INFORMATION TO USERS

This was produced from a copy of a document sent to us for microfilming. While the most advanced technological means to photograph and reproduce this document have been used, the quality is heavily dependent upon the quality of the material submitted.

The following explanation of techniques is provided to help you understand markings or notations which may appear on this reproduction.

1. The sign or "target" for pages apparently lacking from the document photographed is "Missing Page(s)". If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting through an image and duplicating adjacent pages to assure you of complete continuity.

2. When an image on the film is obliterated with a round black mark it is an indication that the film inspector noticed either blurred copy because of movement during exposure, or duplicate copy. Unless we meant to delete copyrighted materials that should not have been filmed, you will find a good image of the page in the adjacent frame.

3. When a map, drawing or chart, etc., is part of the material being photographed the photographer has followed a definite method in "sectioning" the material. It is customary to begin filming at the upper left hand corner of a large sheet and to continue from left to right in equal sections with small overlaps. If necessary, sectioning is continued again—beginning below the first row and continuing on until complete.

4. For any illustrations that cannot be reproduced satisfactorily by xerography, photographic prints can be purchased at additional cost and tipped into your xerographic copy. Requests can be made to our Dissertations Customer Services Department.

5. Some pages in any document may have indistinct print. In all cases we have filmed the best available copy.

University
Microfilms
International

300 N. ZEEB ROAD, ANN ARBOR, MI 48106
18 REDFORD ROAD, LONDON W13 8HA, ENGLAND
7919632

SINCLAIR, JAMES BARTLETT
DYNAMIC REASSIGNMENT IN DISTRIBUTED PROCESSING SYSTEMS.

RICE UNIVERSITY, PH.D., 1979
RICE UNIVERSITY

DYNAMIC REASSIGNMENT IN DISTRIBUTED PROCESSING SYSTEMS

by

JAMES B. SINCLAIR

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

THESIS DIRECTOR'S SIGNATURE

[Signature]

HOUSTON, TEXAS

AUGUST, 1978
ACKNOWLEDGEMENTS

I would like to express my sincere appreciation to my thesis advisor Dr. G.S. Rao for his direction, encouragement, and faith. I would also like to thank him for his friendship over the past three years.

Dr. D.S. Hirschberg has rendered invaluable aid on several occasions, and I wish to acknowledge his contributions and thank him for his help.

I want to also thank Dr. J.R. Jump and Dr. R.A. Tapia for serving on my thesis committee, and Dr. E.A. Feustel for providing support during much of the preliminary phases of my research.

My fellow students and friends at Rice have endured much over the course of preparing this thesis. I would especially like to thank Anand Jagannathan, Timothy Gonsalves, and Daniel Dias for their interest, their comments, and their endurance.

Two persons who have made my years at Rice far more enjoyable and easier than they might otherwise have been are Mrs. Velma Goodwin and Mrs. Ruth Collins. Rice is fortunate to have them.

Debbie Howe, who has done an outstanding job under less than ideal circumstances in typing the final draft, deserves a special thanks for her efforts.

Many other persons at Rice, both faculty and students, have helped to make my graduate as well as undergraduate experience at
Rice rich and rewarding, and I would like to thank them all for kindnesses large and small.

Lastly and most importantly, my wife Diane deserves much of the credit for any success I may have had because of her understanding and support for these three years. Her perservance and supererogatory help during the actual preparation of this thesis have been beyond thanks. To her I dedicate this thesis.

This research has been supported under NSF grants MCS72-03609 A01 and MCS77-04364.
# TABLE OF CONTENTS

