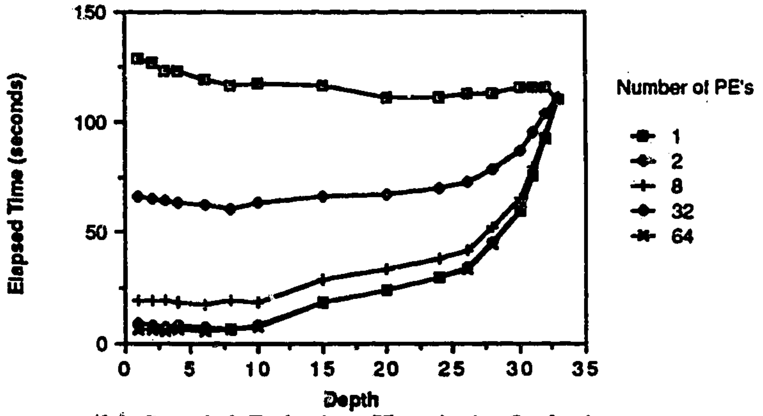


(b) Spatial Relation Heuristic Ordering

Figure 9: Timing results for 33 objects as number of PE's is varied

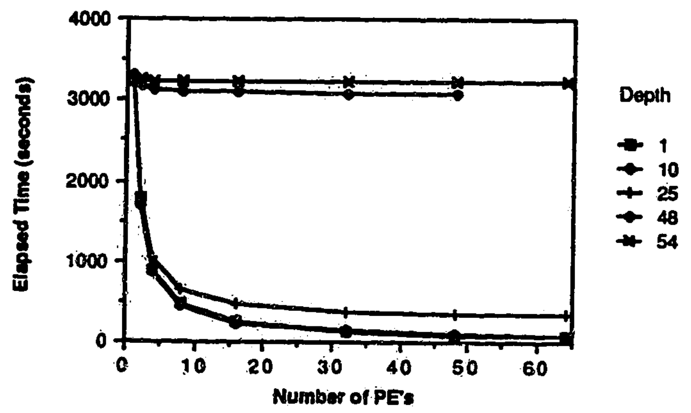


(a) Unit Constraint Heuristic Ordering

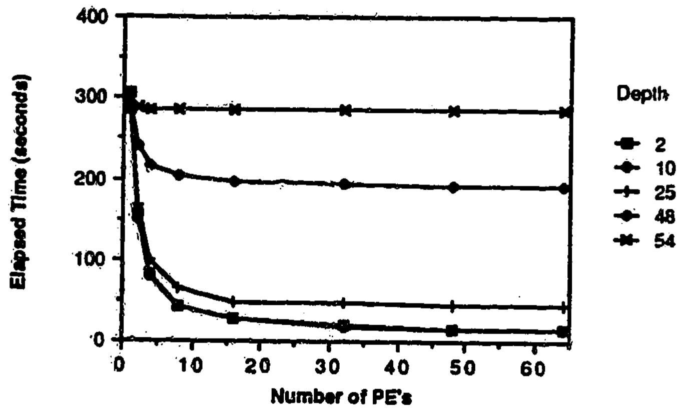


(b) Spatial Relation Heuristic Ordering

Figure 10: Timing results for 33 objects as sequential search depth is varied

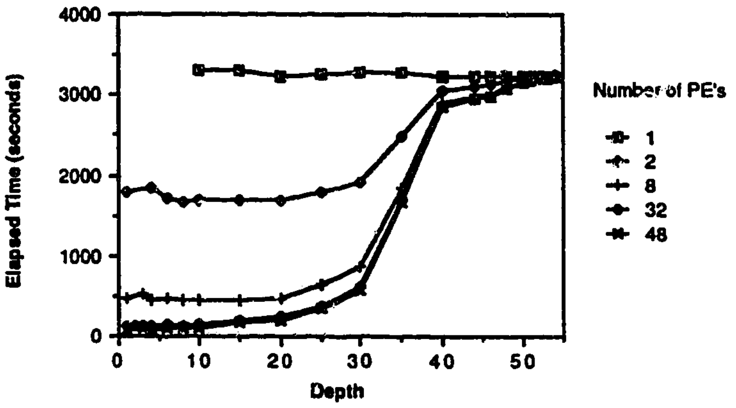


(a) Unit Constraint Heuristic Ordering

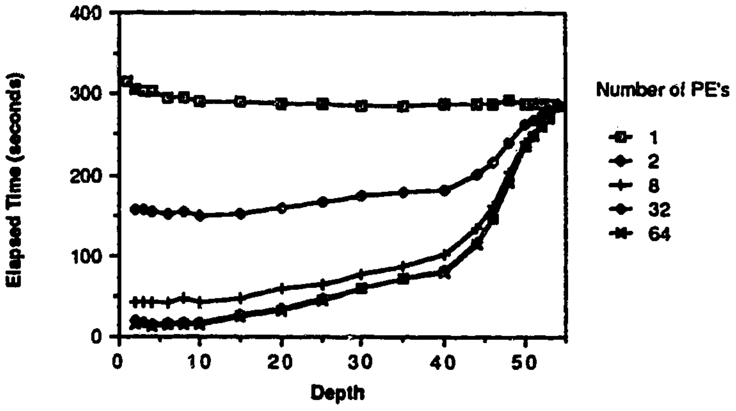


(b) Spatial Relation Heuristic Ordering

Figure 11: Timing results for 54 objects as number of PE's is varied

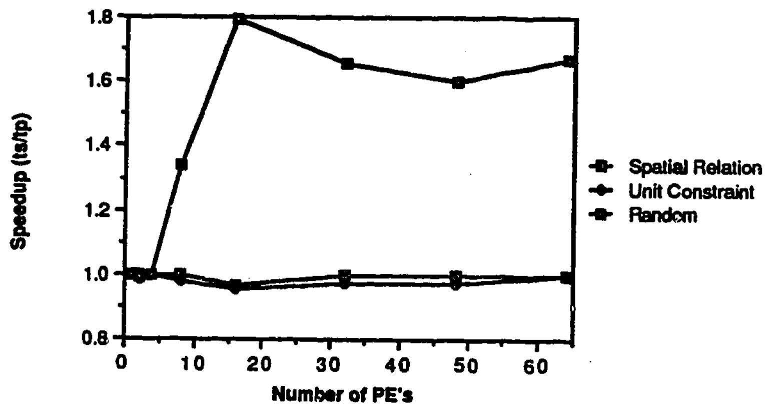


(a) Unit Constraint Heuristic Ordering

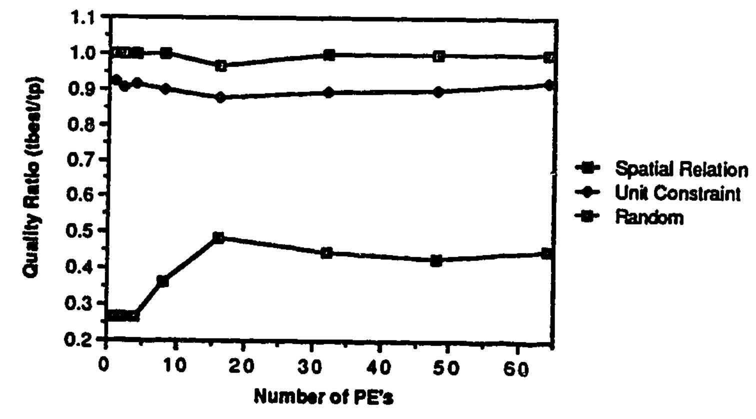


(b) Spatial Relation Heuristic Ordering

Figure 12: Timing results for 54 objects as sequential search depth is varied

