

PROCESS ALLOCATION AND LOAD BALANCING IN PARALLEL LOGIC
PROGRAMMING

by

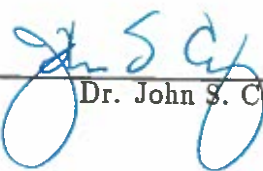
MOATAZ ALI MOHAMED

A THESIS

Presented to the Department of Computer and Information Science
and the Graduate School of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Masters of Science

August 1990

APPROVED:

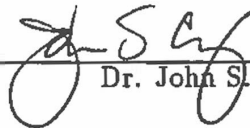


Dr. John S. Conery

An Abstract of the Thesis of

Moataz Ali Mohamed for the degree of Masters of Science
in the Department of Computer and Information Science to be taken August 1990
Title: PROCESS ALLOCATION AND LOAD BALANCING IN PARALLEL LOGIC
PROGRAMMING

Approved:



Dr. John S. Conery

We study the problem of process allocation in the course of executing parallel logic programs. Very little work has been done in addressing the process allocation problem in the logic programming paradigm. We observe that the processes that arise in the course of executing AND/OR model programs have very regular and repetitive communication patterns, which justifies the design of tailored dynamic allocation algorithms that utilize such characteristics to achieve better performance. The contributions of this research are: first, addressing the dynamic process allocation problem as a graph embedding problem, where two recently developed graph embedding algorithms are adapted to be used in OM (a virtual machine based on the AND/OR execution model). The performance of the algorithms is evaluated using simulations. Second, developing distributed heuristics for process allocation in OM, one of which is the first algorithm to appear in the literature that combines both sender initiated and receiver initiated strategies. Third, describing a queuing model for OM and analyzing the key parameters of the model and their effect on performance.

VITA

NAME OF THE AUTHOR: Moataz Ali Mohamed

PLACE OF BIRTH: Alexandria, Egypt

DATE OF BIRTH: July 26, 1966

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon
Alexandria University, Alexandria, Egypt

DEGREES AWARDED:

Master of Science, 1990, University of Oregon
Bachelor of Science, 1988, Alexandria University

AREAS OF SPECIAL INTEREST:

Parallel Logic Programming
Parallel Processing and Parallel Architectures
Load Balancing and Scheduling
Programming Languages and Compilers

PROFESSIONAL EXPERIENCE:

Associate System Administrator, Department of Computer and Information
Science, University of Oregon, Eugene, June 1990–August 1990

Graduate Research Assistant, Department of Computer and Information Science,
University of Oregon, Eugene, December 1988–June 1990

Graduate Teaching Fellow, Department of Computer and Information Science,
University of Oregon, Eugene, Fall 1988; Fall 1989; Spring 1990

Visiting Research Programmer, AI Lab, Department of Computer Science,
University of Utah, Salt Lake City, Summer 1987

AWARDS AND HONORS:

Alexandria University, School of Engineering, Dean's honors list, 1986-1988.

PUBLICATIONS:

Lo, V. M., Rajopadhye, Mohamed, M. A., Nitzberg, N., Gupta, S., Keldsen, D.,
Telle, J. LaRCS: A language for describing parallel computations.
In *Proceedings of the 1990 Hawaii International Conference on Systems Sciences*,
accepted for publication.

Lo, V. M., Rajopadhye, Gupta, S., Keldsen, D., Mohamed, M. A., Telle, J.
OREGAMI: Software tools for mapping parallel algorithms to parallel architectures.
In *Proceedings of the 1990 International Conference on Parallel Processing*,
(August 1990), pp. to appear.

Lo, V. M., Rajopadhye, Gupta, S., Keldsen, D., Mohamed, M. A., Telle, J.
Mapping Divide and Conquer Algorithms to Parallel Architectures
In *Proceedings of the 1990 International Conference on Parallel Processing*,
(August 1990), pp. to appear.

ACKNOWLEDGEMENTS

I wish to express my gratitude to my adviser, John Conery for his guidance and support throughout this research and for his valuable comments on earlier drafts of this thesis. I also wish to thank Virginia Lo for supplying many references to the classical load balancing literature, and for choosing me to work with her in a related project, OREGAMI. I learned a lot about parallel processing, especially process allocation and the mapping problem through my work in this project. Thanks are also in accord to Evan Tick for the discussions on the Japanese research projects, Michael Stafford for the many discussions about the details of OM's implementation, Michael Quinn for comments in the early stages on developing load balancing heuristics, and Amanda Ronai for helping in drawing some of the figures.

DEDICATION

To my parents and two sisters.

TABLE OF CONTENTS

Chapter	Page
I. INTRODUCTION	1
A Tour in the Parallelism Galaxy	1
The Parallel Programming Dilemma	3
Why Parallel Logic Programming?	5
Goals	5
Thesis Overview	6
II. CLASSICAL LOAD BALANCING	8
Introduction	8
Definitions and Preliminaries	8
Taxonomy for Load Balancing Algorithms	12
Load Balancing Algorithms	14
Conclusions	16
III. PROCESS ALLOCATION AS A GRAPH EMBEDDING PROBLEM	17
Dynamic Graph Embeddings	17
Algorithms for Mapping Dynamic Trees onto Hypercubes	17
User Guided Process Allocation	20
Conclusions	21
IV. LOAD BALANCING IN PARALLEL LOGIC PROGRAMMING	23
Introduction	23
The Processing Power Plane	23
Multi-level Load Balancing	28
Load Dispatching Strategies for the Parallel Inference Machine	32
Contracting Within Neighborhood	36
V. OVERVIEW OF THE OPAL MACHINE	40
Introduction	40
Parallel Logic Programming Models	40
The AND/OR Process Model	41

	Communication Characteristics of the AND/OR Model	43
	The OPAL Programming Environment	43
	The OPAL Machine	44
	The Process Allocator	45
VI.	PROCESS ALLOCATION ALGORITHMS FOR OM	47
	Generalized Random Walks	49
	The Crawl Algorithm	50
	ANDs Crawl, ORs Run	51
	Walk With Your Neighbors	51
	The Rendezvous Algorithm	51
	The AND-Shipping, OR-Stealing Algorithm	53
	Conclusions	54
VII.	SIMULATIONS AND PERFORMANCE EVALUATION	55
	The Simulation Environment	55
	Reliability of the Performance Results	55
	The Test Suite	57
VIII.	A QUEUING MODEL FOR OM	68
	Bounding the Number of Processes	72
IX.	CONCLUSIONS AND FUTURE RESEARCH	74
	BIBLIOGRAPHY	76

LIST OF TABLES

Table	Page
1. Calculation of Min-hops and Max-hops in ACWN	37

LIST OF FIGURES

Figure	Page
1. An Example of Contention in a Mapping	20
2. Subtask Distribution in the Multi-level Algorithm	30
3. An Example of a Non-optimal Allocation in CWN	39
4. The Software Layers of OM	45
5. The Interface Between the Process Allocator and the OS Components. . . .	46
6. The Map	57
7. The Map Coloring Program in OPAL	58
8. The Even Path Program in OPAL	60
9. The Speedup in the Map Coloring Program	61
10. The Average Queue Length in the Map Coloring Program	62
11. The Average Dilation in the Map Coloring Program	64
12. The Speedup in the Path Program	65
13. The Average Load in the Path Program	66
14. The Average Dilation in the Path Program	67
15. A Queuing View of Process Allocation in OM	69
16. A Plot of the Average Queue Length Function	71

CHAPTER I

INTRODUCTION

Parallel processing is a rapidly developing field that is attracting the interest of researchers from various disciplines. Speed, being the obsession of every programmer or researcher who has a large scale or computationally intensive problem, is one of the primary motivations. However, as the current developments in the field stand, running a program on a machine with more processors does not necessarily imply better performance.

The parallel processing field is currently partitioned into several competing programming paradigms. In the following section we will attempt to provide a brief overview of these paradigms and point out some of the primary concerns in programming the representative machines of each of those paradigms. We also name some of the applications that seem well suited to those machines, as well as those that programmers have found difficult to map to the architecture.

A Tour in the Parallelism Galaxy

- Shared memory multiprocessors.

A shared memory multiprocessor is composed of several main stream microprocessors usually connected by a bus and sharing a global memory. Typical examples of this class are the Sequent Symmetry, Alliant Fx/80, and the Encore Multimax. The main problems in programming such machines are locking and synchronization over shared data structures and cache coherency. Lack of scalability is also a problem since the architecture is constrained by the physical limitation on the number of processors that can be connected on a shared bus. Several parallel logic programming systems, such as Aurora [27] Andorra [14] have been successfully implemented on shared memory machines.

- Message passing multicomputers.

Multicomputers are composed of autonomous loosely coupled processors connected by an interconnection network [1]. The processors are usually full fledged main stream microprocessors with each processor having its own memory. No global shared memory is supported. The main advantage of these systems is their scalability. The primary disadvantage is the high cost of communication over the network. One of the main problems in programming these architectures is mapping the application programs onto the architectures. Several research projects are currently addressing this problem in the context of developing integrated environments of compilers and mappers that would relieve the programmer from hard-coding the mapping into her/his code. Examples of this class are the hypercube architectures, such as Intel's iPSC/2 and NCUBE; mesh architectures such as ICOT's Multi-PSI; the BBN Butterfly, the twisted torus HP Mayfly; and the Pyramid machine.

- Vector processors.

Vector processors were the first commercially available machines to target the data parallelism application domain. The main idea was to partition the data into vectors so high speed vector operations could be performed on them. Typical examples of this class are the Cray machines, such as the CRAY-1, X-MP, and Y-MP. One of the primary reasons of the success of vector processors is compiler technology, particularly Fortran compilers. The compilers do intelligent optimizations in analyzing the number of iterations in a loop and the data dependences to fit the arrays in vector registers [15].

- SIMD massively parallel array processors.

SIMD (single instruction multiple data) architectures were pioneered by the design of the ILLIAC-IV computer. Recent commercial SIMD machines are Thinking Machines' Connection Machine (CM) [16] and MasPar's MP-1. The main idea in the design of these architectures is to connect a huge number of very primitive processing elements (PE) in an array or hypercube fashion. The architecture targets the

class of applications which are very fine grain with a high degree of regularity, such as image processing, weather simulations, and matrix computations. It is our belief that any problem for which a systolic algorithm can be written could be efficiently programmed on such architectures.

The Parallel Programming Dilemma

From the programming viewpoint, researchers have taken several conflicting approaches towards programming parallel architectures. One current trend in parallel processing research is the development of new programming environments specifically for certain architectural platforms. There are three main directions that seem to be having a degree of success. One is to take popular established languages such as Fortran or C and augment them with extensions for parallelization. Examples of such languages are the iPSC C and CM-Lisp.

Another approach is to devise novel parallel languages that have originally parallel semantics and that do not assume any particular target architecture. An example of this approach is the Strand88 language.

The third approach is to design a portable threads library that can be used as an add-on to various sequential programming environments. Clearly this library is to be portable across various architectural platforms. Linda could be regarded as an example of this direction since Linda is independent of the host programming language and thus can be used as an add-on parallelization extension. The main difference between this approach and the first one is that it is more generic since it is independent of both the target architecture and the programming language. In some sense, this portability is a direct result of factoring out (decoupling) the threads library from the programming language and the architecture-specific details.

The Quest for Efficiency

Every parallel programmer must have experienced at some point the funny feeling of having her/his parallel program run slower than its sequential counterpart. Another

serious problem is having non-monotonic speedup curve, i.e., at certain regions in the curve the time increases while increasing the number of processors. This implies that the program has either sequential bottlenecks or communication-intensive regions that tend to dominate when the program is executed using a number of processors in that range. There is a consensus in the computer science community that making a parallel program more efficient is much harder than optimizing a sequential problem. One of the main factors contributing to this situation is the lack of robust symbolic debuggers, program profilers, trace generators, and similar tools which would help programmers analyze their programs and determine their execution bottlenecks. Many of these tools are currently under construction in academia and in industry.

Architecture Independent Parallel Programming

Architecture independent programming facilitates two properties that are major goals in programming language design: portability, and an abstract level of programming. The programmer should program at a level of abstraction that lets her/him concentrate on the algorithmic issues themselves and not be burdened with machine specific details, such as the number of processors and the interconnection network characteristics.

One approach researchers have taken to provide for architecture-independent programming is to develop automatic mappers that would automate the process allocation problem. Depending on the computation model, the mappers can either operate statically (at compile time) or dynamically (at run time). Strand88, one of the commercial concurrent languages, totally ignores the problem and leaves it to the programmer to do the process allocation.

We believe that the motivations behind architecture independent parallel programming are very similar to those of the sequential declarative programming paradigm in the sense that we want the programmers to concentrate on their algorithm's characteristics, such as, what constitutes a process, how the processes communicate, and what the algorithm does, instead of spending time considering issues like how many processes should be mapped to one processor or deriving a function that maps a process label to a pro-

cessor number. These issues dictate how the algorithm is to be executed, which should not be the programmers' concern. The less the programmer is burdened by control issues the less productive the programming environment is, especially with the ever increasing complexity of control in programming parallel machines.

Why Parallel Logic Programming?

The elegance of logic programming as a programming paradigm is attributed to its declarative nature. Programmers only specify what the algorithm computes without specifying how it should compute it. Thus the implementation is left open to the underlying environment. It is this freedom that opens the door for parallelism. It is now clear that overspecification is often the main obstacle in achieving efficient parallelism. This clean separation of semantics and control is what gives logic programming its power and suitability to parallelism.

Parallel logic programming offers a unique mix of the advantages of the above approaches. It has the same "look and feel" of an old successful language, namely Prolog. The semantics of pure logic programming do not have the sequential restrictions of the conventional Von Neumann languages, and thus lends itself very naturally to parallelism. The last advantage is portability which is a direct consequence of the emphasis on logic rather than control. Clearly, programs that do not explicitly specify control issues will be machine independent which facilitates source-level portability.

Goals

In this research, we study the problem of allocating dynamically spawned processes arising in the course of the execution of parallel logic programs. There has been very little work on this problem in the logic programming literature. However, the problem of process allocation and load balancing has been extensively studied in its own right, independent of a particular computation model. In this study, we survey the recent work that has been done in the context of existing parallel logic programming systems, and introduce new process allocation heuristics that achieve good performance according to

the metrics of speedup, load sharing, and bounded dilation.

The setting of the problem is as follows. The target architecture is assumed to be a message passing architecture with any type of interconnection network. We do not assume the existence of a global address space. We assume that each processor has a minimal kernel resident at that node. The programming language used for coding the programs (OPAL, a Prolog-like language with AND/OR parallelism) does not express any user-specified mapping information and thus the user does not interfere with the mapping of the processes to the processors. Hence, the allocation strategy is totally automatic and transparent to the user.

The main goal of this study is to design efficient process allocation heuristics for message passing parallel logic programming systems. One of the main motivations of this study was to incorporate ideas from three areas: graph embeddings, classical load balancing, and queuing theory. This study is also an exercise in evaluating heuristics for an open problem for which even the performance metrics are subject to debate. We have looked at the problem from two different views: graph embedding and process allocation. The original goal was to implement the heuristics on message passing parallel architectures such as the hypercube, but since the parallel implementation of OM was not yet ready at the time of this study, simulations were chosen to test the ideas presented here.

Thesis Overview

In chapter II we first present a taxonomy of load balancing algorithms, followed by a survey of some of the most relevant classical load balancing algorithms that were developed for either distributed computing or parallel processing systems. Each of the algorithms is evaluated and classified in view of the presented criteria. In chapter IV we survey the load balancing algorithms that were specifically designed to be used in parallel logic programming systems. Graph embedding is presented as a model for dynamic process allocation in chapter III, where two recent algorithms for embedding trees into hypercubes are discussed. Then, the proposal of using configuration languages as a tool to express the programmer's knowledge about the program's characteristics and pass it on to

the allocator to achieve efficient embeddings is examined. An overview of the AND/OR process model and the architecture of the OPAL machine is presented in chapter V together with a discussion of the most prominent communication and execution characteristics of the processes. The process allocation heuristics we developed are presented in chapter VI. In chapter VII, we discuss the simulation environment and the performance evaluation of the algorithms according to the various proposed metrics. Chapter VIII presents a queuing model for OM and discusses some of the important parameters of the model and their impact on performance. Finally, our conclusions and future extensions to this research are presented in chapter IX.