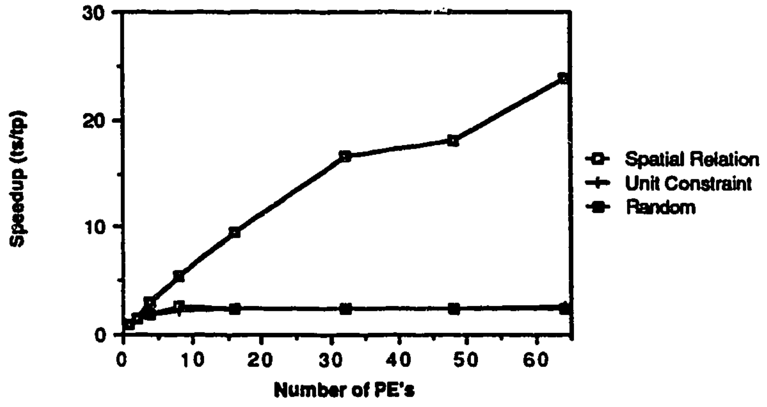


(a) Speedup

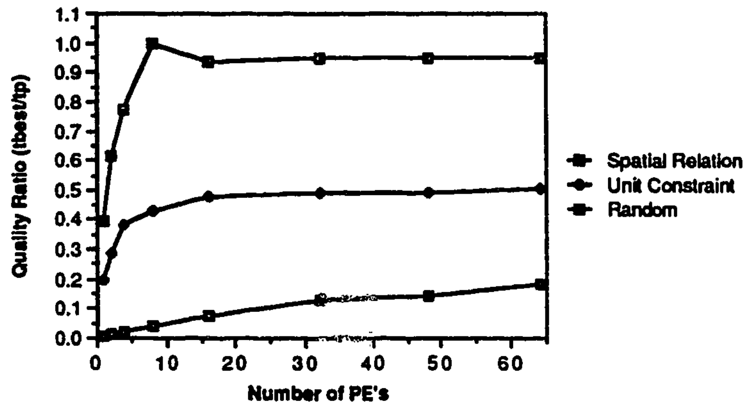


(b) Relative performance

Figure 13: Speedup and relative performance for the 9 object test case

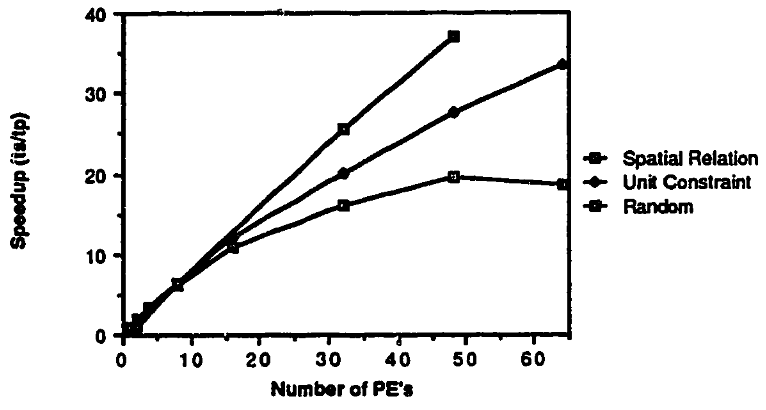


(a) Speedup

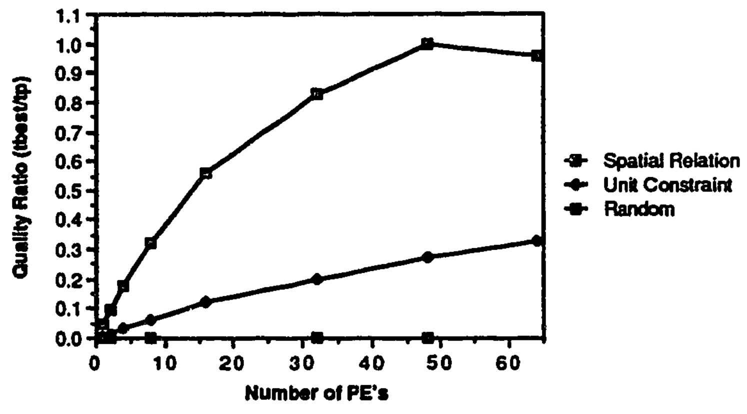


(b) Relative performance

Figure 14: Speedup and relative performance for the 20 object test case

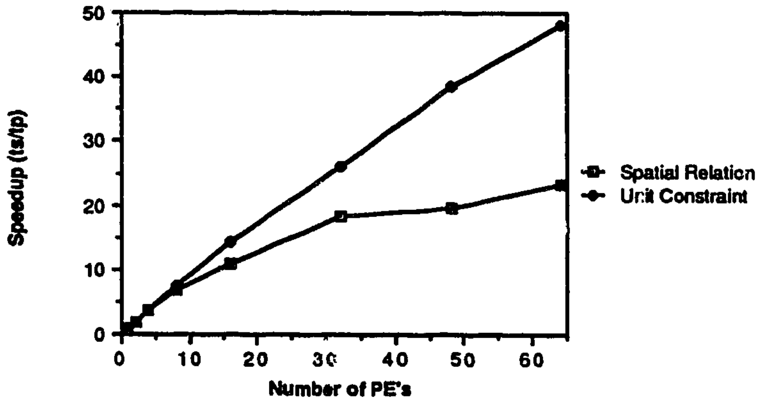


(a) Speedup

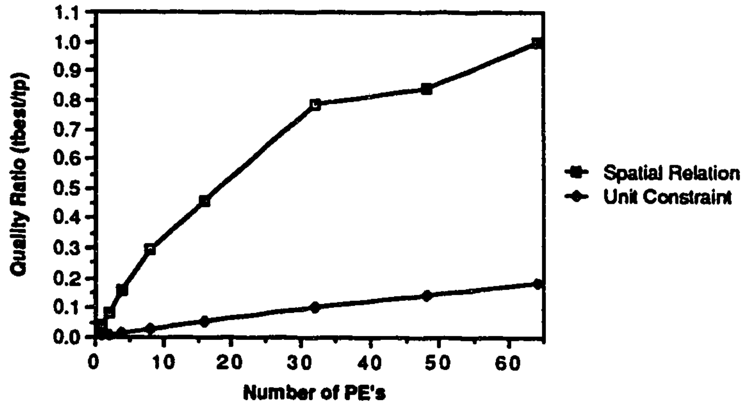


(b) Relative performance

Figure 15: Speedup and relative performance for the 33 object test case

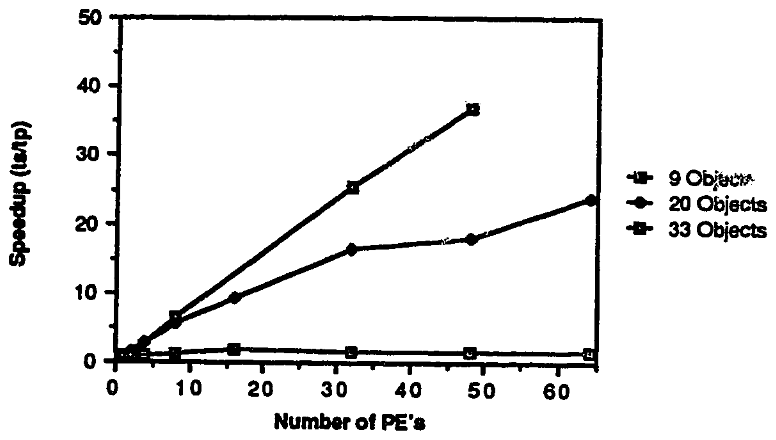


(a) Speedup

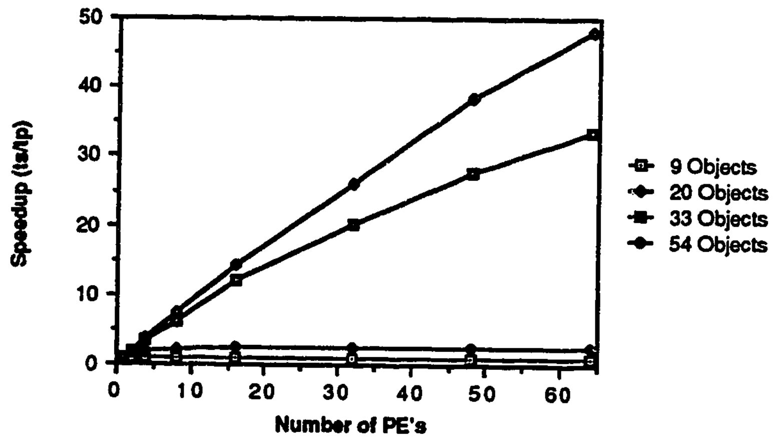


(b) Relative performance

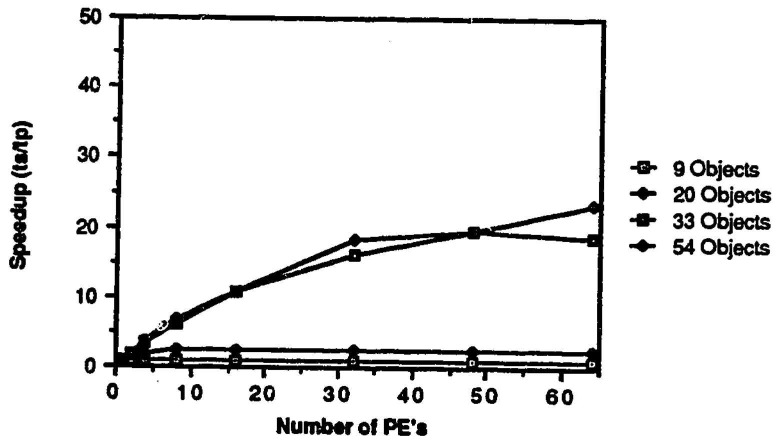
Figure 16: Speedup and relative performance for the 54 object test case



(a) Random Ordering



(b) Unit Constraint Heuristic Ordering



(c) Spatial Relation Heuristic Ordering

Figure 17: Speedup comparison of the three orderings

Chapter 4

Other MIMD Approaches: Split-Label-Merge and Eureka Jump

The two techniques presented in this chapter are provided as alternative parallel approaches to the consistent labeling problem. Both approaches are potentially good methods for object labeling on suitable MIMD architectures. Due to practical implementation limitations, neither approach proved practical on the Myrias SPS-2 architecture. The following sections discuss these algorithms and their behavior. Experimental results are presented to illustrate the viability of the approaches, but are not meant to be a rigorous treatment.

4.1 The Split-Label-Merge Technique

The approach described in Chapter 3 examines partial "paths" in the search tree and then launches parallel tasks on paths which remain viable. Conceptually, this corresponds to partitioning the search tree vertically.

One alternative approach is to subdivide the problem *horizontally*. This corresponds to dividing the set of objects to be labeled into smaller groups. Each group is then labeled in parallel to produce a set of partial labelings which are consistent for objects in that group. Then, in parallel, the groups are merged to produce a set of consistent labelings for all objects.

The desired behavior of this approach is to reduce the number of possible labelings by eliminating many labelings for the small groups. The potential size of a search tree is exponentially related to the number of objects. This relation is given by l^n, where l is the number of labels (branching factor) and n is the number of objects in the tree.

In the worst case where objects do not constrain each other, this technique will not have any advantage. However, in real applications, objects constrain each other and it is reasonable to expect that the number of partial labelings produced for each smaller group will be considerably less than the total number possible for all objects. Merging together these smaller sets of partial labelings is then a much smaller task. Many impossible labelings have been eliminated in the initial labeling phase. Refer to Figure 18 for an illustration of the overall behavior of this technique.

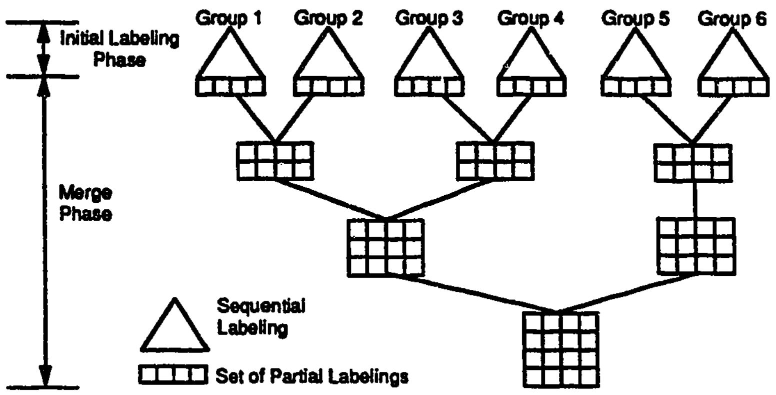


Figure 18: Overall behavior of the Split-Label-Merge algorithm

4.1.1 Choice of Sub-Groups

One important consideration for this algorithm is how to divide the initial set of objects into smaller groups. At one extreme is to have one object in each group, produce the valid labelings for those objects in parallel, and then merge the results to form the final solution. At the other extreme is to put all objects in one group, which corresponds to doing the labeling sequentially.

It would appear that the one object per group approach would be desirable. There are few partial labelings possible for each object and the merging phase will yield only consistent labelings for the objects in the group at each step. However, due to the overhead of launching parallel tasks, it is necessary to do enough work in each task to outweigh this cost. This is the same problem of choosing the proper task *granularity* as encountered in Chapter 3.

To eliminate as many labelings as possible at each step, it is important to group objects together which locally constrain each other. For the purpose of this investigation, the objects were grouped manually using their spatial relationship.

That is, objects which were adjacent in the line drawing were grouped together. This is similar to the way that objects in a real scene interact. Objects in an image constrain other objects which are physically adjacent. Some discussion of this clustering and possible automated approaches to it are discussed in Chapter 5.

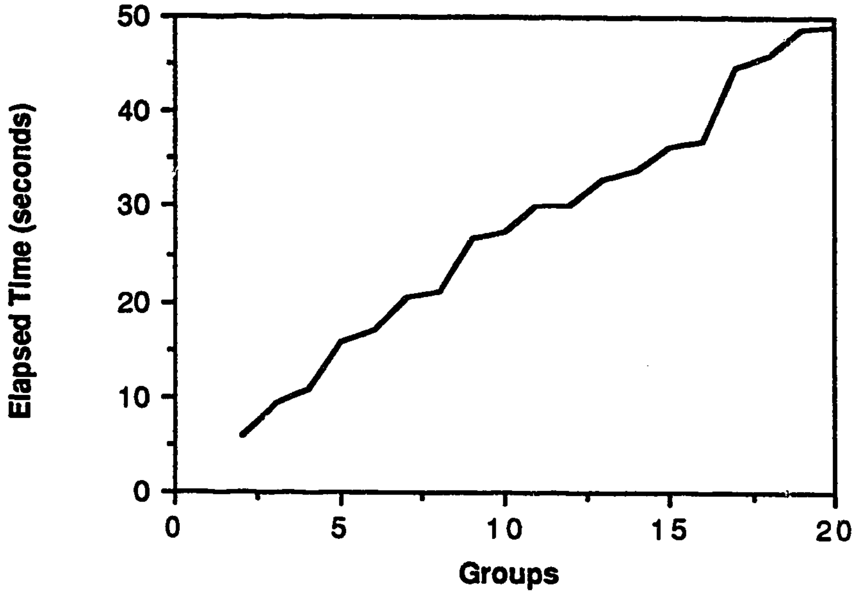
4.1.2 Practical Considerations

The decision of how to split up the set of objects is fundamental to this technique. The memory model used by the Myrias architecture requires that children tasks each write to a different part of its address space if the resultant memory image in the parent is to behave in a predictable fashion. Therefore, enough memory must be allocated by the parent task to provide each child with enough space for its own use.

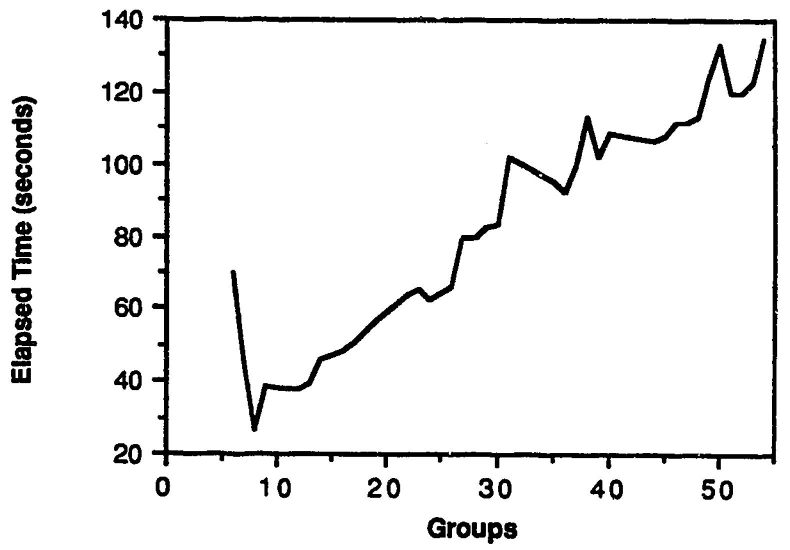
This consideration affects the choice of how many objects to place in each group. Recall that the number of possible partial labelings is exponentially related to the number of objects in a group. If there are too many in each group, the amount of memory which the parent needs to allocate for each child to store the set of partial labelings is more than that which is available. Also, if there are too few objects in each group the parallel task overhead becomes the dominant factor.