CHAPTER II

CLASSICAL LOAD BALANCING

This chapter gives a survey of classical process allocation and load balancing algorithms, with an emphasis on dynamic algorithms. We present a taxonomy for such algorithms according to some of their common characteristics. The impact of these characteristics on some performance metrics is also discussed. Finally, we address the analogy between the results obtained from the work on load balancing algorithms for distributed systems and dynamic allocation algorithms for multicomputers.

Introduction

The complexity of designing a multiprocessor is rapidly decreasing as the field of parallel architecture is advancing. However, the efficient programming of such sophisticated machines is not getting any easier. The problems that arise in the course of programming a multicomputer are primarily due to the difficulties and the overhead incurred in mapping the computation to the processors of the architecture. This problem can be tackled either statically (i.e., at compile time), or dynamically (i.e., at run time).

Throughout this thesis, we will mainly focus the discussion on homogeneous message passing multicomputers, where all the processors are identical and no global shared memory exists. We will review some of the dynamic load balancing algorithms that addressed the problem in both distributed and parallel processing systems.

Definitions and Preliminaries

In this section, we will give the definitions of the terms used throughout the paper. Some of these definitions are not widely used yet, but we believe they are gaining enough popularity that their standardization is to be expected.

- **Mapping Algorithm**

A mapping algorithm assigns each process in the computation a processor in the architecture. The typical goals of mapping algorithms are to achieve constant load among the processors (load balancing) and to minimize the dilation (defined below).

- **Load Balancing Algorithm**

A load balancing algorithm distributes the computation tasks among the processors, maintaining constant load among all the processors of the system. One might think that a load balancing algorithm is simply a mapping algorithm and there should not be any need for the distinction. It is certainly true that load balancing algorithms are a subclass of mapping algorithms, however the distinction is important since some systems do the process allocation in two stages: first some kind of a random mapping is used, then a load balancing algorithm is invoked to improve the mapping decisions.

- **Load Sharing Algorithm**

A load sharing algorithm is one that distributes the computation tasks among the processors, but is not guaranteed to maintain constant load across all the processors of the system. It is important to note that almost all claimed load balancing algorithms are really load sharing algorithms since guaranteeing uniform load on all the processors in the system is almost always impossible. Therefore, we will use the terms interchangeably throughout this document.

- **Thrashing**

Thrashing is the situation where all the processors are busy transferring tasks and no actual computations are executed. This phenomenon is often encountered when the mapping algorithm does not incorporate a control policy for terminating the transfer of a task.

- **The Degree of an Interconnection Network**

The degree of an interconnection network is the maximum number of links incident

on one node (processor) in the network. A network is said to have a uniform (constant) degree if all the nodes have the same degree, such as in the hypercube. An example of a non-uniform degree network is a mesh, where nodes in the interior have degree four and nodes on the perimeter have degree two or three.

- **The Diameter of an Interconnection Network**

The diameter of an interconnection network is the longest path between any two processors in the network. The radius is half the diameter.

- **Dilation**

The dilation is defined to be the stretching that the edge in the computation suffers from due to the mapping of nodes connected by the edge to distant processors in the architecture. The dilation of the mapping is the maximum distance in the architecture between nodes which are adjacent in the computation graph. The average dilation of the mapping is calculated by taking the average over all the dilations of the communication edges in the computation graph through the execution of the program. There is some subtlety involved in dilation measurement, since in most networks there will exist more than one path of varying lengths between any two processors in the system. We will assume that the minimum length path is the dilation of that communication edge. This assumption implies that the routing algorithm will consistently choose the optimal route (in terms of hop-count), which might not always be possible.

- **Load**

The load of a processor is the number of processes (computation nodes) that are mapped to it, and thus have to be time multiplexed on that processor. However, it is very important to note that this measure is not accurate, as it assumes that processes have comparable computational requirements. If indeed, a large variation in the processing time exists between the processes in the system, more sophisticated means for estimating the execution time of each process should be provided and the later should be used in calculating the load. Korry did a study of load assessment

in a distributed system consisting of a network of Sun 2 workstations [20]. His experiments showed that a combination of the processor utilization, the number of context switches, and the number of page swaps yielded the best result in terms of predicting response time. The load of a mapping is defined to be the maximum of all the loads of the processors involved in the mapping. Load is an indication of the computation cost, since, as the number of processes mapped to the same processor is increased, the overhead of context switching is greater. Throughout this thesis, the default definition will be the number of processes residing on a given processor, unless otherwise stated.

- **Communication Contention**

Contention over a link in the interconnection network occurs when two messages (or packets) attempt to use the link at the same time. Communication contention is costly for several reasons. First, since one of the messages will have to wait, the destination processor of that message will be idle while awaiting this message. Second, if a different route is chosen for one of the contending messages, most probably it will be longer and thus will take more time, in addition to increasing the probability of contention along the links of this longer route.

- **Completion Time**

Completion time is the gross figure that indicates the wall clock time the machine took to execute the program. This figure is often used in speedup measurements and not given directly since it is highly dependent on the architecture, compiler, load conditions, etc.

- **Speedup**

Speedup is by far the most common performance metric used in parallel processing. It is defined to be the ratio of the completion time of the program when run using more than one processor to the time it takes to run on one processor (i.e., sequentially) [12].

- Processor Utilization

A processor's utilization is the ratio of the time the processor was busy to the total execution time of the program. The average of the utilization of all the processors in the machine is taken to be the overall utilization figure. The drawback of this metric is that it does not indicate the percentage of time useful work is done as opposed to overhead such as packing and unpacking of messages. An ironic example is that when the system is thrashing, almost all the processors will exhibit very high utilization factors.

Taxonomy for Load Balancing Algorithms

There has not been a standard unifying classification model for load balancing algorithms in the literature. Below are several classifications (some of which are orthogonal) which will be used throughout the paper. It is important to note that some of these classifications might seem very similar. This taxonomy is based on and contains extensions to those presented in [9], [20].

- Adaptive vs. Preset

An adaptive algorithm is one that uses the system's state information in making its decisions. A preset algorithm has a canned fixed strategy which it always executes independently of the state of the system.

- Static vs. Dynamic

A static mapping algorithm is one that operates on a static computation graph, i.e., a graph in which the number of processes and their communication edges are fixed. In other words, no more processes will be spawned and no more communication edges will be established. Hence, a static mapper can get all the information it needs at compile time. On the other hand, a dynamic mapper is one that maps dynamically evolving computations (i.e., graphs that shrink and grow). Hence, dynamic mappers have to be invoked during the execution of the computation whenever a new process is spawned or when some other event occurs.

- **Deterministic vs. Probabilistic**

A probabilistic mapping algorithm is one that associates a probability with each of the possible destinations of the transferred node. A deterministic algorithm is one that is certain about the destination node for the transfer.

- **Sender Initiated vs. Receiver Initiated**

A sender initiated algorithm is one in which lightly loaded processors ask for more load. In a receiver initiated algorithm a congested processor tries to transfer a task to some other lightly loaded processor.

- **Topology Dependent vs. Topology Independent**

Whether or not an algorithm depends on the interconnection topology of a system is an important metric in evaluating a mapping algorithm, particularly if the algorithm is to be used on a system with a reconfigurable interconnection network (IN). An algorithm that was designed to work on a particular IN is highly likely not to perform well on another IN, if it works at all. On the other hand, for machines with fixed IN, an algorithm that was tailored for such an IN will probably be optimal compared to a generic topology independent algorithm.

- **Permitted Migration vs. Prohibited Migration**

A mapping algorithm that permits process migration is one that allows processes to be transferred after they were started. Clearly, this constitutes high run-time overhead since it involves the transfer of large status tables over the interconnection network. In an algorithm that prohibits process migration, once a process is started on a given processor, it will stay there until it terminates.

- **Fairness**

Fairness is the guarantee that no process in the system suffers from indefinite postponement. This criterion is most commonly used in conjunction with schedulers, but since some of the load balancing algorithms do process allocation as well (as opposed to doing the balancing stage after the allocation is done) it is important to

incorporate fairness as a classification criterion.

Load Balancing Algorithms

Most previous work in load balancing addressed the problem of balancing load across the nodes of a distributed system and local area networks. In both cases, a node is an independent computer. A survey of some of those algorithms that are relevant to our approach will be given in the following sections. We only consider algorithms for homogeneous systems. An interesting common feature of all these algorithms is that they are all independent of the interconnection network topology, i.e., architecture independent. The algorithms vary in their complexity according to the amount of system state information they collect.

Pressure Gradient

The gradient model load balancing algorithm introduced by Lin and Keller is a topology-independent adaptive algorithm [23]. The algorithm is based on the idea that a lightly loaded processor triggers the balancing mechanism by informing its immediate neighbors that it demands more work load. The algorithm is composed of two basic steps. First, each processor determines its own loading condition and sends this information to its neighbors. The second step establishes a gradient surface. Each processor should calculate its proximity value, where the proximity of a processor is defined to be its distance from a lightly loaded processor. The gradient surface is the aggregate of all the proximities. However, since proximity calculation requires the collection of all the proximities from all the processors in the system, they use a pressure gradient to approximate the gradient surface. The propagated pressure of a processor is defined to be zero if the processor is lightly loaded. Otherwise, the pressure is one plus the minimum of the propagated pressures of its immediate neighbors. Notice that this propagated pressure reflects the closeness of a processor to a lightly loaded processor. The pressure surface of the network is the collection of the propagated pressures of all processors.

This algorithm can be regarded as receiver initiated, since task transfers are indi-

rectly initiated by lightly loaded processors. The algorithm is clearly deterministic and does not employ randomness at any level. Most importantly it is adaptive since the allocation decisions are influenced by the proximity information which is gathered at run-time.

Random

In a random load balancing algorithm, a processor decides where to map the new process by selecting a node at random and transferring the process to that node. It has been shown that if the system is restricted to such a simple strategy, it will certainly exhibit the thrashing phenomenon in the steady state under high loads. A simple cure for this incorporates a static transfer limit that restricts the number of times that a task can be transferred.

This algorithm is an example of a preset mapping algorithm since it does not collect any state information and thus does not react to changes in the system state. Purely random allocation algorithms are rapidly gaining popularity and almost any distributed or parallel environment that has been implemented has employed some variant of a simple random allocation algorithm. The most obvious benefits are simplicity and scalability which are direct consequences of randomness.

Threshold

In a system based on thresholds, a static threshold of the maximum number of processes on a given processor is predetermined. To transfer a process, the sender probes a candidate receiver to see if transferring that process would place the receiver above the threshold. If so, another candidate receiver is picked, otherwise, the task is transferred to that processor. A control policy has to be incorporated to govern this probing scheme. One proposed policy is to continue with the probing until either a viable processor accepts the task or the number of probes exceeds a static predefined probe limit.

We notice that the threshold is not an upper bound on the load maintained by the algorithm, since when the static limit on the number of probes is reached, the receiver's load is actually placed over the threshold. Thus, a tight bound on the load achieved by

this algorithm cannot be determined.

Shortest Queue

A variation on the threshold algorithm tries to make the best assignment based on the amount of information it collects. A sender probes a group of receivers seeking one that can host the task. The processor with the shortest queue length is chosen to host the task unless the transfer would make it exceed the load threshold. In the latter case, the sender processor is obliged to process the task locally. This algorithm also does not have a tight bound on the maximum load of the mapping it provides.

Conclusions

The results of the work on the load balancing in homogeneous systems indicate that simple strategies which do not react to system state fluctuations, and thus have very little run time overhead, work nearly as well as complex strategies which collect a lot of information. In other words, according to the classification provided earlier, this means that algorithms that use a preset strategy are to be preferred to adaptive algorithms. It has also been shown that sender initiated policies outperform receiver initiated methods at light to moderate loads [10].

Dynamically evolving computations arise very often in a number of programming techniques. The difficulties that arise in executing these computations are due to the fact that very little information about the computation can be inferred statically and thus mapping algorithms have to be incorporated at run time. Thus, the overhead due to the complexity of the mapping algorithm has a great impact on the total execution time of the computation. Extensive research on load balancing algorithms for distributed computing and parallel processing systems has shown that simple randomized algorithms are preferred to complex deterministic ones.

CHAPTER III

PROCESS ALLOCATION AS A GRAPH EMBEDDING PROBLEM

Dynamic Graph Embeddings

A parallel algorithm can be viewed as a partial order on the set of processes that comprise the algorithm. The relation of the partial order is the “precedes” relationship [21]. Thus, two processes are said to be concurrent if they are incomparable with respect to that relation. Dynamic computations can be divided into bounded computations, where a bound on the total number of processes can be determined, and unbounded computations, where there is no limit to the number of processes that might be spawned. Because of non-determinism, computations generated by the OM execution model are of the unbounded type. The AND/OR tree is the canonical graph-representation of the computation. This motivated us to study graph-theoretic algorithms that embed dynamically evolving graphs into graphs of interconnection networks, and use their ideas in devising tailored process allocation algorithms for OM.

Algorithms for Mapping Dynamic Trees onto Hypercubes

There has been considerable interest in mapping tree computations to a variety of target architectures. The motivation being that tree computations are generated by many programming techniques such as divide and conquer and backtracking.

Recently, two randomized algorithms were presented for embedding dynamically evolving trees in hypercubes [2], [22]. The algorithm introduced by Batt and Cai embeds a dynamically growing binary tree of unbounded size in the hypercube [2]. Their algorithm maintains an M vertex binary tree dynamically on an N node hypercube with dilation $O(\log \log N)$ and with probability $1 - \frac{1}{\text{poly}(N)}$ no processor has more than $O(1 + \frac{M}{N})$ processes. This means that the constant load property and the bound on the dilation are

independent of the size of the tree. Thus, the algorithm is general enough to handle trees whose size cannot be predetermined. The probabilistic component of the algorithm is the step that handles spawning new processes. This is done by taking a short random walk and allocating the subtree whose root is the new spawned process to the subcube starting at the destination of the walk.

The random walk is constructed by choosing the next dimension to walk along at random, i.e., with probability $\frac{1}{n}$, where n is the dimension of the hypercube.

This algorithm can be classified as a preset algorithm and is very much topology dependent, since it was designed specifically to map binary trees to hypercubes and the properties of both are utilized in the analysis of the algorithm. We will later discuss how to generalize the algorithm to work on other architectures with reasonable performance. It is also clear that the algorithm is sender-initiated since a walk is initiated only when a new process is spawned. It is important to note that the algorithm maintains constant load only if the number of nodes in the tree is of the same order of magnitude as the number of nodes in the hypercube N , i.e., if $M = C \times N$, where C is a constant.

The algorithm also prohibits process migration since it is observed that no control over dilation can be guaranteed if process migration is allowed.

The second interesting algorithm is the one proposed by Leighton, Newman, and Schwabe, where their goals are to achieve constant dilation, and maintain constant load with high probability [22]. The assumptions they made were that the upper bound on the number of nodes in the tree must be N , i.e., the number of nodes in the hypercube.

The algorithm is also randomized and in some sense constructs a random walk similar to that of [2] but the use of randomness is different and the walk length is dependent on the depth of the node in the tree and the maximum allowed dilation b . In their algorithm, the candidate dimensions traversed by the walk depend on b and the depth of the node in the tree, where each candidate dimension has a probability of 0.5 of being actually traversed in the walk. The basic result of the algorithm is that any star centered at a node x in the hypercube will get at most $O(n)$ tree nodes mapped to it, where a star centered at node x is defined to be the set of nodes consisting of the n neighbors of x and

x itself (where n is the dimension of the cube).

A very important result obtained by [22] is that no deterministic algorithm can achieve constant load without having a dilation of $\Omega(\sqrt{\log N})$.

Observations

It is important to note that both algorithms have preset strategies that are always applied independent of the system state. This also complies with the results from distributed systems load balancing algorithms, where it was shown that preset algorithms are better than complex adaptive ones with high overhead. Another similarity is the observation that both algorithms use sender initiated strategies for handling newly spawned processes.

The similarity between the results obtained here and those obtained in the distributed systems work indicates that the issues involved in both fields are very similar and thus we believe that research in mapping algorithms to multicomputers should benefit from previous work on load balancing algorithms for distributed systems. This study is one such attempt.

Performance Metrics

Both algorithms considered load and dilation as their performance metrics. In both algorithms, one metric is good in the deterministic sense while the other is good in the probability sense. Clearly, it is impossible to achieve optimal or even good performance according to both metrics deterministically since the two goals are often conflicting and thus improving one will tend to worsen the other. A trivial example is the case when all the processes are mapped to the same processor which obviously makes the dilation zero. However the load of the mapping is maximum, i.e., no load balancing is achieved. The opposite extreme case is when every processor has an equal number of processes mapped to it. In most cases such a mapping will have high dilation.