4.1.3 Experimental Results

The results presented in Figure 19 represent the elapsed time for the 20 and 54 object test cases described in Appendix A. The experiment was run on a domain size of 16 processors. These results illustrate that this technique is sensitive to the number of objects in each group.



(a) 20 Objects



(b) 54 Objects

Figure 19: Results for Split-Label-Merge technique

Figure 19(a) shows that the 20 object case appears to give the best performance if divided into 2 groups. Further subdivision causes a decrease in performance. The missing data for 1 group is due to the fact that there was not enough memory to provide room for storage of *all* possible labelings of 20 objects ($4^{20} \approx 1.1 \times 10^{12}$). In reality, it is not necessary to provide storage for all labelings because the constraints limit the number possible. However, it is not possible to know in advance how many there will be and so the "worst case" must be accounted for. Reducing the memory requirements of this algorithm, perhaps by estimating the space needed, would be one modification necessary to make this approach practical.

Figure 19(b) shows that for the 54 object case, 8 groups provide the best results. Again, the missing data points for fewer than 6 groups indicates that memory requirements were excessive. Here we also see the trade-off between parallel overhead and granularity. The fact that 6 and 7 groups perform worse than 8 indicates that the task granularity is too coarse—too much work is being done sequentially. For more than 8 groups, the task granularity is too small. The overhead to launch parallel tasks becomes increasingly dominant.

These results indicate that this technique is a possible approach to object labeling. Further research with this technique to overcome practical limitations would be necessary to make it viable and to obtain more information on the predicted behavior over a more comprehensive range of problem and processor domain sizes.

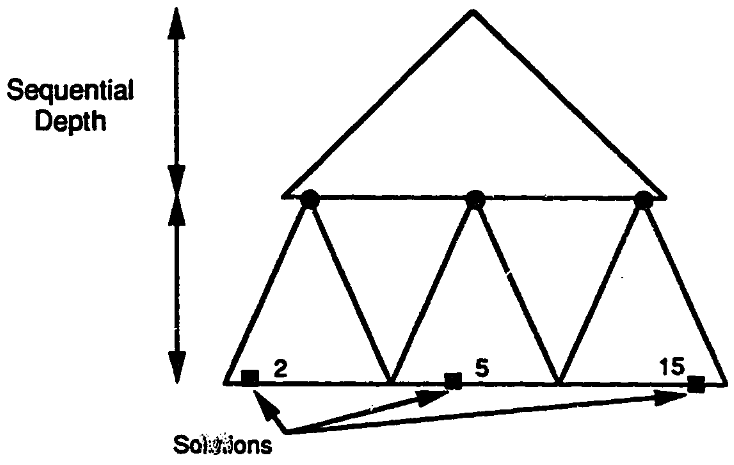


Figure 20: The Eureka jump finding the shortest solution

4.2 The Eureka Jump

In a situation where it is known that only one solution exists or where only one solution is needed, it is desirable to stop the search when that solution is found. This introduces the idea of an *Eureka Jump.* As soon as one solution is found, the program terminates without searching the rest of the tree for other solutions.

The advantage of this technique is that given enough processors, it is possible to guarantee that the fastest solution will be found. To illustrate this, consider Figure 20.

In this example, there are four tasks indicated by the four triangles. The first task launches parallel tasks on three possible labelings after it searches part of the tree to a certain depth using a depth first search. Each of these three tasks will eventually search its part of the tree until it finds a solution, indicated in the figure by the leaf nodes.

A recursive descent algorithm searches the "leftmost" branches of the tree first. In this case, we will assume that the solutions exist at the second leaf node of the first task, the fifth leaf node of the second task, and the fifteenth leaf node of the third task. Using a Eureka jump, the program will terminate when it discovers the solution in the first task at the second leaf node. The time to find a solution is

$$\min(t(2), t(5), t(15)) = t(2)$$

where $t(x)$ is a function relating the position of the leaf node to the time it takes to search to that point.

In a sequential implementation, the time it would take to find a solution depends on the order the three sub-trees are searched. If the first sub-tree is searched first, the time to find a solution is $t(2)$. If the third tree is searched first, the time would be $t(15)$.

For the sake of comparison, assume that the function $t(x)$ is linear. This is a conservative assumption. In fact the function $t(x)$ has an exponential component related to the exponential growth of a subtree. Further, assume that the values can be normalized such that $t(2) = 2, t(5) = 5, t(15) = 15$. On average, the time required to find a solution in the sequential case would be:

$$\frac{t(2) + t(5) + t(15)}{3} = \frac{2 + 5 + 15}{3} = 7.3$$

which is longer than the Eureka jump case, $t(2) = 2$. This shows that the Eureka jump is guaranteed to locate the fastest solution.

The Myrias computation model allows for a Eureka jump. It is expressed simply as a *break* statement within the parallel extension to the C programming language, *pardo*:

```
pardo(i=1:n) {
     ...
     if (solution)
          break;
     }
```

This causes the children tasks of the pardo to terminate and execution to continue after the pardo.

At the time of developing this algorithm, the Eureka jump had not been implemented on the Myrias machine. Therefore, in order to investigate this technique, the Eureka jump was implemented by having child tasks periodically check for the existence of a file. This file was created by the first task to discover a solution. Any other tasks noticing the existence of this file would then terminate.

However, while the behavior is similar to a real Eureka jump, the repeated checks for the existence of the file drastically reduce program performance since it requires repeated calls to I/O instructions.

The advantage to this approach is especially evident in the case where there are more children tasks than there are available processors. In this case, the children tasks must wait for the availability of a free processor. If a solution is found by a task currently executing, then the program can terminate sooner than if all child tasks had to execute.

4.2.1 Experimental Results

The actual implementation of this technique is inefficient. The repeated calls to I/O instructions to attempt to simulate the Eureka jump cause a severe performance penalty. To compensate, an attempt was made to measure and subtract

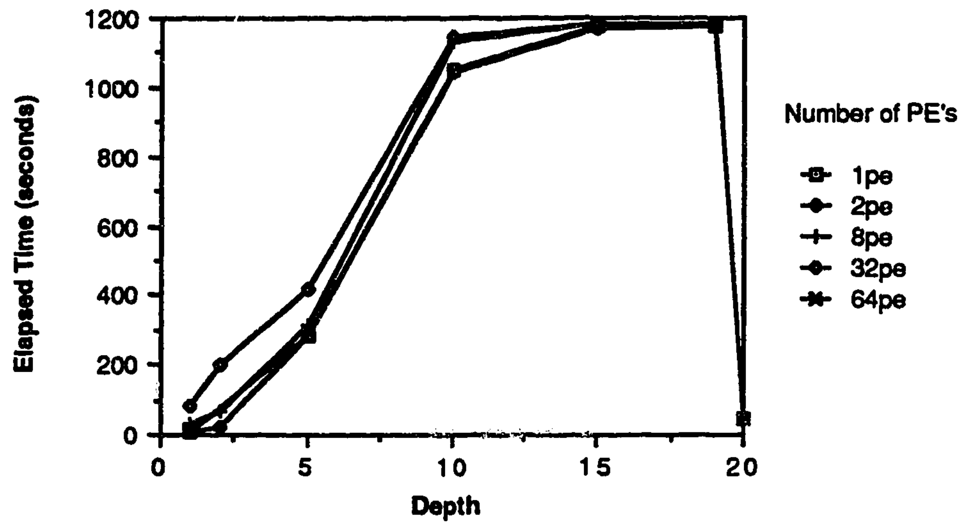


Figure 21: Elapsed time for Eureka jump—20 objects, random ordering

off the time spent in I/O operations. The results presented in Figure 21 have been "adjusted" to remove time spent doing I/O. The experiment was run on the randomly ordered 20 object test case described in Appendix A.

Little can be said regarding the performance of this technique. The shape of the graphs gives some indication of the expected behavior of the Eureka jump. Cautiously, one can observe that the behavior is similar to the normal recursive descent with deepening technique in that the overall shape of the curves is roughly similar. However, speculation beyond that would be inappropriate.

The results serve to indicate that this approach is possible. However, simulation of this technique is disappointing. The usefulness of the Eureka jump is in the quick termination of the algorithm if the solution is found. If the overhead introduced by the Eureka jump is large, then this technique is not viable.

Further investigation on a later version of the Myrias architecture after a real

Eureka jump is implemented would be in order. This approach has the potential to improve the performance of the consistent labeling algorithm if it behaves as promised when it is delivered.

The effectiveness of the Eureka jump is a function of granularity. A coarser granularity implies that there will be greater savings. A fine granularity means that the Eureka jump will provide less savings.

The overhead introduced by the use of the Eureka jump could be made negligible if interrupts are used. This is not possible on the SPS-2, however. Theoretically, Eureka jump should provide an advantage and promises better speedup.

Chapter 5

Conclusion

5.1 Evaluation of the Algorithms

Three MIMD techniques for the solution of the consistent labeling problem have been implemented and tested. The recursive descent with deepening algorithm is fairly easy to program and is suitable for implementation on the Myrias SPS-2 MIMD architecture. Results indicate that this algorithm benefits from parallelism. For larger problem sizes, good speedup is possible indicating that the parallel resources are being used well.

The effect of object ordering to reduce the size of the search tree was also investigated for this technique. Random ordering is the worst case and produces the largest search tree. However, the random case also benefits the most from parallel implementation in that it can use the resources the most effectively. Ordering the objects based on the Unit Constraint relation provides significant improvement over the random case and should be possible in most cases. The best ordering is produced by considering application dependent criteria for the specific problem domain under consideration. If this information is available, then a practical implementation should utilize it for better performance.

Another result is that while "better" sequential search algorithms benefit less from parallelism, if the problem size is increased, the performance speedup improves. For some problems it is sometimes desirable to use a "dumb" algorithm and many processors to obtain better speedup. However, in the case of consistent labeling problems it is indicated that the parallel approach does benefit from smarter sequential techniques.

The split-label-merge algorithm and the Eureka jump technique provide interesting alternative approaches, but do not lend themselves to efficient implementation on the Myrias machine. The split-label-merge algorithm has harsh memory requirements under the Myrias memory model and the Eureka jump technique must be implemented at an operating system level using interrupts for efficient performance.

The split-level-merge approach has potential applications in object labeling problems where the objects constrain each other locally in "clusters". An example would be scene analysis, where objects in an image are clustered together physically in the scene.

The Eureka jump approach has potential for speeding up branch and bound tree search labeling algorithms, such as recursive descent with deepening. It is in the class of branch and bound tree search problems that the apparently anomalous behavior of *super-linear* speedup is possible [13]. This occurs because a solution may be found more than k times faster in parallel than sequentially (k = number of processors). The behavior is not really anomalous in that the speedup is not calculated relative to an "optimal" sequential algorithm but instead relative to the *same* algorithm on one processor. In practice, however, the optimal sequential solution is not known in advance and so *effective* super-linear speedup is possible when calculated this way.

5.2 Contribution of this Thesis

This thesis builds largely on the work of Ullmann *et al.* [29]. They examine theoretically the implementation of the consistent labeling problem on parallel architectures. Their proposal is a fairly simple method of partitioning the consistent labeling problem into sub-problems to be solved in parallel on a number of interconnected processing elements.

The solution offered by them to overcome the inefficient utilization of parallel resources is to have idle processors request additional work from processors with too much. This is similar to the dynamic load leveling provided by the Myrias architecture and is provided transparently to the programmer.

This thesis makes the contribution of examining the effect of object ordering on the efficiency of the labeling algorithm. A method is developed which orders objects based on the relationship of objects as indicated in the unit constraint relation, which is provided as part of the problem specification. This ordering method performs significantly better than a random ordering and so is of benefit in instances where application specific information is not available.

In addition, the deepening modification to the recursive descent technique allows some control over task granularity which can be tuned for a particular architecture in a practical implementation. Experimental results were gathered which illustrate the behavior of the recursive descent with deepening algorithm and demonstrate that significant speedup is possible.

Two alternative approaches were proposed and implemented to demonstrate their feasibility. Both alternative techniques have potential in some cases or on other MIMD architectures.

5.3 Future Work

The split-label-merge algorithm and the Eureka jump technique were not fully investigated due to practical limitations. The first area for future work would be to investigate methods of reducing the memory requirements for the split-label-merge algorithm. Second, after the Eureka jump has been implemented, more comprehensive experiments could be run to investigate its potential for object labeling.

The split-label-merge algorithm required the manual specification of how to group objects. Further research would be useful to develop some sort of clustering technique which would group objects together which are highly constrained. This would involve the design of a distance metric which would characterize the constraints between objects. Objects could then be grouped using standard clustering techniques from the field of pattern recognition.

Some thought has been made to the development of this distance metric. One idea was to represent the problem as a connected graph with the nodes representing the objects and the edges representing the constraints between objects. The length of the shortest path between two objects could then be used as the distance measure.

Another factor which could be examined is the choice of sequential search depth in the recursive descent with deepening technique. This experiment used a fixed sequential depth. That is, the depth the tree was searched was constant at each node in the tree where parallel tasks were launched. It may be desirable to use different values of depth, depending on the current location in the tree. For instance, using the ordering schemes presented by this thesis, the assignment of labels early in the search tree will have a large effect in terms of reducing the number of consistent labelings possible further down in the tree. Perhaps then, it is desirable to use a small value of depth initially, while eliminating inconsistent paths and use a larger value farther down in the tree where fewer paths remain.

One possibility is to have the depth calculated based on the current position in the tree. For example

$$d(u) = a_1 u + a_2$$

is a simple polynomial which is monotonically increasing if $a_1, a_2 > 0$ for $u = 0, 1, \ldots, n$ where a_i are arbitrary constants which depend on the particular application and u is the current position in the search tree.

Another area for future work is in the development of better ordering heuristics for the consistent labeling problem for sequential algorithms. One constraint imposed by the parallel programming model on the Myrias SPS-2 is that tasks must be independent and cannot communicate. However, in a sequential algorithm, it may be possible to use information obtained in searching part of the tree to help order the search in later parts. Schaeffer [25] discusses a technique called the

"History Heuristic" which may have applicability to the labeling problem.

The Eureka jump technique may be influenced by the order in which *labels* are assigned to objects. This thesis has only looked at object ordering, but further work could be done to investigate how to order the labels.

Bibliography

[1] Dana H. Ballard and Christopher M. Brown. Scene labeling and constraint relaxation. In *Computer Vision*, section 12.4, pages 408–430. Prentice-Hall, Inc., New Jersey, 1982.

[2] M. Beltrametti, K. Bobey, R. Manson, M. Walker, and D. Wilson. PAMS/SPS-2 system overview. In *Proceedings Supercomputing Symposium*, pages 63–71, Toronto, 1989.

[3] Ralph Duncan. A survey of parallel computer architectures. *Computer*, 23(2):5–16, February 1990.

[4] R. E. Fikes. REF-ARF: A system for solving problems stated as procedures. *Artificial Intelligence*, 1:27–120, 1970.

[5] Jun Gu, Wei Wang, and Thomas C. Henderson. A parallel architecture for discrete relaxation algorithm. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-9(6):816–831, November 1987.

[6] R. M. Haralick, L. Davis, A. Rosenfeld, and D. Milgram. Reduction operations for constraint satisfaction. *Information Science*, 14:199–219, 1978.

[7] R. M. Haralick and J. Kartus. Arrangements, homomorphisms, and discrete relaxation. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-8:600–612, August 1978.

[8] Robert M. Haralick and Linda G. Shapiro. The consistent labeling problem: Part I. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-1(2):173–184, April 1979.

[9] Robert M. Haralick and Linda G. Shapiro. The consistent labeling problem: Part II. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-2(3):193–203, May 1980.

[10] Thomas C. Henderson. A note on discrete relaxation. *Computer Vision, Graphics, and Image Processing*, 28:384–388, 1984.