The degree of parallelism is another metric which we believe is also vital in the overall performance of a parallel algorithm especially those whose computation graphs are

trees. This metric is especially valuable in precedence constrained computations such as the dynamically evolving AND/OR trees where the notion of time is natural and apparent. The following is an example that illustrates the value of this metric. Consider the case when two processes that are completely independent, i.e., can run in parallel, for example, processes that are in different subtrees, get allocated to the same processor. They will have to be time multiplexed and the time of the computation is at best doubled (see Figure 1).

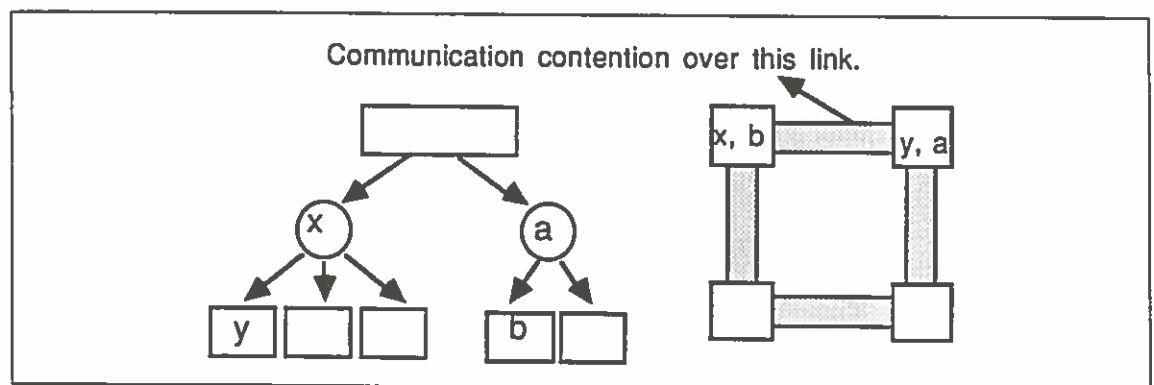


Figure 1: An Example of Contention in a Mapping

Thus, we believe that an algorithm that tends to achieve a good compromise between these metrics is highly likely to be superior in the overall completion time metric. One possibility for achieving the compromise between the metrics is to use a weighting function that incorporates all the conflicting performance indices.

User Guided Process Allocation

The parallel programming field is now mature enough that there exists a large body of parallel algorithms that were initially designed to be executed on parallel machines, as opposed to the previous trend of parallelizing existing sequential algorithms.

A common property in most of these originally parallel algorithms is that they usually explicitly specify what constitutes a process and do not leave it up to the implementation environment to partition the code into tasks that can be run in parallel. It is

also noted that a cognitive model such as a process graph is almost always employed in the design stage of the algorithm and most probably in its analysis.

We believe that only specifying the processes is not sufficient to efficiently execute parallel programs. It is also necessary to provide additional information that will help the compiler (and any additional software components involved) make the right decisions concerning the execution of the parallel program on the target architecture. It is clear that among the most important and vital information that should be provided is the mapping-relevant issues such as communication patterns and phase behavior.

This approach has been adopted by several researchers in academia, and has resulted in the development of a few configuration languages such as Conic [28], GDL [4], and LaRCS [26], [25]. Most of these languages are based on a graph model as the canonical representation of the program. These languages are intended to aid the programmer in describing the configuration of the processes in the program and their communication and execution characteristics. Such information could be utilized by the mapper-compiler combination to make informed allocation decisions.

We believe that if the programmer is to be allowed to describe extra-logical features of her/his program such as the execution and communication patterns, such an approach might well be used in parallel logic programming also. In other words, the programmer would write her/his normal parallel logic program in OPAL, and separately describe its characteristics in a configuration language. Both programs would be fed to the corresponding compilers and then linked together to form a load module. The linking phase should provide information for the process allocator so that it could make intelligent decisions at run time based on the information supplied by the programmer in the configuration program.

Conclusions

This chapter addressed the process allocation problem from a graph embedding perspective. A couple of recent algorithms that embed dynamically evolving trees on hypercubes were presented. Some generalizations to these algorithms to work on generic

architectures and generic AND/OR trees will be discussed in a later chapter. We also presented some ideas as to how to incorporate dynamic mapping information in the LaRCS configuration language. This would be of interest if annotations are to be employed in logic programming systems.

CHAPTER IV

LOAD BALANCING IN PARALLEL LOGIC PROGRAMMING

Introduction

Researchers in the parallel logic programming (PLP) area have recently started addressing the problem of process allocation and load balancing in the context of available PLP systems. Until very recently most PLP systems have been built around an off the shelf generic process allocation algorithm. This was encouraged by the fact that a large body of load balancing algorithms were available, and it was usually an easy task to select one which would yield acceptable performance on the average. However, clearly there was no global consensus on what acceptable should denote as most of these algorithms were heuristics. Recently, researchers started addressing the problem of designing tailored process allocation algorithms for parallel logic programming execution models. In this chapter, we will survey those research efforts in the context of message passing parallel logic programming systems and evaluate each of the presented algorithms according to the criteria presented in chapter II.

The Processing Power Plane

A processing power plane (referred to as PPP) is an abstract model that is intended as an intermediate level between the physical hardware and the application programs [33]. The motivation of the PPP is to provide means for modeling communication cost without exposing the details of the architecture to the application programs. Thus, somehow, the distance between the physical processors had to be incorporated in the PPP. This led the designers to modeling the physical structure of the architecture by considering the system to be an N -dimensional cube in which computing power is uniformly distributed. Since the algorithm was designed to be implemented on the Multi-PSI/v2 architecture, a loosely

coupled multiprocessor system with a 2-dimensional mesh interconnection network, N was chosen to be 2. The model is a 2-dimensional cube, i.e., a plane, hence the name processing power plane. We note, however, that this model assumes a homogeneous architecture where all the processors are identical and would not accurately model a heterogeneous system.

The algorithm breaks up the allocation process into two consecutive stages. The first stage is allocating the process to a certain location on the processing plane. The second stage is the actual allocation of the processing plane to the physical processor. The load balancing phase takes place at the level of each PPP. Finally, a dynamic reallocation phase makes localized enhancements based on collecting local load information.

- Load balancing on the PPP level

Since the algorithm uses the notion that the size of the rectangle allocated to the process determines how much computation power is given to that process, some estimate of the computation power needed by each process is required. The designers rightly state that such information might not always be feasible to specify. However, one proposal was to have the user provide this information as an annotation to the program. For example, for Prolog-like languages, an example of such annotations is:

```
p :- ← (2 * q), → r.
```

This specification states that the plane given to the predicate p is subdivided for q and r and that the subgoal q should be given twice as large a subplane as the subgoal r . Clearly, when such information can not accurately be provided, the user can give relative estimates as to how much power each process needs. One reasonable approach to estimating this is the size of the arguments to the goal.

- Load balancing on the Physical level

The PPP must be covered by the physical processors in the system for the program to be executed. For optimal load balancing, every processor in the system should be

responsible for an equal area of the PPP. This, however, might not be the case, as some areas of the PPP might get more load than others, in which case, appropriate adjustments to the size of the PPP region each processor is responsible for should be made. Processors responsible for dense areas should narrow their regions, whereas those responsible for sparse areas should widen theirs.

The authors point out the following example in which physical proximity on the PPP is not preserved in the mapping to the architecture which causes higher communication overhead. Consider the case where a process is allocated near the boundary of the region allocated to a certain processor and it communicates very frequently with another process allocated on the opposite side of the boundary. Although these processes are almost adjacent on the PPP, the cost of their communication will actually be much higher than the communication between that process with processes that are incident on the same processor. The problem here stems from the fact that physical proximity on the PPP was not preserved on the architecture. The distance between the processes on the PPP was stretched when mapped onto the architecture. This is exactly analogous to the dilation we defined in our graph model. The authors mention that some randomness in mapping the PPP to the physical processors might be useful in alleviating this problem.

The reallocation stage is best done by having a centralized controller that would attempt to achieve a better balance of the load on each processor. However, this clearly would require gathering a huge amount of global information which entails an enormous number of messages exchanged between all the processors and that controller. Moreover, such a centralized controller clearly makes the system non-fault-tolerant and is itself a bottleneck. Therefore, a simpler localized scheme was chosen to do the reallocation. Each corner point between every four processors acts as the reallocation pivot, where messages are exchanged between the four processors to determine the load of each processor and how load balancing could be achieved amongst them. This process repeats for every corner point in the PPP until computational load diffuses throughout the plane.

The PPP Algorithm on the Multi-PSI

We will briefly describe the architecture of the Multi-PSI machine as it is the target architecture for the PPP algorithm and the following one. It is also one of the primary computing resources that was developed by the fifth generation computer systems (FGCS) project. The Multi-PSI is a loosely coupled MIMD machine that has a 2-dimensional mesh interconnection network. The machine can have up to 64 processors in an 8×8 network. The processors are those of the PSI-II. The machine was basically designed to run the concurrent logic programming language KL1. KL1 is based on Flat GHC augmented by metaprogramming and pragma facilities. In fact, the operating system of the machine, PIMOS, is wholly written in KL1. The performance of the Multi-PSI is measured in terms of reductions per second (RPS) where a reduction is a logical inference step.

Flat GHC is a concurrent logic programming paradigm that deviates from the pure logic programming model by assigning a set of guards for each clause. Note that only input unification is allowed for the guards. Once a guard succeeds, the procedure commits to the solution obtained by the clause, and no backtracking is allowed. Thus a typical clause looks like the following:

$$H(X_1, X_2, \dots, X_k) : -G_1, G_2, \dots, G_m \mid B_1, B_2, \dots, B_n.$$

Input unification means that if any argument X_j is a constant, it must match a corresponding constant in the call. In Prolog X_j may unify with a variable, but this “outputs” the constant to the calling environment, which is not allowed in FGHC.

An algorithm based on the PPP model described above was implemented on the Multi-PSI/v2 architecture. The algorithm uses the notion that throwing a goal basically amounts to sending a packet to the PE responsible for that point on the PPP where the goal is allocated. Had the goal thrower PE known the entire mapping of the PPP to the physical PEs determining the destination PE would have been trivial. However, as such global information is costly to gather, the designers developed an incremental packet transmission algorithm that uses local information, together with the hardware mechanism

to implement the algorithm using table look-up. We will not go into the details of these schemes as they are irrelevant to our discussion. The interested reader is referred to [33].

The authors note a few problems with the implementation among which the following are the most relevant to our perspective:

- Network deadlock

Since a packet may follow any direction at any given node in the network, potential network deadlock is possible. The use of a deadlock-free routing algorithm, however, would restrict the freedom of the direction of movement of packets. The authors note that since the network has a finite longest path, a deadlock-free network architecture with a packet-pool buffer could have been used. For a larger scale machine, more efficient algorithms will be required.

- Load Measurement

The authors, rightly, note that their notion of “load” is not accurate and propose that a better approach might be to assign priority values to processes and use the sum of such values of processes executed per unit time as a measure of load.

In addition to the self-criticism presented by the authors of the algorithm, we will evaluate it in light of the classification criteria we introduced in chapter II.

- Sender initiated or receiver initiated

This algorithm can not be classified as being sender initiated or receiver initiated per se as the mapping of each process (goal) to a location on a PPP is done at compile time. At run time, whenever a goal is thrown to be solved, all is required is to determine which processor that location on the PPP was allocated to. Thus in some sense the allocation is static or compile-time and only reallocation enhancements are dynamic or run-time. Hence, the algorithm is independent of the sender vs receiver initiated criterion.

- Not fully distributed

The algorithm is not fully distributed and the initial allocation of the processes to the

PPP was not said to be done in parallel. Hence, a centralized processor is assumed to be responsible for that stage. Also, the goal thrower seems to be a dedicated PE that would be responsible for determining which goal is to be solved next and sending it to the array where the packet transmission algorithm ships it to its final destination PE.

- **Randomized**

Randomness was said to be incorporated at the level of mapping the PPP to the physical processors.

- **Architecture independence**

This algorithm is not architecture-independent as it assumes a homogeneous architecture. It also has the mesh interconnection built into the packet transmission algorithm and it relies on custom hardware that would make the realization of the localized packet transmission algorithms feasible. However, we believe that the main scheme of the PPP could be incorporated in any MIMD machine, assuming routing algorithms exist that are capable of implementing the packet transmission algorithms. The existence of some sort of a host that would act as the goal thrower and would perform the initial process allocation to the PPP is also assumed. This latter assumption, however, is reasonable and realistic enough as most commercially available MIMD machines have such a host node.

- **Process migration**

Since the reallocation stage is done dynamically (i.e., at run time) it is effectively doing process migration. Thus, it constitutes high run-time overhead due to the transfer of process tables and structures from one processor to the other over the interconnection network.

Multi-level Load Balancing

The Multi-level load balancing algorithm was designed specifically for OR-parallel exhaustive search programs [13]. The target architecture for which it was designed was also

the Multi-PSI machine. However, the algorithm is claimed to be architecture-independent. The main idea of the algorithm is to partition the program into mutually independent tasks (subtask generation) and then distribute those subtasks to the physical processors (subtask allocation).

The multi-level load balancing algorithm was arrived at as an extension of a one-level scheme that simply used one master processor as the subtask generator that generates subtask to be distributed to the other slave PEs in the system. However, it was clear that the subtask supply was the bottleneck of the system as the number of processors in the system increased. Hence, a natural solution to alleviate that problem was to incorporate more subtask generators.

The multi-level algorithm divides the processors into groups, each with its own subtask generator, and attempts to balance the load at both the group level and the processor level. The generated subtasks are allocated to the PEs according to the task allocation strategy.

A "super-subtask" generator is allocated to one master PE which divides the program into super-subtasks until the search reaches the first distribution level. The super-subtask generator distributes super-subtasks to M group masters which will act as subtask generators. Each group master is responsible for a fixed number of processors whose number is $\frac{N}{M}$ where N is the number of processors available in the system. The first level of distribution is the distribution of super-subtasks to idle group masters to balance the load across processor groups (PG). The second level is the load balancing within a group where subtasks are distributed to idle PEs (see Figure 2). As mentioned earlier, the algorithm was motivated by the idea of avoiding the subtask bottleneck by keeping the number of processors competing for subtasks small. However, if N becomes increasingly large, a potential bottleneck can still occur within groups, in which case, more distribution levels should be introduced. Clearly, it is this hierarchical scheme that makes the algorithm scalable and hence portable to families of architectures.

The allocation strategy is basically a simple receiver initiated (or on-demand) strategy. Idle PE's send a demand message to the subtask generator which in turn distributes

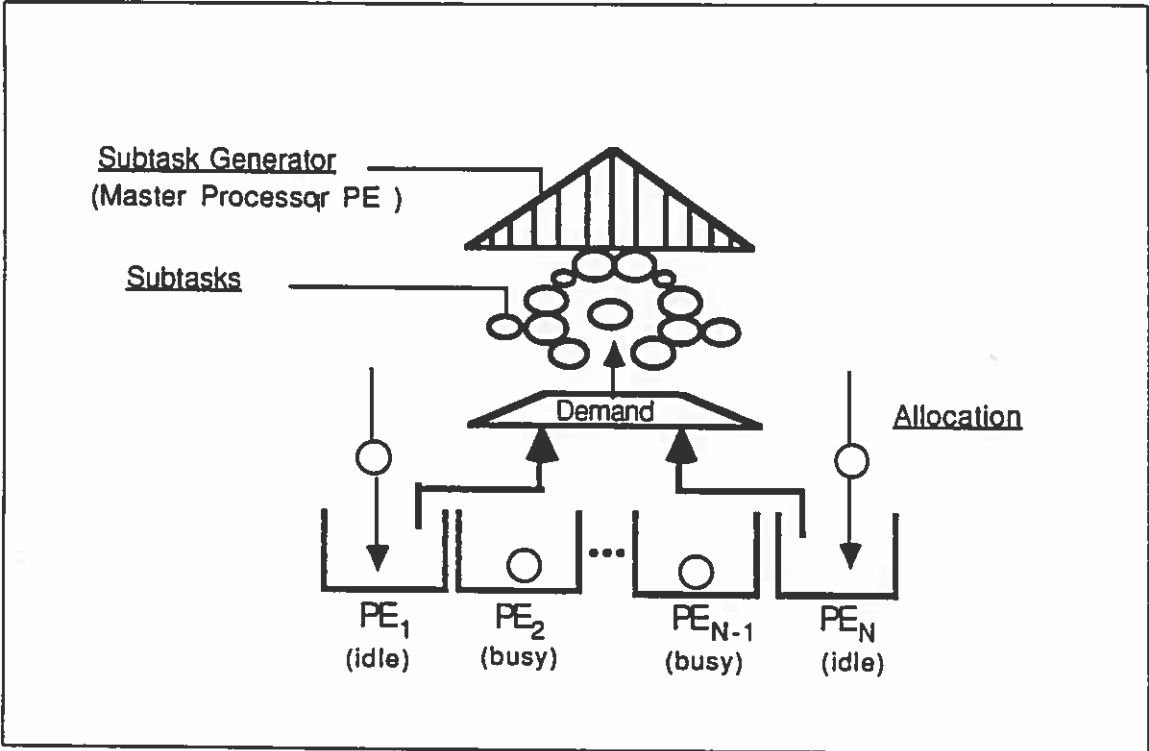


Figure 2: Subtask Distribution in the Multi-level Algorithm (from [13])

tasks to the idle PE's which execute these tasks and then send a new demand message upon completion of their tasks.

A technique called group merging is employed to handle the situation when a group of processors become idle because they finished executing their subtasks whereas other groups are still busy. Clearly, such a situation arises as a result of load imbalance at the super-subtask level. The situation is remedied by a scheme that merges idle groups with busy groups to form larger units. However, as groups are merged the subtask generator of the newly formed group is now responsible for feeding a much larger number of processors which creates the evil subtask bottleneck.

The algorithm was tested on the all solution exhaustive search program of the Packing Piece Puzzle on the Multi-PSI machine and almost linear speedups were obtained when varying the number of processors.

Analyzing the multi-level load balancing scheme according to the dimensions introduced in chapter II yields the following observations:

- Receiver initiated

The multi-level algorithm is receiver initiated. A study by Eager *et al* [10] showed that under heavy system load, receiver-initiated strategies perform better than sender-initiated strategies. This result complies with the measurements taken from the performance study that was conducted to test this algorithm. In fact, since this algorithm is targeted towards exhaustive search OR-parallel programs, we believe that it was a very clever design decision to use a receiver-initiated strategy.

- Randomness

When several processors are idle and requesting work and there are enough subtasks for all of them, randomness could be used as the arbitration scheme.

- Not fully distributed

This point is of great interest here, since the algorithm is distributed when you take the viewpoint of the processor masters alone and view the PEs in the processor groups as simply the computational power of the masters. However, if one takes the

global view of the system, then clearly the group masters constitute centralization points which are vulnerable points in terms of faults, since a breakdown at one of the master PEs would freeze all the PEs in the processor group controlled by that master.

- Architecture independence

The algorithm is fairly architecture independent since no knowledge of the architecture was employed at any point in the distribution scheme. While group merging is influenced by the mesh interconnection, we believe that it could very easily be modified in order to group the processors in a different fashion that is independent of the topology of the network, for instance, grouping the one-away neighbors of an idle group.

- Process migration

It is not explicitly stated whether process migration is permitted when processor groups are merged or not.

Load Dispatching Strategies for the Parallel Inference Machine

Another machine built as part of the Japanese fifth generation computer systems project is known as the Parallel Inference Machine (PIM). The PIM is a message passing MIMD machine that is currently under construction. Similar to the Multi-PSI, the PIM also executes programs written in the KL-1 language. Load balancing strategies for the PIM are described in this section.

The PIM has an interesting hierarchical architecture that in essence resembles that of the *cm**. The bottom level is made of the individual PEs. The next level up is called the cluster-layer, where a cluster is a group of tightly coupled PE's. The third level is the bunch layer, where a bunch is a group of loosely coupled clusters. The top level is the integration layer where an integration is a group of bunches.

The machine that was actually being built had only a 2-layer hierarchy, i.e., up to the bunch layer. There were 10 clusters interconnected using an equal length network such

as a cross-bar switch. Each cluster had 10 PEs coupled tightly through shared memory and caches.

Several algorithms for load dispatching on the PIM have been presented [31]. The algorithms are basically influenced by two main ideas: first, anticipate idle targets to dispatch load to, and second, stop load dispatching when conditions are not appropriate, such as the sender becoming idle as a result of the dispatch or if the target is much more overloaded than the sender.

All the algorithms were tested using simulations on a virtual machine emulating the PIM. The main results in the paper are drawn from one test program, the 6-Queens program; however similar results were reported to be obtained from running the algorithms on two other parsing programs.

Four strategies were used:

- Strategy A: The clusters to which goals are dispatched is determined totally at random.
- Strategy B: The cluster with the minimum ready goals is chosen as the target of the dispatch. This strategy uses the number of processes in the ready queue as an estimate of the load.
- Strategy C: Same as strategy A but the dispatch is aborted if the target cluster has more ready goals than the dispatching cluster. As the authors do not specify whether the dispatch is retried in hope of obtaining a different target, we assume that once aborted the task would simply be allocated locally.
- Strategy D: The cluster with maximum ready goals dispatches to that with minimum ready goals. This strategy requires global information about the load of all the clusters in the system.

A minor modification is also made to strategies B and C, resulting in strategies B' and C'. The modification is that before dispatching is started, the load of the dispatching cluster is first compared to a threshold minimum below which dispatching is aborted.

Dispatching while the load is below the threshold might result in the dispatching cluster becoming idle very soon. Analytical or statistical analysis to determine this threshold value was not provided

The goals of the authors were to minimize the parallel processing overhead and to maximize the processor utilization. The parallel processing overhead is defined to be the overhead due to the suspend/resume processes that are supported in KL-1. Recall that processor utilization is the ratio of the processor's busy time to the total time of execution of the program. It is interesting to note that both these factors depend on the load granularity in conflicting manners. For instance, the finer the granularity, the higher the processor utilization, and the higher the parallel processing overhead also. On the other hand, the bigger the grain size, the lower the parallel processing overhead, and the lower the processor utilization. Note that these factors are much more significant at the bunch layer (i.e., between clusters) than at the cluster layer (i.e., between PEs). Inside a cluster, load balancing can be achieved using frequent inexpensive communication. Also the shared memory can be used for storing the queue of ready processes, which decreases the cost of suspension and resumption.

the criterion that the authors used was the load dispatching rate, which is defined as the ratio of all goals dispatched to the total number of reduced goals. Their definition of granularity is that it is the reciprocal of the load dispatching rate. Clearly, at high load dispatching rates, the parallel processing overhead dominates the processing time, whereas at low dispatching rates, utilization is higher.

the following properties:

- Sender initiated

The authors deliberately made the decision that the algorithms be sender initiated as they claim that for a large scale PIM (i.e., with more than 100 processors) the communication overhead due to broadcasting load requests would place severe requirements on the channels. This decision was influenced by the architecture rather than the nature of the application programs (i.e., OR vs. AND parallelism) as was the case in the other efforts.

- Randomness

The authors incorporated randomness in determining the target of the allocation in all algorithms. In fact, strategy A is a blindly random one. This, we believe, is well in accordance with the decision to use sender-initiated allocation.

- Fully distributed

The algorithms are fully distributed as all clusters execute the same allocation routine whenever a new process is spawned. The algorithm is fault-tolerant and insensitive to failures at the cluster level.

- Process Migration

Once a process is allocated to cluster, it is not allowed to migrate to other clusters. This was also motivated by the cost of communication across clusters.

- Architecture independence

While several of the design decisions that influenced the algorithms were motivated by the architecture, the resultant algorithms are very architecture independent and could be very easily run on various architectural platforms.

ideas might be interesting extensions to the PIM load balancing schemes:

- Hierarchical load balancing

The idea here is to incorporate the multi-level load balancing scheme, as the clusters lend themselves very naturally to the concept of processor groups. Thus one processor in each cluster would serve as the subtask generator for the rest of the PEs in that cluster and try to balance the load within the cluster.

- Hybrid load balancing

While sender initiated strategies might be good for inter-cluster load balancing, a receiver-initiated strategy might very well be more appropriate within clusters since the cost of communication is much less and hence broadcasting load information would not be intolerable. Within the cluster, a receiver initiated strategy such as that reported in the multi-level load balancing algorithm might be more powerful

for intra-cluster load balancing. We will pursue this idea in one of the algorithms devised for OM to evaluate whether the performance of a hybrid strategies warrants their extra overhead or not.

Contracting Within Neighborhood

been tested in a real setting and on more than one despite the fact that they are still toy programs. An algorithm known as contracting within neighborhood is used in the Chare-Kernel which is the base implementation on top of which the Reduce OR model is implemented [18]. The algorithm was first proposed in [19] and has been extensively modified and extended to be responsive to system changes in [30]. The original algorithm is called the Naive Contracting Within Neighborhood (NCWN), while the new adaptive version is called ACWN.

The NCWN algorithm is sender initiated and attempts to localize communication as much as possible by avoiding communication between non neighboring processors. The motivation is that such communication is not scalable and would tend to make the cost of communication overhead dominate the performance. Henceforth, we will use our generalized notion of a “star” to refer to the immediate neighbors of a processor.

The algorithm has two parameters, min-hops and max-hops. These two parameters are the implementations of the horizon and the radius , respectively. The horizon is the minimum distance a process should travel from its source. The radius is the maximum distance a process is allowed to travel. It is important to distinguish between the distance and the hop-count, since, in a thrashing situation, a process could have a huge hop-count without traveling any distance from its source. Each processor keeps track of the loads of its immediate neighbors (the star centered at that processor). Whenever a process is spawned it is sent to the least loaded processor in that star, and it is stamped that it has traveled one hop. The recipient of the process compares its hop count with the radius and the horizon, and accordingly decides whether to send it to the least loaded processor in this star or to put it in its local pool. If the hop count is less than the horizon the process must be shipped away. On the other hand if the hop count is greater than the horizon it

must be executed locally.

In the adaptive extension to the algorithm (ACWN) the horizon and the radius are calculated dynamically based on periodically gathered system load information. This new version is considerably more complex than its predecessor. The algorithm is composed of three phases. The first is the periodical update of the min-hops and max-hops parameters. The second phase is the original NCWN, while the third is a dynamic redistribution phase.

The adaptive component of the ACWN is expressed primarily in the min-hops and max-hops parameters. Each PE has its own version of these parameters. Let the k th PE load function be $F(k)$. Each PE periodically consults with its immediate neighbors to determine whether the neighborhood is in the light, moderate, or heavy load region. A function $B(k)$ returns the minimum of the loads of all the immediate neighbors of the k th processor. $B(k)$ is compared to two more load-range parameters, low-mark and high-mark, to determine which load status of the PE and the min-hops and max-hops are updated accordingly (see Table 1). Low-mark is used to switch between light load, where a ship away policy is adopted, and moderate load, where new processes are executed locally. On the other hand, high-mark is used to detect if the neighborhood is in the heavy load region and thus work should be kept locally. Experiments indicated that the value of low-mark between 2 and 5 and high-mark around 8 were satisfactory.

Table 1: Calculation of Min-hops and Max-hops in ACWN

STATE	min-hops	max-hops
light-load: $B(k) < \text{low-mark}$	MIN-HOPS	MAX-HOPS
moderate-load: $\text{low-mark} \leq B(k) < \text{high-mark}$	0	MAX-HOPS
heavy-load: $\text{high-mark} \leq B(k)$	0	0

Varying MAX-HOPS, the constant used to update the max-hops variable whenever a change in load status is detected, beyond a value of 3 did not influence performance notably.

The ACWN algorithm has the following properties:

- Sender initiated and fully distributed

The algorithm is clearly sender initiated and fully distributed as every processor in the system executes the same routine whenever a process is spawned.

- Architecture independent

The algorithm is architecture independent as it relies on the notion of immediate neighbors (stars) which is applicable to any topology.

- Prohibited migration

Processes are not allowed to migrate after being sent to their final destination. The author notes that this decision might be relaxed later as the processes in the system are largely fine grain, and the cost of moving them after being started would not constitute high overhead.

As the author notes, the algorithm rules out the possibility of executing a process locally, since the horizon is normally set to a value greater than 0. This is a limitation since sequential execution is sometimes the only alternative to avoid swamping the machine.

As the algorithm follows the steepest load gradient locally, the absolute minimum of the neighborhood is not guaranteed to be reached. An example that illustrates such a situation is depicted in Fig 3, where processor *a*, following the steepest gradient, would ship a new process to processor *b* even though *c* is the least loaded processor in the neighborhood of radius 2. This, however, is not a major limitation as only algorithms that collect global information, and hence introduce unbearable overhead, can locate the global minimum.

Regardless of the simplicity of the calculations used to assess the load status, we believe that periodically polling for load information as a background task will generate high message traffic that would contend with the communication needs of the program. A pathological case would be in a very heavy load situation, such as in an exhaustive search program (not necessarily with OR parallelism only) with considerably larger grain

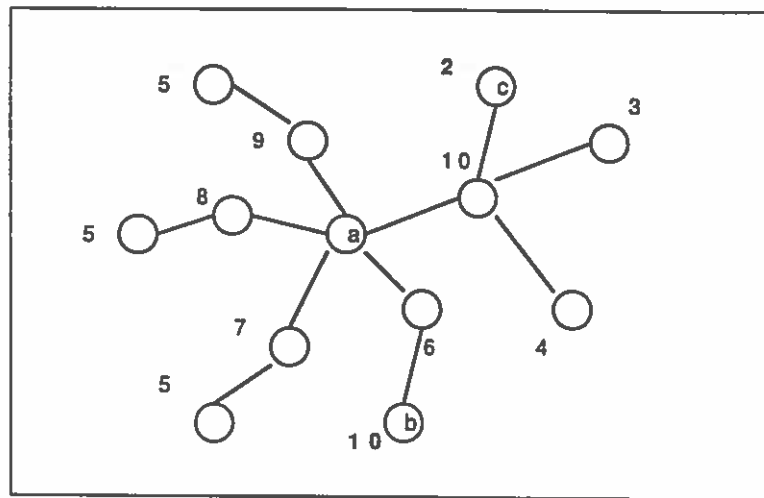


Figure 3: An Example of a Non-optimal Allocation in CWN

processes than those that arise in the Queens program. In that case, the processors would be constantly polling each other's load information just to find that the system is indeed in the heavy load state and no work should be shipped out.

CHAPTER V

OVERVIEW OF THE OPAL MACHINE

Introduction

The main contribution of this thesis is the development of task allocation strategies for OM, a virtual machine for executing programs of the AND/OR process model. This chapter contains a description of the AND/OR process model and its most important characteristics that are pertinent to process allocation and load balancing. The implementation of the OPAL Machine is then briefly described. Finally, the interface between the process allocation modules we will develop and the layers of the machine is described. Process allocation algorithms for OM are described in the next chapter.

Parallel Logic Programming Models

It is clear that the suitability of logic programming to parallelization stems from the nondeterminism that is inherent in its model of computation. There are two aspects of logic programming where nondeterminism arises. The first aspect is the subgoal selection (also known as the search rule) which is the order of execution of the procedure calls (or subgoals) in the body of a clause. The second is the choice of a clause of a given procedure. The clause selection is also known as the control rule. Logic programming as a computation model does not impose any restrictions on either the search rule or the control rule. It is the decision of using specific strategies to implement these rules that characterizes the various execution models of parallel logic programs.

There are two types of parallelism that most naturally arise in logic programming computation models, OR parallelism and AND parallelism. OR parallelism refers to the case when independent processes are assigned to the various clauses of a procedure; thus the worker processes are generating alternative solutions to the procedure. On the other

hand, in AND parallelism, processes co-operate in generating one particular solution to a query or a clause body. In exploiting AND parallelism one is faced with the problem of consistent bindings since clauses that share variables must bind the variables to the same values. Two approaches were devised to solve this problem: independent-AND parallelism and stream-AND parallelism. In independent-AND parallelism, only independent literals within a given goal can be solved in parallel. Literals are said to be independent if they do not share common variables (see the section on AND processes for details). Stream-AND parallelism is based on the producer-consumer model where one process acts as the producer of the bindings of the shared variables, while the other process acts as the consumer.

The AND/OR Process Model

The AND/OR process model is one of the few execution models that support both OR and AND parallelism [5], [6]. The AND/OR process model is an actors-style parallel execution model where processes can be regarded as inactive objects that are activated by messages.