[11] Tom Henderson and Ashok Samal. Multiconstraint shape analysis. *Image and Vision Computing*, 4(2):84–96, May 1986.

[12] Robert A. Hummel and Steven W. Zucker. On the foundations of relaxation labeling processes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-5(3):267–287, May 1983.

[13] Ten hwang Lai and Sartaj Sahni. Anomalies in parallel branch-and-bound algorithms. *Communications of the ACM*, 27(6):594–602, June 1984.

[14] Masaru Kamada, Kazuo Toraichi, Ryoichi Mori, Kazuhiko Yamamoto, and Hiromitsu Yamada. A parallel architecture for relaxation operations. *Pattern Recognition*, 21(2):175–181, 1988.

[15] Lefteris M. Kirousis. Effectively labeling planar projections of polyhedra. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-12(2):123–130, February 1990.

[16] J. Kittler and J. Illingworth. Relaxation labelling algorithms — a review. *Image and Vision Computing*, 3(4):206–216, November 1985.

[17] Eva Leung and X. Li. Generalized parallel algorithms for relaxation labeling. In *Proceedings International Symposium on Computer Architecture and Digital Signal Processing*, pages 320–325, Hong Kong, October 1989.

[18] Eva Kwan-sheung Leung. Parallel algorithms for relaxation labelings. Master's thesis, University of Alberta, Fall 1989.

[19] Jeanette Tyler McCall, Joseph G. Tront, F. Gail Gray, Robert M. Haralick, and William M. McCormack. Parallel computer architectures and problem solving strategies for the consistent labeling problem. *IEEE Transactions on Computers*, C-34(11):973–980, November 1985.

[20] Fernando A. Mota and Flavio Roberto D. Velasco. A method for the analysis of ambiguous segmentations of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-8(6):755–760, November 1986.

[21] Shmuel Peleg. A new probabilistic relaxation scheme. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-2(4):362–369, July 1980.

[22] Keith E. Price. Relaxation matching techniques - a comparison. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-7(5):617–623, September 1985.

[23] Elaine Rich. *Artificial Intelligence*, pages 351–358. McGraw Hill, 1983.

[24] Azriel Rosenfeld, Robert A. Hummel, and Steven W. Zucker. Scene labeling by relaxation operations. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-6(6):420–434, June 1976.

[25] Jonathan Schaeffer. The history heuristic and alpha-beta search enhancements in practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-11(11):1203–1212, November 1989.

[26] Howard Jay Siegel, Leah J. Siegel, Frederick C. Kemmerer, Philip T. Mueller, Jr., Harold E. Smalley, Jr., and S. Diane Smith. PASM: A partitionable SIMD/MIMD system for image processing and pattern recognition. *IEEE Transactions on Computers*, C-30(12):934–946, December 1981.

[27] Demetri Terzopoulos. Image analysis using multigrid relaxation methods. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-8(2):129–139, March 1986.

[28] S. Ullman. Relaxation and constrained optimization by local processes. *Computer Graphics and Image Processing*, 10:115–125, 1979.

[29] Julian R. Ullmann, Robert M. Haralick, and Linda G. Shapiro. Computer architecture for solving consistent labelling problems. *The Computer Journal*, 28(2):105–111, 1985.

[30] Benjamin W. Wah, Guo jie Li, and Chee Fen Yu. Multiprocessing of combinatorial search problems. *Computer*, 18(6):93–108, June 1985.

[31] Andrew M. Wallace. A comparison of approaches to high-level image interpretation. *Pattern Recognition*, 21(3):241–259, 1988.

[32] David Waltz. Understanding line drawings of scenes with shadows. In P. Winston, editor, *The Psychology of Computer Vision*, chapter 2, pages 19–91. McGraw Hill, New York, 1975.

[33] S. Yalamanchili and J. K. Aggarwal. A system organization for parallel image processing. *Pattern Recognition*, 18(1):17–29, 1985.

Appendix A

Line Drawings Used in the Experiments

There were four test drawings designed for the experiments which were run on the Myrias SPS-2 parallel computer. Figures 22, 23, 24, and 25 illustrate these cases. The first three (9, 20, and 33 objects) are increasingly complex combinations of simple cubes. The fourth (54 objects) is an attempt to demonstrate what a "real" scene would look like after pre-processing to extract the line segments.