As suggested by the name, there are two types of processes in the AND/OR model, AND processes and OR processes. Both types of processes can be either sequential or parallel. Sequential processes simply mimic the traditional depth first left to right Prolog execution model and thus would not be explained any further.

A parallel OR process is one that is created to solve one literal. In a pure logic programming language, solutions of the literal can be generated in any order. The process spawns as many descendants as there are clauses with heads that unify with the call to that procedure. Each of the descendants is an AND process for solving the body of the corresponding clause. The OR process can be executing in one of two modes: waiting mode or gathering mode. In waiting mode, the parent of the OR process has not received any success messages yet and thus is waiting for a solution from the OR process. On the other hand, an OR process is operating in the gathering mode when its parent has already received a solution and thus the OR process is just gathering the other solutions

and storing them in case they are needed by the parent (i.e., if the parent sends it a redo message).

Only independent-AND parallelism is currently supported in the model. A parallel AND process is responsible for solving a goal (or a clause body) with several literals in parallel. At compile time, an ordering algorithm based on a data dependency graph is used to order the literals of an AND process so that all independent literals are solved in parallel. Run time checks are used to verify the independence of the literals since groundness is not checked at compile time. The AND process eventually spawns as many OR processes as there are literals in the clause body. The reader is referred to [7] for more details.

There are four types of messages in the AND/OR process model:

- Success

Success messages carry the bindings of the variables and thus normally have the biggest volume. Clearly success messages only go up the tree. Originally success messages were only sent from a child to its immediate parent, which might in turn pass it up to its parent, and so forth. A new scheme of using success continuations and last call optimization will send success messages directly from a child to one of its non-immediate ancestors.

- Fail

A fail message is sent from a child to a parent reporting its failure to solve a goal.

- Start

A start message is the one that creates the seed of a new process and initializes the process structure with an initial process-id.

- Redo

A redo message is sent whenever a parent process needs another solution from one of its descendants. Such a situation may occur if all the solutions are required or if the previous solution could not be completed (for example some other descendent

failed to use those bindings, in case of an AND process). Redo messages are only sent from a parent to a child.

- **Cancel**

A cancel message is sent from a parent to a child to kill that process when solutions from that descendant are no longer needed. Such a situation could arise, for instance, in case of an OR process that has already obtained a solution. Another possible situation is when an AND process has failed due to the failure of one of its descendants; all other descendants will receive cancel messages.

Communication Characteristics of the AND/OR Model

One of the most important characteristics of the AND/OR model is that communication occurs only between a parent and its children and never between siblings. Moreover, in the original model, communication between a child and one of its non immediate ancestors did not exist. However, last call optimizations and the use of success continuations to avoid the forwarding of success messages could introduce communication between non immediate descendants. This type of optimization can be exploited in some AND processes. Our process allocation algorithms will not assume such optimizations and the locality of the AND/OR model will be assumed to persist independently of the type of parallelism found in the programs.

The execution wavefront is defined to be the set of processes that are active (executing) at a given point in time. It is significant to note that the processes that constitute the execution wavefront in the AND/OR model are not the ones in the frontier of the process tree.

The OPAL Programming Environment

The AND/OR model is the underlying computation model upon which an integrated programming environment has been built. The environment is composed of a logic programming language named OPAL (Oregon PARallel Logic) and a virtual machine named OM for executing compiled OPAL programs.

OPAL is a logic programming language which has almost the same syntax as Prolog except for the elimination of side effect operators and minor syntactic sugar (such as the use of "&" instead of commas for connecting the literals in the body of a clause). A compiler translates OPAL programs into the instructions of the OM virtual processor. It automatically breaks the program into pieces which will become parallel AND and OR processes.

The OPAL Machine

OM is a message passing virtual architecture based on the operational semantics of the AND/OR process model. The design of the OM instruction set was influenced by the WAM in areas such as unification instructions. However, the compilation schema for control instructions to implement OR and AND parallelism in OM are novel. The architecture of OM uses a heap-based memory organization. In a multicomputer implementation, there will be one copy of OM on each processor in the architecture. The OM responsible for solving the initial goal query (process) will be referred to as the Mother OM (MOM). This set of communicating OMs constitutes a higher level virtual parallel machine on top of the conventional virtual machine on each node. This higher level system is fully distributed, fault tolerant, and architecture independent. These features are primarily results of the decision to replicate the entire virtual machine on each node. The system is fully distributed because our process allocators and mailers are replicated on each node, and therefore no centralization bottlenecks are introduced. Fault tolerance is a consequence of being fully distributed, since aside from a failure in the host of the MOM, any node failure can be gracefully recovered by having other processors share the work of the failed processor. Clearly, the system is architecture independent, since a copy of OM is made to reside on every processor on the system independent of how the processors are connected. Moreover, the mailers and process allocators do not rely on any characteristics of the interconnection network.

There are three basic layers in OM: the kernel layer, the OS layer and the virtual machine layer (see Figure 4). The kernel is the minimal core of the system that constitutes

the lowest level of the machine. It handles functions such as memory allocation, changing the machine status between idle, running, and suspended, etc.

The OS layer is responsible for the handling the inter-PE message routing (mail forwarding), process allocation, handling suspension and resumption of processes, context switching, and maintenance of process, seed, and message queues.

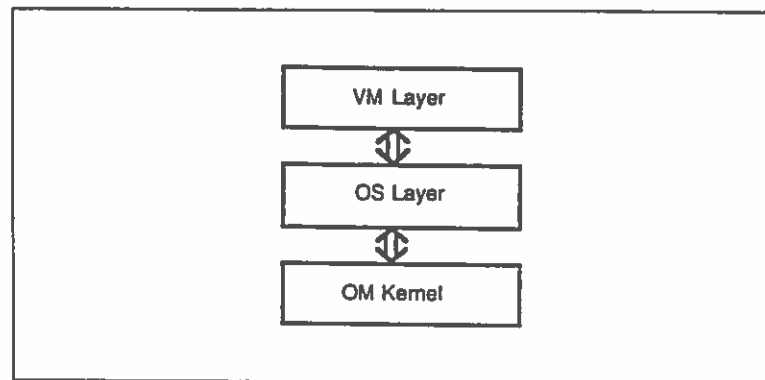


Figure 4: The Software Layers of OM

The virtual machine layer is the topmost layer and constitutes the interface to the outside world, i.e., the OPAL programs. In other words, from the point of view of the OPAL program, the virtual machine layer is the operating system.

The Process Allocator

The process allocator is one of the components of the OS layer in OM. Depending on whether a sender initiated or a receiver initiated strategy is used, the process allocator routine has a different interface. If the algorithm is sender initiated, the routine is called whenever a process is spawned. An OS routine, Ship-Seed calls the Spawn procedure in the process allocator in order to place the new seed. Once a process terminates, or receives a kill message, the Terminate procedure of the allocator is called to deallocate the process and adjust the various load and performance parameters. On the other hand, for receiver initiated schemes, the routine is invoked whenever a processor becomes idle.

For our new hybrid strategies, a mix of the above two calling sequences is employed. The interface between a sender initiated process allocator and the various components in the OS is depicted in Figure 5.

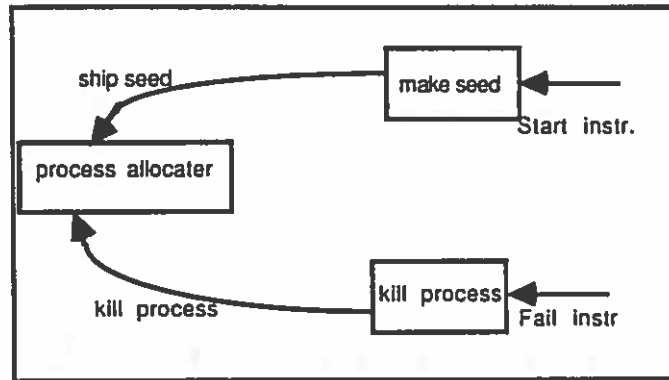


Figure 5: The Interface Between the Process Allocator and the OS Components

CHAPTER VI

PROCESS ALLOCATION ALGORITHMS FOR OM

In studying the AND/OR model and the architecture of OM, I focused on characterizing the communication and execution behavior of the processes and determining the parameters that might be necessary to obtain through compile time analysis and pass onto the dynamic load balancing routine.

Some interesting parameters of OPAL programs are the maximum (average) number of clauses per procedure and the maximum (average) number of the literals in the body of a clause. The first parameter indicates the maximum (average) number of children an OR process has, while the second gives the maximum (average) number of children an AND process has. It is not true that these two parameters have equal importance or could be employed in similar manner, because AND and OR processes have different characteristics (e.g. an AND process has to handle the problem of consistent bindings). Notice that the maximum of the above two maxima gives the (tight) upper bound on the number of children any process in the computation has, which is important to determine because it affects, in association with the degree of the network, the bound on dilation.

Since the AND/OR model is primarily a message passing execution model, the ratio of communication to computation is expected to be fairly high. Thus intuitively, it seems that dilation is a more crucial factor in improving the performance than balancing the load.

Among the five message types (start, success, fail, redo, and cancel) the success messages have the biggest volume and hence concentrating on the patterns in which these messages are sent is given higher priority. Clearly success messages are only sent upwards in the tree. Thus, if we consider any two processes, messages of significant volume can only be sent in one direction. This information is significant if we have an interconnection

network in which some of the links are unidirectional (I believe this is rare). More interestingly, in case of bidirectional communication links, we can consider quadruples of processes at a time. Consider the following case: x is an ancestor of y and a is an ancestor of b . If we map x and b to the processor 1 and y and a to processor 2, then the heavy communication between x and y will use the link in the direction $2 \Rightarrow 1$, whereas the communication between a and b will use it in the direction $1 \Rightarrow 2$. The above two communications do not conflict because the channel can accommodate duplex communication.

One key property of the communication characteristics of the processes of the AND/OR model is that neither siblings nor non-immediate descendants ever communicate. This information could be incorporated in the allocation strategy in a variety of ways: one approach might be to map the siblings to very distant processors, since we are sure they never communicate. Another approach is to always map one of the OR descendants of an AND process to the same processor as the AND process. This is motivated by the fact that none of the children of the OR process will attempt to communicate with the parent AND and hence no contention over that link will occur.

It is also significant to note that the processes that constitute the execution wavefront are not the ones in the frontier of the tree.

My initial intuition regarding the main characteristics of methods that should work well for OM are:

- Sender initiated

The processor at which the new process is spawned is responsible for consulting the process allocation routine to figure out the destination of that process (including the possibility that the destination might be to keep it to itself). Thus the task allocation is regarded as a function from the process-id space to the processor space. Notice that the scheme that was originally implemented in OM (the pressure gradient) was receiver initiated.

- Randomized

Since the general consensus in both the parallel and distributed computing commu-

nities is that placement of the processes should be random, and that very simple algorithms that collect little information at run time tend to perform nearly as well as complex ones that collect a lot of system state information, my heuristic will be randomized. However, from the study I did of the two recently developed randomized algorithms [2] and [22] for embedding dynamically evolving trees in hypercubes, I came to the conclusion that subtle differences in the use of randomness can indeed yield significant differences in the performance. Hence, random placement can be viewed as a broad design decision which has to be refined to its lower level components. In case of the hypercubes, for example, we could employ randomness to decide on a certain star in the cube, whereas allocation to a specific processor in the star might use another substrategy that might not be randomized.

- **Prohibited Migration**

Process migration should not be allowed after a process starts executing. This avoids structure copying and similar housekeeping work that would have been needed if full blown processes are allowed to migrate.

I devised the following algorithms specifically for OM, based on our understanding of the characteristics of the communication and execution patterns of the AND/OR model, and my analysis of the various algorithms presented in the previous chapters. It is important to note here that the algorithms are described in terms of the AND/OR tree as the underlying representation of the computation. By taking this approach we are in effect giving the operational semantics of our process allocators. This is to be distinguished from the approach usually taken in describing the schedulers and load balancing algorithms used in shared memory systems [3], which are almost always described as operating on the tree representing the solution space of the problem.

Generalized Random Walks

The initial step in this work was to generalize the algorithms to work for any message passing architecture and not just hypercube interconnected multicomputers. This

generalization took place in three key components of the algorithms: the construction of the random walk, the star definition, and the dilation measurement.

Generalized stars are the most natural extension of the concept. The star centered at node x will be defined to be the set of neighbors of node x that are one link away from x in addition to x itself. Clearly for non-uniform degree networks, not every node has the same star size. For example in meshes, nodes at the corners have stars of size 3 whereas nodes on the edge boundaries have stars of size 4, while the rest of the nodes have stars of size 5.

To generalize the concept of random walk, we have to induce an ordering of the neighbors of each node in the network. Effectively we are emulating the dimensions of the hypercube by making the analogy between choosing the k th neighbor and traversing the k th dimension. One has to deal with the non-uniformity problem here also. A simple solution was to use lexicographical ordering of the labels of the nodes. Whenever a dimension is to be traversed, a number between 0 and $N - 1$, where N is the size of that node's star, is generated and the neighbor corresponding to that number in the ordering is chosen to be visited.

It is important to note that we do not claim that the theoretical bounds on the algorithms will still hold after these generalizations but simulation results indicated that they still perform well.

The Crawl Algorithm

In this algorithm, the newly spawned process is allowed to take walks only of length one. Thus, effectively, only the processors of the star centered at the source processor are considered for allocation. Effectively, this algorithm lets the load spread slowly over the interconnection network in hope that processes spawned at close intervals need to communicate a lot whereas those having a long intervening time will rarely communicate. The process is shipped to the least loaded processor in the star and therefore avoids full localization. Thus, in effect, each processor has to know the load information of all the processors in its star.

ANDs Crawl, ORs Run

In this algorithm, we take the approach that since OR processes are independent and never communicate with each other, they should run away from each other to leave room in the network for AND processes to stay close to each other.

Walk With Your Neighbors

This algorithm is a modification to the dynamic embedding random walk algorithms. The main idea is that checking the star at the target might not be enough. Checking the star centered at the source could also be beneficial, since if an immediate neighbor of the source had been less loaded than all the processors in the target's star, then clearly the former is a much better choice since lower dilation would be achieved in addition to a better load balance.

Note that no time overhead is introduced by considering the star's source since the two stars can be checked in parallel and the minimum of the two minima can be compared to choose the final target of the walk. Thus, in effect the process is actually taking the neighbors of the source processor along with it during the walk.

The Rendezvous Algorithm

This algorithm is a combination of sender initiated and receiver initiated strategies. To our knowledge this is the first algorithm that incorporates both policies to be proposed either in the load balancing or in the logic programming literature. The basic idea of the algorithm is to have the idle processors and the newly generated processes (seeds) play a dating game where both search for potential mates. Only when a process meets an interested processor does it get shipped to that processor. In one variant of the algorithm, we require them to meet at the very same processor, while in another variant, just being in the same star is a close enough proximity to be considered a match.

Whenever a process is spawned it takes a virtual random walk only to look for potential host processors at the star it reaches. The process simulates the random walk

by generating the walk and then storing the id of the target processor it is bound for in its structure. The process is said to be looking for a mate at the star centered at that target id. Idle processors also go out looking for work by walking around the network looking for seeds to host. A processor is said to be walking by executing a loop that generates a sequence of processor-ids to which it sends a query message to find out if any processes are wandering around in that star. Since all idle processors follow this scheme, it is guaranteed that processors do not keep spinning forever while there are processes waiting to be picked up. Note that in this scheme every process experiences exactly one physical transfer since the first transfer (the random walk) is virtual. The second transfer does the actual physical placement of the process, when it gets picked up by its final destination processor.

The algorithm is composed of two components, spawn and demand. The spawn algorithm is executed whenever a new process is spawned. Demand is the algorithm executed whenever a processor becomes idle. Formally, the algorithm is described as follows:

Algorithm Rendezvous

Spawn:

1. Generate a virtual random walk of length WalkLen and store the id of the target processor in the process's BoundFor field.
2. Wait for a request from an idle processor (IdleId).
3. If Match(IdleId, BoundFor) then ship the process to the processor whose label is IdleId, otherwise keep waiting.

Demand:

1. For $i = \text{MinId}$ to MaxId do
 2. Send a Match query (tagged with the processor's label, MyId) to every other processor i .
 3. If Match(MyId, BoundFor) then exit loop and load the process on the processor labeled MyId.

4. Endfor.

The Matching algorithm determines the conditions which have to be satisfied for a successful match to occur. Two variants are presented. The first, match-exact, requires the process to meet the processor at the exact same processor. In the second, match-star, it suffices for the rendezvous to take place at the same star, not necessarily the same processor. In either case, if a processor becomes idle while there is a process that resides physically on it waiting to be picked up, then that process will be executed locally regardless of its BoundFor field. This latter condition is checked at the beginning of the matching algorithm (in both versions) since local execution is given priority over transferring the process to a distant processor. Note also that adding this condition diminishes the possibility of indefinite postponement significantly, since the probability that a process resides indefinitely on a processor that never becomes idle and none of the other idle processors in the system are able to get close enough to its BoundFor vicinity is very low.

We believe this algorithm would perform most efficiently on machines that have a dedicated message processor (post office) for handling communication on each node. The HP Mayfly, currently under construction, is an example of such an architecture. In this case, generating the random walks would not consume any real computing cycles and processors which are idle in actuality would not offset the machine's utilization measurements by appearing to be busy walking.

The AND-Shipping, OR-Stealing Algorithm

In this algorithm we also combine both the sender and receiver initiated schemes but in a more clear cut fashion. The motivating idea here is driven by the extreme cases where a program is dominantly OR-parallel (such as exhaustive search programs) or dominantly AND-parallel (such as list processing applications). We basically employ totally different strategies for AND and OR processes. For OR processes, a receiver initiated strategy is used, whereas a sender initiated one for AND processes is employed. Therefore, as the

name implies, AND processes are being shipped by the spawning processor, while OR processes wait until they get grabbed (stolen) by one of the idle processors.

Conclusions

Several heuristics for process allocation in OM were presented. The heuristics are designed to work on processes of the AND/OR tree but could be easily adapted for use in other process models. To our knowledge, one of the heuristics, named Rendezvous, is the first to combine both sender initiated and receiver initiated policies. All the heuristics presented are fully distributed, scalable, and architecture independent. They use simple preset strategies that do not involve collecting system load information and thus they should exhibit very low run time overhead. The performance of some of these heuristics is examined in the following chapter.

CHAPTER VII

SIMULATIONS AND PERFORMANCE EVALUATION

The Simulation Environment

Our simulation environment was the sequential implementation of the OM virtual machine with a simulation of multiple threads of control. Simulating multiple threads was implemented on two levels: the virtual machine layer and the target architecture. To simulate multiple OMs running in parallel, we had to create as many OMs as there are processors in the virtual architecture. Each OM has its own kernel, operating system, mailer, and process allocator. The target architecture was simulated by an array of pointers to queues, where each queue represents the queue of processes allocated to that processor. The index of the queue pointer in that processor array is the processor's label (*id*). Simulating the interconnection network amounted to implementing a distance function between any two pairs of processor-labels. In other words, given any pair of processor labels x and y , $d(x, y)$ returns the length of the shortest path in the network between x and y .

For sender-initiated algorithms, the chosen process allocation routine is simply linked to the virtual machine (using a compiler flag) and the Ship-Seed routine simply calls the Spawn procedure of that process allocation algorithm. On the other hand, for receiver initiated algorithms, the load balancing routines are called whenever a processor becomes idle.

Reliability of the Performance Results

It is important at this point to address the question of reliability and accuracy of the results of this study. Our performance metrics can be divided into two categories; those sensitive to the simulation environment (i.e., the implementation issues) and those that

are insensitive to the environment. Clearly, the load and dilation performance metrics are insensitive to the simulation environment and are highly accurate and could be used as true indicators of the figures to be expected from parallel benchmarks.

One point that is usually being overlooked is the interaction between calculating these various performance metrics. For instance, while calculating our load metric is insensitive to the simulation environment, it certainly affects the completion time (speedup) metric since calculating measures such as the average load of the processors in the system throughout the execution requires housekeeping overhead that would certainly offset the completion time measures. In a parallel setting, the overhead might be even higher since a performance monitoring routine that resides on a particular processor (normally it would be the host processor in hypercube-like architectures) which would even create higher message traffic due to communicating the load information to that monitor.

The most important metric that is very sensitive to the simulation environment is the completion time. In essence, we mimicked the virtual time model to implement distributed autonomous clocks in the system [17]. Since we assume that messages never arrive out of order, the notion of roll back did not arise. This virtual time is used in calculating the speedup measurements.

From a statistical point of view, it is important to note that the hypotheses (observations) we make in this chapter and the comments on the graphs are made only with confidence proportional to the size of the sample space, which throughout the simulations was 3 or 4 points. Clearly, if the size of the confidence interval is to be increased (i.e., raise the level of confidence with which the hypotheses are made) larger sampling will be required. These experiments, in addition to running complex application programs, constitute the immediate extensions of this work. Before any of the strategies we proposed is to be adopted in the system, performance evaluation based on real-life applications under a parallel version of OM on an actual parallel machine will have to be conducted.

The Test Suite

Most of the process allocation algorithms described in the previous chapter were implemented in the OM kernel of the sequential C version and were run on Sun SPARCstations and HP 835 workstations. However, because of time constraints and the concurrent projects on parts of the OPAL compiler, not all the heuristics we proposed were actually simulated on real programs. Therefore we will present the performance data we obtained from running two programs - MapColor and EvenPath - using the various heuristics. Simulating the remaining heuristics on actual applications is the subject of future work.

The map coloring program solves the problem of coloring a map of five regions with four colors (see Figure 6). The problem is to assign a color to each region in the map such that no two neighboring regions have the same color. For example, if *D* is colored yellow, then neither *A*, *C*, nor *E* can be colored yellow. The AND/OR tree of the map coloring program is very shallow since it only consists of one huge AND process with eight descendant OR processes. The OPAL code of the program is shown in Figure 7.

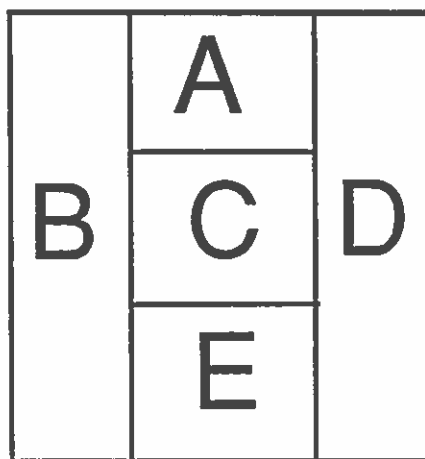


Figure 6: The Map

In the EvenPath program, we are supposed to find even length paths between any two pairs of vertices in an arbitrary graph. The program is very simple and has only two

```
goal <- color(A,B,C,D,E).

color(A,B,C,D,E) <- next(A,B) &
next(A,C) &
next(A,D) &
next(B,E) &
next(C,D) &
next(B,C) &
next(C,E) &
next(D,E).

next(green,yellow).
next(green,red).
next(green,blue).
next(yellow,green).
next(yellow,red).
next(yellow,blue).
next(red,green).
next(red,yellow).
next(red,blue).
next(blue,green).
next(blue,yellow).
next(blue,red).
```

Figure 7: The Map Coloring Program in OPAL

clauses and a set of facts representing the connectivity information. The OPAL code for the program is shown in Figure 8.

In Figure 9, we show the speedup obtained in the map coloring program by running six different load balancing strategies. In all cases we used a bounded maximum dilation of 2. The experiments were run using four different numbers of processors: 2, 4, 8, and 16. We observe that the best speedup was obtained when using the Cai and Batt algorithms with walk length equals 3. The interesting phenomenon was that the version that did not use the distribution among the star centered at the target of the walk was the better one, regardless of the fact that the former had worse average queue length than the algorithm that did the star distribution. Clearly, distributing the load among the star centered at the target is expected to yield better average queue length (load balance) since the distribution chooses the least loaded processor in the star for the allocation.

Another interesting phenomenon happened when using Leighton's algorithm without star distribution. The program ran slower on 8 processors than it ran on 4. This phenomenon is what we described in the introduction as a non-monotonic speedup. However, given our small sample space, it is hard to infer from this single instance any certain bottlenecks or characterizing limitations of the map coloring program.

As mentioned earlier, we used the simplest load assessment policies, which use the average queue length as an indication of load. To calculate the average load we take the average of the averages of the queue lengths of all the processors in the system over the lifetime of the computation. No anomalies were observed in performance according to this metric. All the strategies exhibited an almost linear decrease in the average queue length with the increase in the number of processors (see Figure 10).

To calculate the average dilation, we had to execute the following procedure. On every spawn, we calculate the distance over the interconnection between the processor of the parent process and the target processor of the newly created seed. This distance is the dilation of that communication edge. Note that this scheme works only because all our algorithms prohibit the migration of processes after they are allocated. If a seed could be forwarded after it has been allocated to a given processor, then the dilation of that

```
% epath(?X,?Y) -- there is an even length path from node X
% to node Y in the acyclic graph defined by arc/2.

goal <- epath(X,Y).

epath(A,C) <- arc(A,B) & arc(B,C).
epath(A,D) <- arc(A,B) & arc(B,C) & epath(C,D).

arc(0,1).
arc(0,2).
arc(0,4).

arc(1,2).
arc(1,3).

arc(2,3).
arc(2,4).

arc(3,5).
arc(3,6).

arc(4,5).
arc(4,7).

arc(5,8).
arc(5,10).

arc(6,10).

arc(8,9).

arc(9,10).
```

Figure 8: The Even Path Program in OPAL

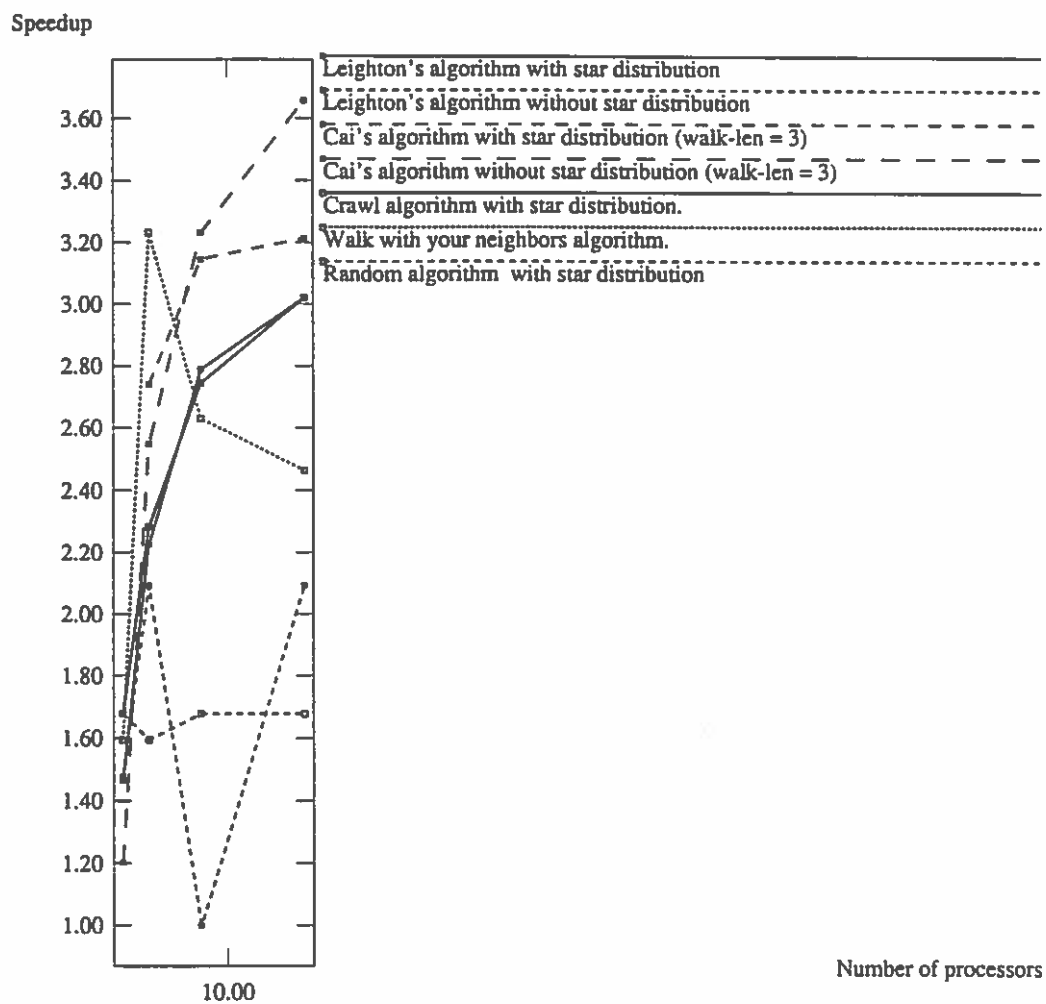


Figure 9: The Speedup in the Map Coloring Program

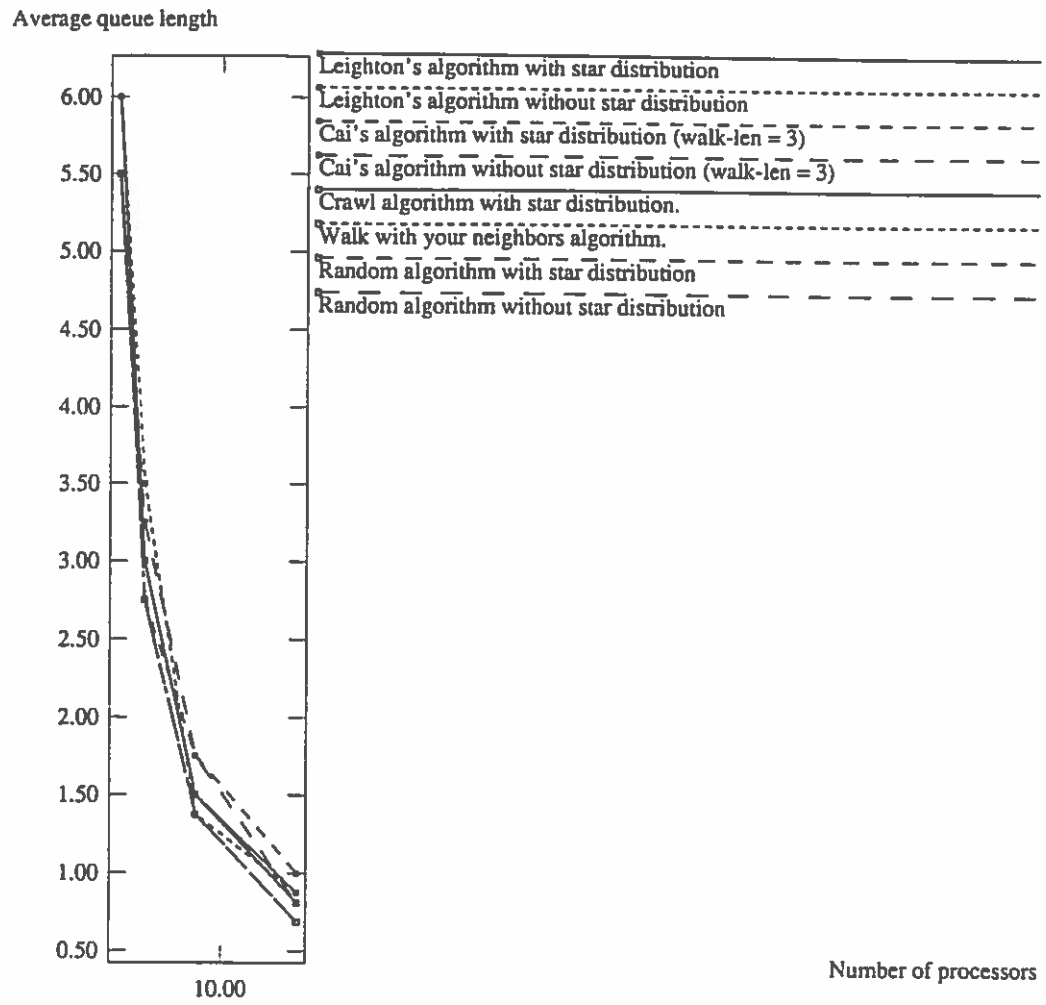


Figure 10: The Average Queue Length in the Map Coloring Program

communication edge should be adjusted accordingly. The average dilation is calculated by taking the average over all the spawns over the lifetime of the program.

In general, it is expected that the average dilation will increase with the number of processors. This is particularly true in randomized algorithms, because the probability is higher that the random walk will take the new process far away from its parent.