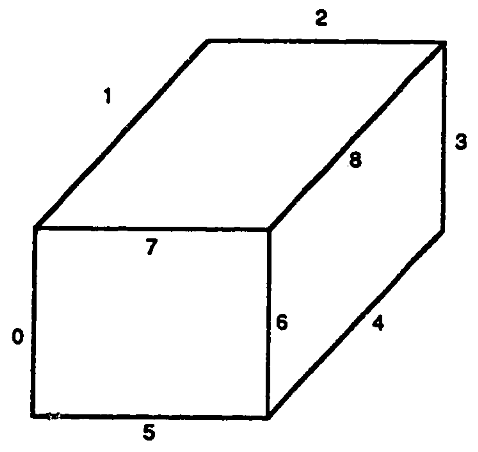


Figure 22: 9 Object Line Drawing

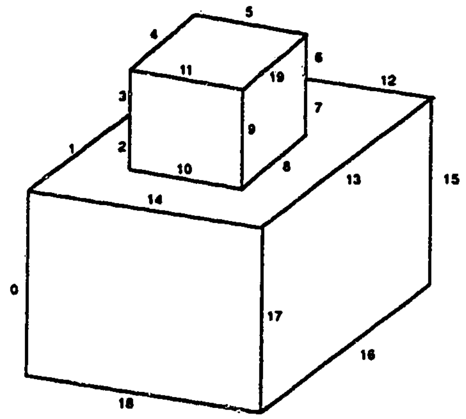


Figure 23: 20 Object Line Drawing

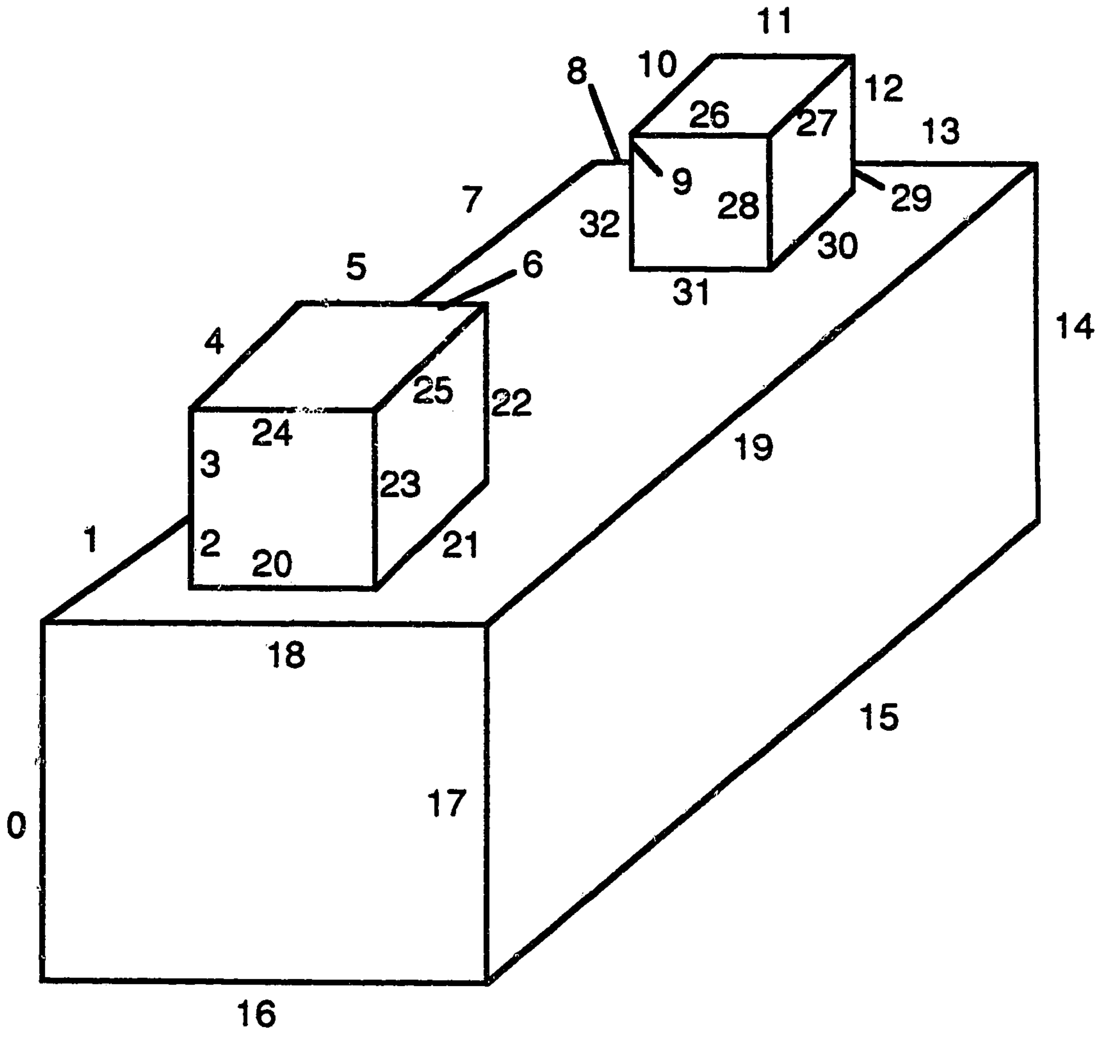


Figure 24: 33 Object Line Drawing

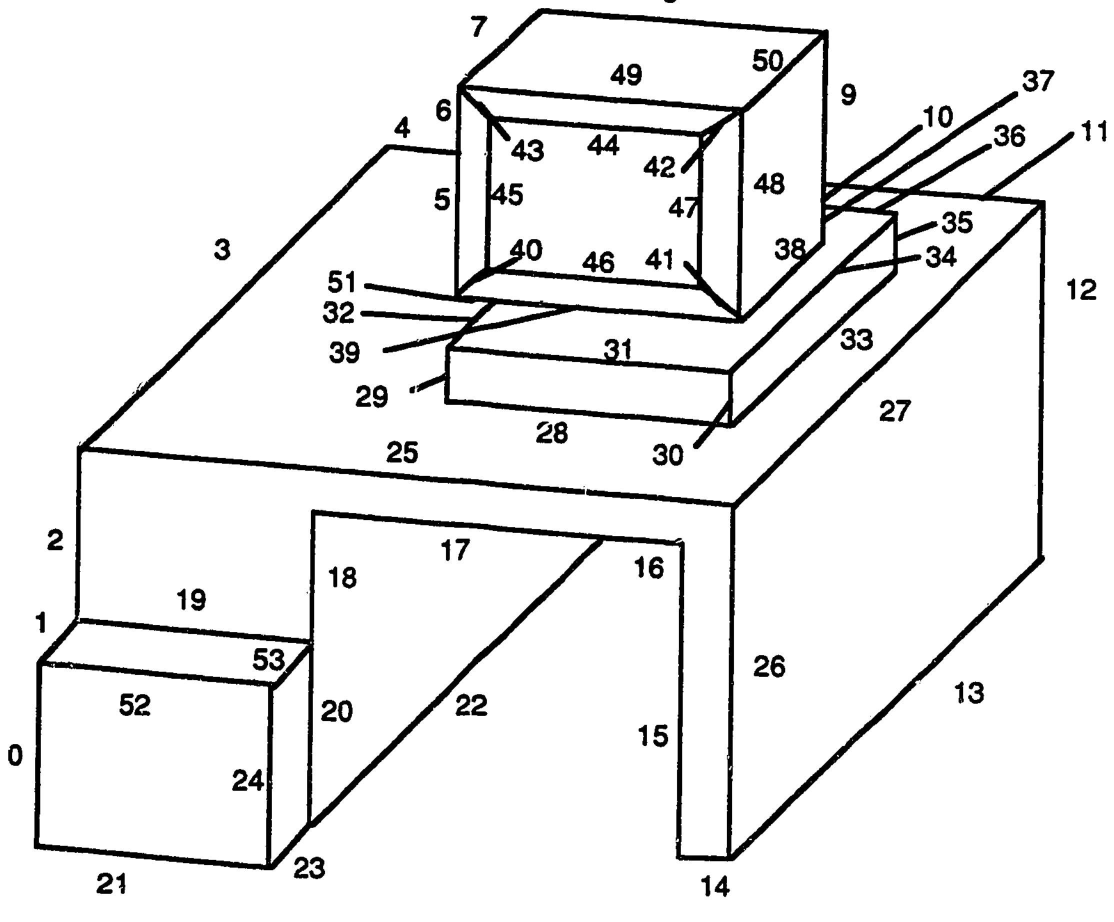


Figure 25: 54 Object Line Drawing

Appendix B

Format of the Input Files for the Line Drawings

The format of the input file used by the programs in this thesis is:

- D: followed by the iterative depth. In the case of the merge algorithm, this number indicates the number of sets to partition the objects into. For this experiment, the text string *DEPTH* was substituted with the desired value of iterative depth using the Unix utility "sed".

- U: followed by the number of units (objects) in the problem.

- L: followed by the number of labels in the problem and a number of text strings indicating the names of the labels. The line drawings used in this thesis had a label set consisting of four possible labels: convex (+), concave (-), occluding right (→), occluding left (←).

- T: followed by a number of digits separated by commas. This indicates the unit-unit constraint relation T. Each line indicates a number of units which constrain each other.

- R: followed by a number of digits separated by commas. This is the unit-label constraint relation R. Each line indicates a number of unit-label pairs which are permissible for the objects constrained in set T. The format is:

  unit,label,unit,label,. . . .

  This unit-label constraint set was determined by considering the type of intersection at which the line segments meet. The intersection types are: L, Fork, Arrow, and T.

Appendix C

An Overview of Parallel Computer Architectures

C.1 Introduction

There are a wide variety of parallel processing technologies. Recent developments have seen advances in parallel hardware architectures, interconnection technologies, and programming paradigms. This appendix attempts to summarize the types of parallel architectures which exist and their relationship to one another. Duncan [3] provides a good survey of parallel computer architectures from which the following is abstracted.

C.2 Terminology

The following discussion will exclude architectures which contain only *low level parallelism.* This includes such features as instruction pipelining, multiple CPU

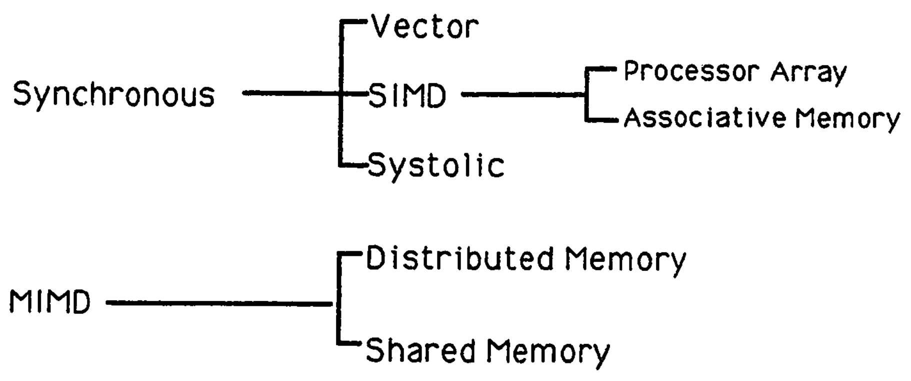


Figure 26: Taxonomy of Parallel Architectures

functional units, and separate I/O and CPU processors. The reason these are not considered is that, although these contribute to overall performance improvement, their presence does not make a computer a parallel architecture.

Duncan [3] offers the following definition:

> ... a *parallel architecture* provides an explicit, high-level framework for the development of parallel programming solutions by providing multiple processors, whether simple or complex, that cooperate to solve problems through concurrent execution.

Figure 26 illustrates an informal taxonomy based on this proposed definition. This serves to demonstrate the principal approaches to parallel computer architectures.

C.3 Synchronous Architectures

The key feature of synchronous parallel architectures is the concurrent operation of instructions which are coordinated globally by a central control unit under the direction of a global clock.

C.3.1 Pipelined vector processors

Vector processors are distinguished by the presence of multiple, pipelined functional units. These implement arithmetic and Boolean operations on both vectors and scalars and can operate concurrently. This architecture provides parallel processing by sequentially streaming vector elements through a functional unit pipeline and chaining the output of one pipeline to the input of another.

After an initial delay to fill the pipeline, the architecture can provide a result every cycle. Due to this initial delay, performance of pipelined architectures is sensitive to the startup overhead of the algorithm.

C.3.2 SIMD Architectures

SIMD architectures generally consist of a central control unit, multiple processors, and an interconnection network for communication between processors or between processors and memory. The control unit broadcasts a single instruction to all processors. Every processor then executes the instruction in a "lockstep" fashion on data local to the processor. The interconnection network allows processors to communicate results to each other for use in subsequent calculations.

Individual processors may be able to ignore the current instruction from the central control unit. SIMD algorithms consist of the set of instructions to be issued by the central control unit and a mask indicating which processors will participate.

C.3.3 Systolic Architectures

Systolic architectures are generally proposed to solve special-purpose problems which must balance intensive computations with severe I/O bandwidth demands. Systolic arrays are pipelined multiprocessors in which data is pulsed rythmically through a network of processors before returning to memory. This pipelined data flow is synchronized by explicit timing delays and a global clock. During each time interval, the processing elements execute a short sequence of instructions.

This class of architecture can address the requirements of special purpose systems by providing significant parallel computation and decreasing memory and I/O bottlenecks. Systolic architectures maximize the computations performed on a datum by passing it to whichever processors need it without having to store it to memory. Only the processors on the edge of the array need to perform I/O.

C.4 MIMD Architectures

The processors in a MIMD architecture can execute independent instruction streams using local data. Therefore problems which require processors to behave in an autonomous manner are well suited to MIMD machines. Software processes executing on MIMD architectures synchronize by either passing messages through an interconnection network or by accessing shared memory. This class of parallel architecture is asynchronous, demonstrating decentralized hardware control.

The driving force behind the development of MIMD architectures is that higher level parallelism can be exploited (subprogram and task level). One type of algorithm which is well suited to this technique is known as "divide and conquer", where a problem is broken down into smaller subproblems which are largely independent.

C.4.1 Distributed Memory Architectures

Distributed memory architectures connect processing nodes (an autonomous processor and local memory) with an interconnection network. Data is shared explicitly by passing messages between processors since there is no shared memory. The benefit [illegible] architecture is the ability to "scale" a system. It is well suit[illegible] applications which make use of local data references.

[illegible]mory Architectures

S[illegible]ectures provide a global, shared memory that each processor can [illegible]. This type of architecture does not have some of the problems associated with message passing architectures such as latency caused by queueing and forwarding messages.

Other problems must be resolved. Data access requires atomic synchronizing mechanisms to prevent a processor reading a memory location before another has finished updating it. Each processor in this type of system also typically has a memory used as a cache. Multiple copies of the same data may exist in several processors' caches at the same time. Ensuring that a consistent version of the data resides at each cache is known as the cache coherency problem.

C.5 MIMD/SIMD Architectures

There are also a variety of experimental hybrid architectures which allow selected pieces of a MIMD architecture to behave in a SIMD fashion. The main benefit of such hybrid systems is their flexibility. The details of such systems are quite diverse and include such things as dataflow architectures, reduction architectures, and wavefront array architectures.

Appendix D

Myrias PAMS/SPS-2 System Overview

The Myrias multi-layered architecture allows code to function across new developments in technology without change. All user software is targeted for a virtual machine computer called the "G" machine.

Each layer has an associated language. On the top layer are user applications written in Myrias Parallel Fortran or Myrias Parallel C. Next, we find the compilers, standard libraries, and associated tools. The output of the compilers is the G instruction set. Below this layer is the operating system layer. Myrias supports a standard UNIX environment. Below this is the *G* line. System software below this point is transparent to the user. The G translator accepts G object code and translates it into the native machine code of the underlying hardware.

The system control mechanism implements the *pardo* construct. Tasks are mapped onto physical processors and the address spaces of tasks are mapped onto real memory. This control mechanism operates without user intervention. The hardware layer consists of the processing elements including memory and a communication subsystem.

To incorporate a new underlying hardware implementation, only the G translator and the interface control mechanism to the communication subsystem have to be re-implemented.

D.1 Programming on the Myrias System

User programs are written for the Myrias system using either FORTRAN or C. A language extension, *pardo*, is the only modification to standard FORTRAN or C. The *pardo* (parallel do) construct has the same syntax as the *do* loop in FORTRAN. In loops where the iterations are independent, their execution can be made parallel by replacing the *do* statement by a *pardo*. Each "iteration" of a *pardo* is a distinct task with its own memory image. Each task references only its own address space, so no task synchronization is required.

Parallel tasks are created when the *pardo* instruction is executed. The parent which executed the *pardo* is suspended while the child tasks execute in parallel in any order. Each child inherits an identical copy of the parent's address space, with only the iteration variable varying. The child tasks' address spaces are merged to form a new memory image on completion. Merging rules specify that if exactly one child modifies a memory location or if several children modify the same location in exactly the same way, that is the new value in the parent's address space. If two children modify the same location differently, the new value is undefined and its contents are unpredicable.

The system control software handles the assignment of tasks to processing elements, dynamic load balancing by migration of tasks among processing elements, and all copying, storage, and merging of memory images.

D.2 Myrias Supervisor

The Myrias Supervisor provides access to the system and provides a UNIX environment for user programs. The supervisor software allows programs to be run from any machine networked to the system. The *mrun* command is used to run a program on the system. *Mrun* allocates the processing elements specified by the user, loads the program and transfers control to the Control Mechanism.

I/O is done through the master controller. Sustained I/O rates are 500 KB/sec. Myrias promises that future implementations will use Input/Output Processors (IOP's) directly attached to the system. Graphics, networking, and tape systems will directly interface with the IOP's.

D.3 The Control Mechanism

The Control Mechanism manages the execution of user programs. There are three main functions of the Control mechanism:

1. Task Control. The Control Mechanism suspends the parent process when a *pardo* is executed and distributes the newly formed child tasks to processors for concurrent execution. On completion, the parent task is resumed.

2. Dynamic Load Balancing. The Control Mechanism ensures a balanced work load across all processing elements. Tasks are moved from processors with excess work to processors which are idle.

3. Memory Management. The Myrias system is a virtual memory system with demand paging among processing elements. A hierarchical caching scheme is used to minimize the overhead of managing task address spaces.

D.4 Hardware

The Myrias SPS-2 hardware is built up of a number of modular building blocks. At the highest level, a SPS-2 system is composed of a number of interconnected processor frames called cages and a Master Controller (MC). The computation is carried out in the cages and I/O is carried out in both the cages and the MC.

A cage is composed of sixteen multiprocessor card assemblies and a communications card assembly. In addition each cage contains a backplane, a clock card assembly, power supplies, cooling fans, and cabinets.

The processors and communications card in a cage communicate with each other via a portion of the backplane called the Inter Family Link. This provides two completely separate communication channels which may operate simultaneously.

Each processor board consists of four processing elements, a bus interconnecting the processors and a link to the Inter Family Link. The execution of user programs is carried out entirely by the processor elements. Each element consists of a Motorola MC68020 microprocessor with a MC68882 floating point coprocessor, a MC68851 paged memory management unit with four megabytes of dynamic random access memory (DRAM), a controller, additional memory for the detection and correction of bit errors, 8192 bytes of ROM, and control circuitry. The communications subsystem is composed of a proprietary application specific integrated circuit (DMA ASIC) that handles communication between processing elements.

The communication card assembly is responsible for providing communications with other cages and overseeing the use of the Inter Family Link within its own cage. Also, system initialization and environmental monitoring are provided.

D.5 Intercommunication

Communication between components of a Myrias system takes the form of double-ended messages. Each message is sent from a sender's memory space to a target's memory space. A message is essentially an address followed by a variable length stream of bytes representing the contents of the message. Any component in the system can communicate with any other.

A target component has complete control over where the message goes. It can reject a message if it decides it does not want it. This protocol is implemented in hardware and ensures that the communications system is reliable. If a message cannot be correctly transmitted, the sender of the message is advised. The sender is told why the transmission failed and the target is forced to abort it on receipt.

Refer to Beltrametti *et al.* for a more complete description of the PAMS/SPS-2 System.