From Figure 11, we observe that the above phenomenon occurs only in Leighton's and Cai's algorithms. In Leighton's algorithm, without star distribution, the dilation went to its minimum, 0, at 8 processors. One might guess that all the processes were crammed on one processor. Looking back at the average queue length does not help since the average does not convey any information about the statistical distribution. This is the reason we also collected the maximum queue length statistic. This confirmed our suspicion that all the processes were allocated on one processor, since it was the total number of processes in this program (11). We have no justification why Leighton's algorithm without star distribution would yield such terrible load imbalance. Note that the performance was relatively good with respect to both dilation and load when run using 2 and 16 processors.

Figure 12 shows the speedup obtained in the even path program by using the various load balancing strategies. Linear speedup was obtained using the totally random strategy with star distribution. It also achieved the highest speedup among the strategies at 16 processors. Anomalous behavior was detected in using both Cai's with star distribution and Leighton's without star distribution.

In Figure 13 we show the average queue length obtained in the even path program by using the various load balancing strategies. Similar to the map coloring program, in this case also we obtained very consistent and intuitive performance using the average queue length metric. The average was monotonically linearly decreasing with the increase in the number of processors.

Figure 14 shows the average dilation taken over all the communication edges that take part throughout the execution of the even path program by using the various load balancing strategies shown.

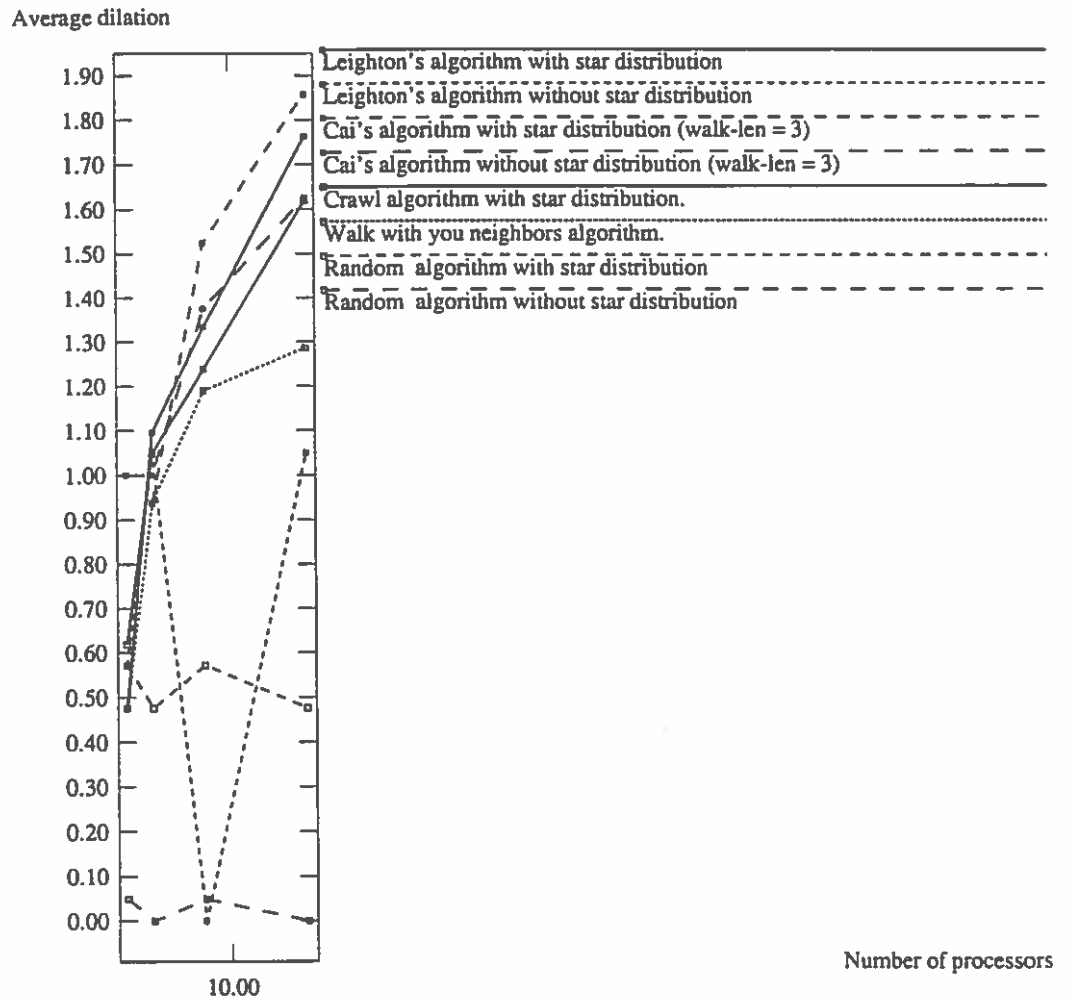


Figure 11: The Average Dilation in the Map Coloring Program

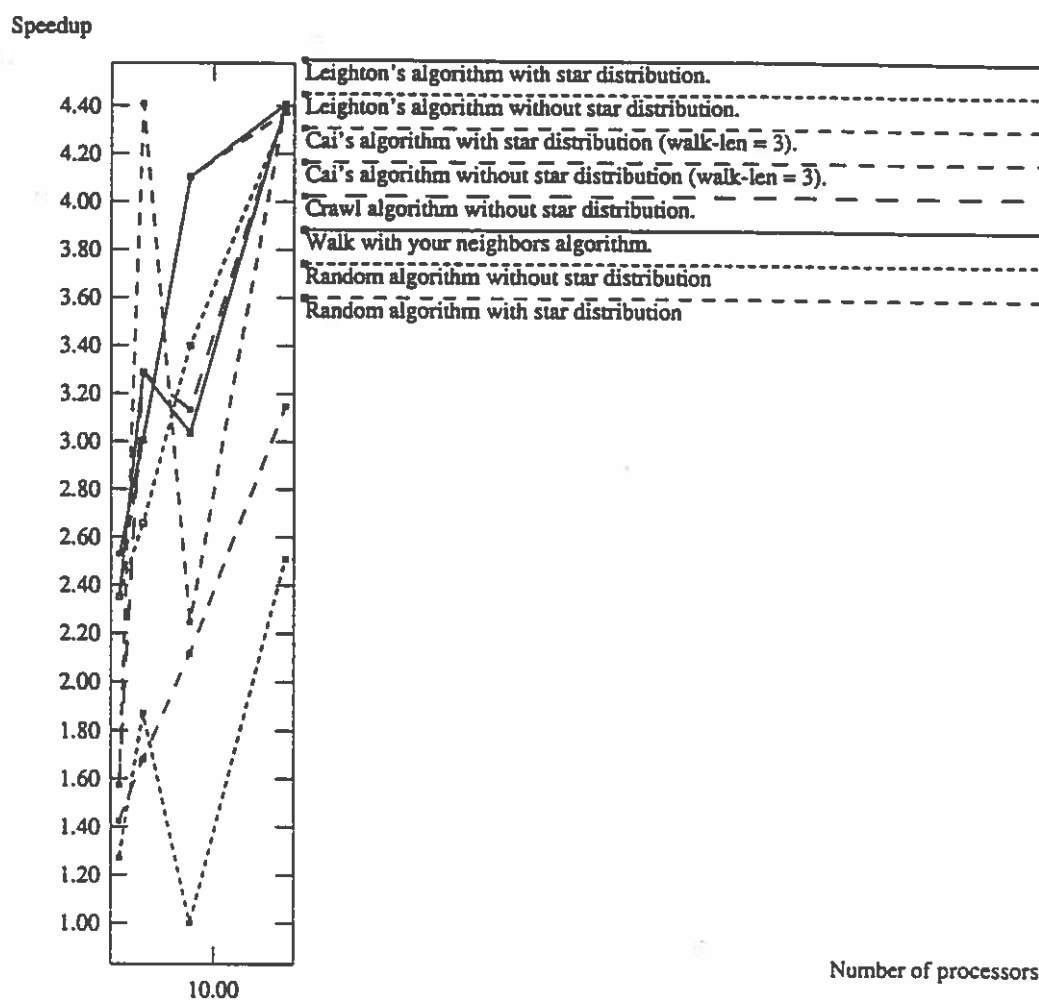


Figure 12: The Speedup in the Path Program

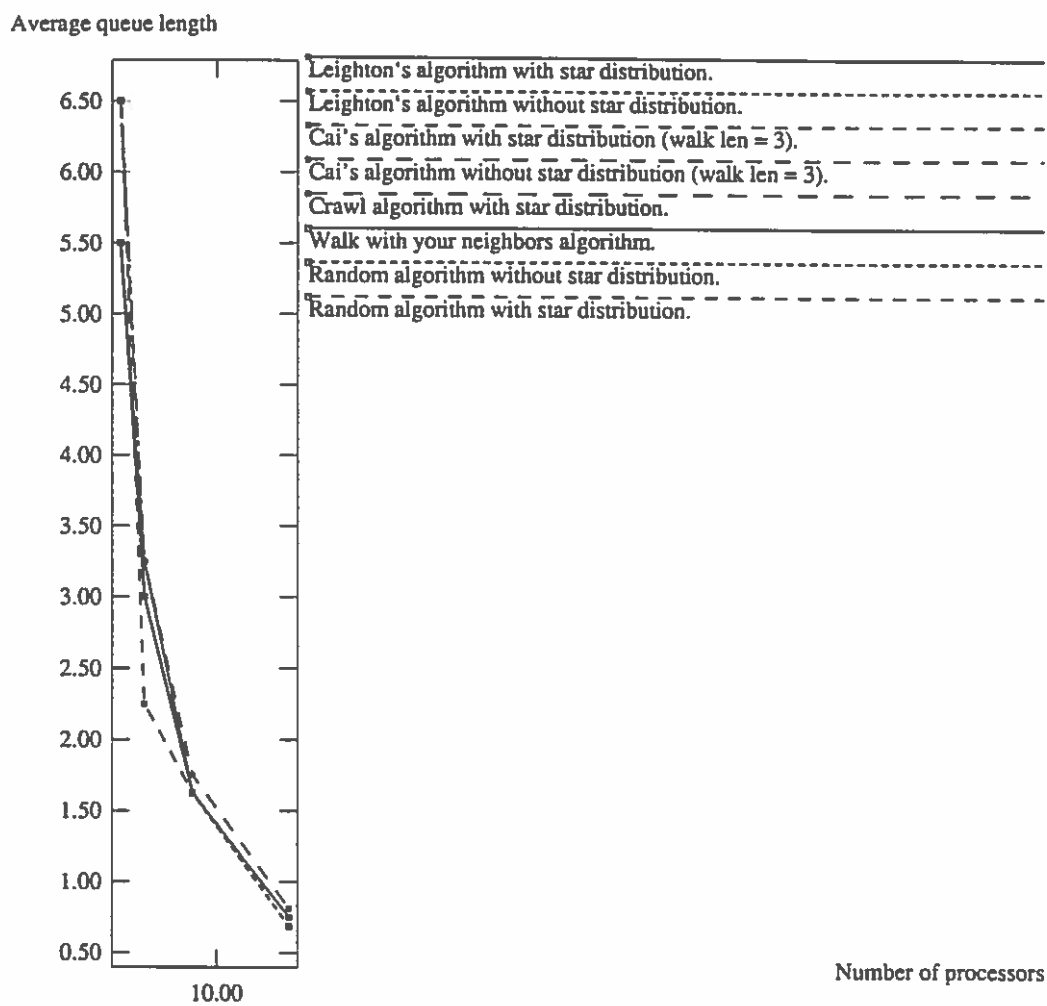


Figure 13: The Average Load in the Path Program

Average dilation

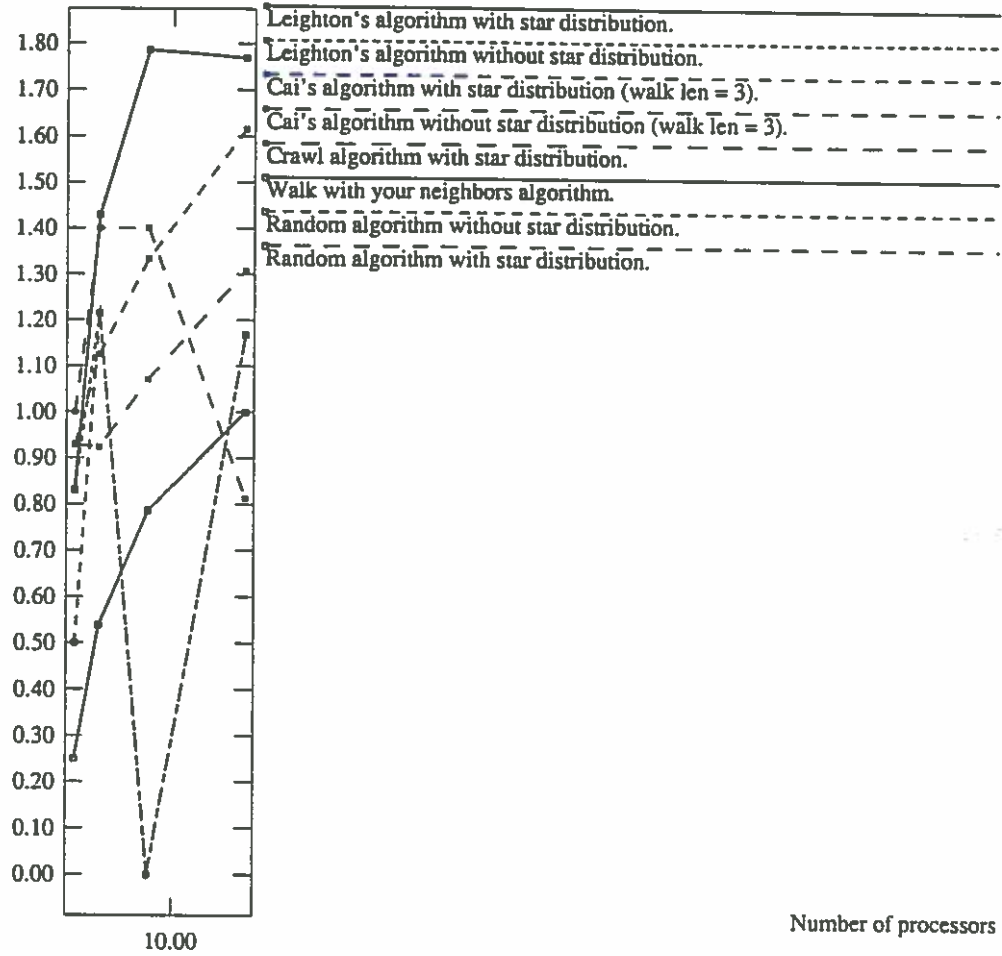


Figure 14: The Average Dilation in the Path Program

CHAPTER VIII

A QUEUING MODEL FOR OM

In this chapter we will study the system from a queuing theory perspective. Queuing theory has been extensively used in modeling distributed computing systems to study the effects of phenomena such as process migration and to analyze various load balancing strategies [11], [10]. However, very little use of queuing models have been reported in the parallel logic programming literature.

The execution of OM on a distributed memory message passing multicomputer can be regarded as a queuing network of C parallel servers where C is the number of processors in the system. Each processor has its own local queue, and the process allocator acts as the arbitration mechanism that distributes the newly generated processes to these parallel queues, which are connected via an interconnection network which is modeled by a distance function (see Figure 15). Such a queuing network can be classified as a closed network [29] when an upper bound on the number of processes in the system is imposed at which the system switches to sequential mode (i.e., no more processes are admitted to the system). In OM, this means that no more processes are created and basically sequential Prolog execution is mimiced on every processor.

The following notation will be adopted [32]. A queuing model $(x/y/z) : (k/l/m)$ is a queuing network where x is the arrival rate distribution, y is the service rate (or inter-departure time) distribution, z is the number of parallel servers, k is the service discipline of the queue (e.g. FCFS, SJF, etc), l is the maximum number of customers allowed in the system, and m is the capacity of the calling source.

It has been universally accepted that the rate of arrivals of customers at a service center has a Poisson distribution, and similarly for the departure rate (i.e., exponential service time). To verify the truth of this phenomenon in OM, we tested that hypothesis by

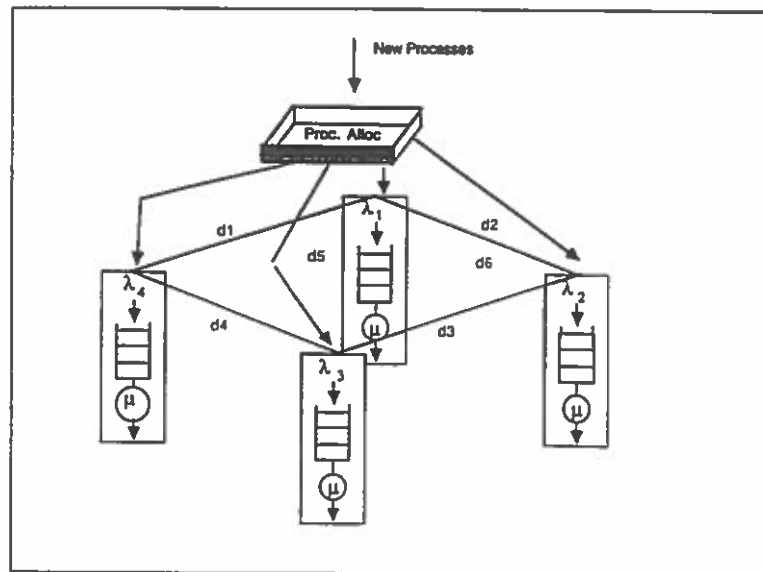


Figure 15: A Queuing View of Process Allocation in OM

process creations (arrivals) and terminations (departures) for the MapColor and EvenPath programs. The Poisson distribution has the unique property that the mean equals the variance equals the average rate of arrivals (λ). This property is conventionally used to test whether a certain random variable obeys the Poisson distribution.

The following experiment was conducted to examine the distribution of the process creation and termination in OM. In the process allocator, a monitor was built that recorded the wall-clock time of each process creation and termination. A time interval of 5 milliseconds (commonly used in timing analysis in UNIX) was chosen to be the sampling interval. By calculating the number of processes created (terminated) in time intervals of 5 seconds, we calculated the mean and variance of the rate of arrivals (departures) distribution. The experiment indicated that, indeed, the rate of arrivals (process creation) and the rate of departures (process termination) in OM obey the Poisson distribution. The use of statistical sophisticated methods to test the hypothesis with a precise confidence interval based on a bigger sample space is the subject of future research. The chi-square test of goodness fit is one of the good candidates for such study.

goodness fit is one of the good candidates for such study. Also, much experimentation with granularity analysis is required to determine a suitable time interval for the various programs.

We concluded that OM can be modeled as $(M/M/C) : (GD/N/\infty)$, where M indicates a Poisson distribution, GD indicates a general service discipline (i.e., unrestricted), N is the bound on the total number of processes allowed in the system, and ∞ indicates that the supply of processes is unbounded. The mean arrival rate will be denoted by λ and that of the departure rate by μ . Note that μ being the service rate implies that $\frac{1}{\mu}$ is the mean of the service time. The equations of the model are usually given in terms of a parameter that combines arrival and departure rates, ρ , where $\rho = \frac{\lambda}{\mu}$. Note that $\frac{\rho}{c}$, which will appear frequently in the equations of the model, is a measure of the average utilization of the system.

This model is analyzed in [32] and the following formulae for the probability of zero customers in the system (P_0) and the average queue length (L_q) were derived:

$$P_0 = \begin{cases} \left[\sum_{n=0}^{c-1} \frac{\rho^n}{n!} + \frac{\rho^c(1-(\rho/c)^{N-c+1})}{c!(1-\rho/c)} \right]^{-1}, & \rho/c \neq 1 \\ \left[\sum_{n=0}^{c-1} \frac{\rho^n}{n!} + \frac{\rho^c}{c!} (N-c+1) \right]^{-1}, & \rho/c = 1 \end{cases} \quad (\text{VIII.1})$$

$$L_q = \begin{cases} P_0 \frac{\rho^{c+1}}{(c-1)!(c-\rho)^2} \left\{ 1 - \left(\frac{\rho}{c}\right)^{N-c} - (N-c) \left(\frac{\rho}{c}\right)^{N-c} \left(1 - \frac{\rho}{c}\right) \right\}, & \rho/c \neq 1 \\ P_0 \frac{\rho^c(N-c)(N-c+1)}{2c!}, & \rho = c \end{cases} \quad (\text{VIII.2})$$

It is worth noting here that the case $\rho \neq c$ can be safely considered the general case since in most cases the utilization will not be equal to one. Furthermore, the number of processors (c) will be fixed for a particular system, whereas the value of ρ will vary from run to run. A three dimensional plot of the average queue length as a function of ρ and N , at $c = 8$ is shown in Figure 16.

In addition to the above equations, we will also need Little's formula for the following analysis. Little derived an important equation that gives the relationship between the

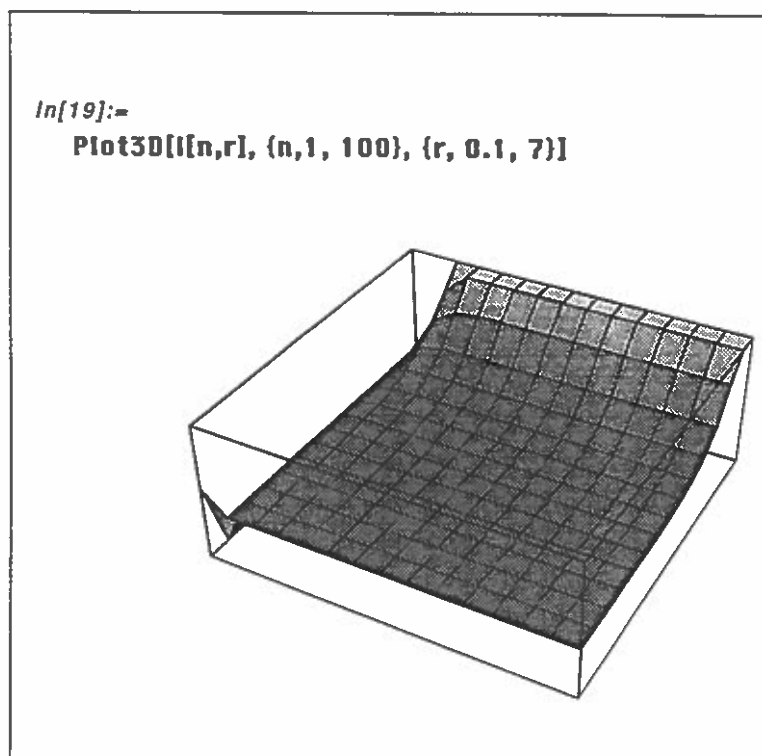


Figure 16: A Plot of the Average Queue Length Function

average (expected) waiting time in the queue W_q , and the expected queue length, L_q [24] as:

$$L_q = \lambda * W_q$$

Bounding the Number of Processes

Our objective is to find the bound on the number of processes (N^*) allowed in the system at which we should switch to sequential mode, i.e., prohibit any more processes from being created. There are several criteria that could be used to dictate the constraints that have to be satisfied at that bound N^* . For instance, one might specify that, if the utilization factor drops below 20%, one should switch to sequential mode. However, this approach of examining only the parallel version of the system in isolation without considering the model for sequential execution did not lead to any useful equations.

For example, the approach that N^* should be the value of N at which the mean waiting time (or the mean queue length, since they are linearly related by Little's formula) is minimum did not lead to any interesting (non-trivial) results as it is clear from the plot of the mean queue length function that it is a non decreasing function in N .

Therefore, we concluded that N^* should be the number at which switching to sequential execution leads to an overall better performance than continuing in a parallel mode. Thus, we believe the following approach is more effective in getting the value of that bound. We calculate the value at which the two models achieve equal performance, and examine the two sides of that boundary to determine the direction in which the sequential mode would be superior. That value is the sought bound.

Note that for the sake of that comparison, the sequential execution can be approximately modeled by C queuing systems of the type $(M/M/1) : (GD/N/\infty)$. The reason this is not a totally accurate model is that it does not account for the fact that there are still multiple queues in one system and they are not totally independent of each other as the above model implies. Investigating this approach and analyzing its limitations is the subject of future research.

We are also interested in feeding this set of equations to multi-dimensional opti-

mization algorithms that do not require differentiability such as Powell's method and the multi-variate grid search [8]. Another alternative is to use constrained search techniques by expressing the desired performance attributes as constraints that have to be satisfied at the solution [8]. For example, one such constraint might be $\frac{\rho}{c} > 0.7$. Stochastic analysis is also being considered in analyzing the system, where modeling the system as a "Filtered Poisson Process" is being studied.

CHAPTER IX

CONCLUSIONS AND FUTURE RESEARCH

We have presented heuristics for process allocation and load balancing in message passing parallel logic programming systems. All the heuristics presented are fully distributed and architecture independent. They also can be classified as simple preset strategies that do not respond to system state fluctuations, and thus have negligible run time overhead. The heuristics do not assume any global shared memory and rely on message passing as the *de facto* style of communicating load information. Thus, they can be implemented on both message passing multicomputers as well as shared memory multiprocessors. The heuristics were implemented and tested in the current sequential implementation of the OM virtual machine using a multiple threads simulation. We have tried to use performance metrics that are insensitive to the simulation environment. Speedup, average load, average dilation, and communication contention were among the most important of these metrics.

This study was an exercise in evaluating several approaches to solving an important problem in the parallel logic programming paradigm, viz, dynamic load balancing. While our original intentions were to conduct a practical engineering-style study, the status of the current implementation of the system and the lack of time to undertake a parallelization effort of the system (before being finalized) obstructed such an approach. Therefore a simulation approach was deemed reasonable as the test bed for the ideas proposed herein. Clearly, the immediate next step in this work is to do the performance evaluation of those algorithms in a realistic parallel setting.

Theoretical analysis based on a queuing model was also presented. An attempt to derive a bound on the number of allowable processes in the system before switching to sequential execution mode was discussed. Conventional techniques were found unsatis-

factory in achieving this bound, and optimization techniques will have to be employed in further investigating this problem. Once a formula for the bound has been obtained, testing it in an actual parallel implementation would be mandatory to verify its impact on performance.

An interesting extension to this work is to incorporate in the process allocator a visualization routine. Since all process creation, termination, seed transfer and similar actions have to go through the process allocator, embedding a visualization routine that would display a visual representation of these actions (such as a tree growing and shrinking) would be a very elegant tool. Such a tool is not only luxury to the programmer, it would indeed help her/him visually observe the spawning and communication patterns in the program. These observations could provide crucial clues into means for improving the efficiency of the execution of the program either by rewriting some of the clauses or procedures, or choosing a different process allocation algorithm. Note that the latter option is actually very simple to implement since several process allocation routines are available and the appropriate one can be linked at run time at the programmer's choice.

BIBLIOGRAPHY

- [1] ATHAS, W. C., AND SEITZ, C. L. Multicomputers: Message-passing concurrent computers. *IEEE Computer* 21, 8 (August 1988), 9-24.
- [2] BATT, S., AND CAI, J. Take a walk, grow a tree. In *29th Annual Symposium on Foundations of Computer Science* (October 1988), pp. 469-478.
- [3] BEAUMONT, A., RAMAN, S., AND SZEREDI, P. Scheduling or-parallelism in aurora with the bristol scheduler. Tech. Rep. TR-90-04, University of Oregon, 1990.
- [4] BERMAN, F. Experience with an automatic solution to the mapping problem. In *The Characteristics of Parallel Algorithms*. The MIT Press, 1987, pp. 307-334.
- [5] CONERY, J. S. *The AND/OR Process Model for Parallel Interpretation of Logic Programs*. PhD thesis, University of California at Irvine, 1983.
- [6] CONERY, J. S. *Parallel Execution of Logic Programs*. Kluwer Academic Publishers, 1987.
- [7] CONERY, J. S., AND MEYER, D. OM: A virtual processor for parallel logic programs. Tech. Rep. CIS-TR-87-01, University of Oregon, February 1987.
- [8] COOPER, L., AND STEINBERG, D. *Introduction to Methods of Optimization*. W. B. Saunders Company, 1970.
- [9] EAGER, D. L., LAZOWSKA, E. D., AND ZAHORJAN, J. Dynamic load sharing in homogeneous distributed systems. Tech. Rep. 85-04-01, University of Washington, CS Department, Seattle, WA., October 1984.
- [10] EAGER, D. L., LAZOWSKA, E. D., AND ZAHORJAN, J. A comparison of receiver-initiated and sender-initiated dynamic load balancing. Tech. Rep. 85-04-01, University of Washington, CS Department, Seattle, WA., April 1985.
- [11] EAGER, D. L., LAZOWSKA, E. D., AND ZAHORJAN, J. Dynamic load sharing in homogeneous distributed systems. *IEEE Transactions on Software Engineering* 12, 5 (May 1986), 662-674.
- [12] EAGER, D. L., LAZOWSKA, E. D., AND ZAHORJAN, J. Speedup versus efficiency in parallel systems. Tech. Rep. 86-08-01, University of Washington, CS Department, Seattle, WA., August 1986.

- [13] FURUICHI, M., TAKI, K., AND ICHIYOSHI, N. A multi-level load balancing scheme for or-parallel exhaustive search programs on the multi-psi. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. (March 1990), pp. 50–59.
- [14] HARIDI, S., AND JANSON, S. Kernel andorra prolog and its computation model. In *Logic Programming: Proceedings of the Seventh International Conference* (1990), MIT Press, pp. 31–46.
- [15] HENNESSY, J., AND PATTERSON, D. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc, 1990.
- [16] HILLIS, W. D. *The Connection Machine*. The MIT Press, 1985.
- [17] JEFFERSON, D. R. Virtual time. *ACM Transactions on Programming Languages and Systems* 7, 3 (July 1985), 404–425.
- [18] KALE, L. V. The reduce or process model for parallel evaluation of logic programs. In *Fourth International Conference on Logic Programming* (May 1987), MIT Press, pp. 616–632.
- [19] KALE, L. V. A comparison between two dynamic load balancing strategies. In *Proceedings of the 1988 ICPP* (August 1988).
- [20] KORRY, R. A load sharing algorithm for a workstation environment. Master's thesis, University of Washington, October 1986.
- [21] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM* 21, 7 (July 1978), 558–565.
- [22] LEIGHTON, T., NEWMAN, M., AND SCHWABE, E. Dynamic embedding of trees in hypercubes with constant dilation and load. manuscript, 1989.
- [23] LIN, F. C. H., AND M., K. R. The gradient model load balancing method. *IEEE Transactions on Software Engineering* SE-13, 1 (January 1987), 32–38.
- [24] LITTLE, J. D. C. A proof of the queuing formula $l = \lambda w$. *Operations Research* 9 (May 1961), 383–387.
- [25] LO, V. M., RAJOPADHYE, S., GUPTA, S., KELDSEN, D., MOHAMED, M. A., AND TELLE, J. OREGAMI: software tools for mapping parallel algorithms onto parallel architectures. In *Proceedings of the 1990 ICPP* (August 1990), p. to appear.
- [26] LO, V. M., RAJOPADHYE, S., MOHAMED, M. A., NITZBERG, B., GUPTA, S., KELDSEN, D., ZHONG, X., AND TELLE, J. LaRCS: A language for describing parallel computations. Submitted to the *IEEE Transactions on Parallel and Distributed Systems*.

- [27] LUSK, E., WARREN, D. H. D., AND HARIDI, S. The aurora or-parallel prolog system. *New Generation Computing* 7, 2 (March 1990), 243-271.
- [28] MAGEE, J., KRAMER, J., AND SLOMAN, M. Constructing distributed systems in conic. *IEEE Transactions on Software Engineering SE-15*, 6 (June 1989), 663-675.
- [29] ONVURAL, R. O. Survey of closed queueing networks with blocking. *ACM Computing Surveys* 22, 2 (June 1990), 83-121.
- [30] SHU, W. W., AND KALE, L. A dynamic scheduling strategy for the chore-kernel system. In *Supercomputing '89* (November 1989), pp. 389-398.
- [31] SUGIE, M., YONEYAMA, M., IDO, N., AND TARUI, T. Load-dispatching strategy on parallel inference machine. In *Proceedings of the International Conference on Fifth Generation Computer Systems* (August 1988), pp. 987-993.
- [32] TAHA, H. A. *Operations Research, an Introduction*. MacMillan Publishing Co., 1985.
- [33] TAKEDA, Y., NAKASHIMA, H., MASUDA, K., CHIKAYAMA, T., AND TAKI, K. A load balancing mechanism for large scale multiprocessor systems and its implementation. In *Proceedings of the International Conference on Fifth Generation Computer Systems* (August 1988), pp. 978-986.