## CHAPTER 1 \ INTRODUCTION

1.1. Distributed processing

1.2. Outline of thesis

## CHAPTER 2 \ DISTRIBUTED PROCESSING SYSTEMS AND DISTRIBUTED PROGRAM EXECUTION

2.1. Introduction 6

2.2. Distributed data processing systems 6

2.3. Program model 12

2.4. Program costs 15

2.4.1. Execution costs 16

2.4.2. Communication costs 16

2.4.3. Reassignment costs 18

2.4.4. Cost interpretation 19

2.5. Previous work 20

2.5.1. Static assignment 20

2.5.2. Critical load factors 22

2.5.3. Limited memory scheduling 23

2.5.4. Dynamic assignment 25

2.6. Thesis direction 26

## CHAPTER 3 \ DYNAMIC ASSIGNMENT WITH MAX-FLOW MIN-CUT ALGORITHMS

3.1. Introduction 28

3.2. Restricted DPS model 28

3.3. Commodity flow graph 30
3.4.  Optimal dynamic assignment using commodity flow networks 36
3.4.1.  Introduction 36
3.4.2.  Dynamic module interconnection graph 37
3.4.3.  Dynamic processor flow graph 40
3.4.4.  Optimal dynamic assignment using dynamic processor flow graphs 41
3.5.  Limitations of the flow algorithm 50
3.6.  Summary of dynamic processor flow algorithm 51

CHAPTER 4  AN APPLICATION OF DYNAMIC PROGRAMMING TO THE OPTIMAL DYNAMIC ASSIGNMENT PROBLEM

4.1.  Motivation for considering dynamic programming 53
4.2.  The dynamic programming algorithm 55
4.2.1.  Relation to shortest path algorithm 55
4.2.2.  Description of algorithm 56
4.2.3.  Time and space complexity 63
4.3.  Expected space complexity 68
4.3.1.  Pruning 68
4.3.2.  Derivation of pruning probability recurrence relations 69
4.3.2.1.  Assumptions 69
4.3.2.2.  Dynamic pruning 73
4.3.2.3.  Quantitative analysis of pruning 73
4.3.3.  Numeric results 82
4.3.4.  Summary of expected space complexity 89
4.4.  Summary of dynamic programming algorithm 90
CHAPTER 1

INTRODUCTION

1.1. Distributed processing

The concept of distributed data processing systems is a fairly recent one, and yet the current level of interest in the topic belies its maturity. Proponents of such systems argue for the desirability of distributed data processing systems from three basic premises: increased performance, system extensibility, and reliability enhancement. Under the heading of increased performance come the promises of fast response time and system availability, resource sharing and load balancing, and the potential for incorporation of very high speed, special function processors not normally economically feasible for single users. The extensibility considerations include the potential for modular system growth, ease of functional adaptation and/or expansion, and the ability to incrementally replace or upgrade system components. Reliability involves such issues as graceful degradation of performance with component failure, replication of resources, and the automatic monitoring of system integrity (ENS78) (JEN78) (AGR76) (ECK78).

Unfortunately, as has often been pointed out (ENS78), most of the imputed benefits of distributed processing systems
have yet to be realized. As an example, the most frequently cited criterion for describing a computer network as a distributed data processing system has been the existence of a decentralized executive which controls and unifies the aggregate resources of the network. Furthermore, the reliability issues make it imperative that there be no binding between the distributed operating system and specific resources on a permanent basis. However, although the necessity of such system-wide distributed executive control has been accepted, the general problem of how we might implement such an operating system seems to have produced little in the way of practical results. Perhaps the best known and most successful system in this respect is the communications subnetwork of the ARPANET; however, its limited functional application makes it difficult to generalize to more diverse systems.

This thesis concerns itself with one facet of the problem of system-wide control of the resources of a distributed processing system, that of computational resource allocation. A program which is to be executed in a distributed system should not be constrained to be executed on a single processing element of the system, but should be allowed to have its execution distributed across the system to take advantage of capability diversifications within the system. In doing so, the program must pay additional overhead in the form of increased interprocessor communications and program relocation costs. We wish to be able
to determine in a reasonably efficient manner the "best" way to distribute the program's execution, taking into consideration all three factors: program execution requirements vs processor capabilities, interprocessor communications overhead, and the cost of transferring the locus of program execution from one processor to another. The latter becomes especially significant if we allow specific parts of a program to "migrate" or dynamically change their processor residences while the program is being executed. That is, it may be advantageous to allow a portion of the program which is repeatedly executed during a single execution of the program to be executed on different processors at successive iterations.

1.2. Outline of thesis

In considering this problem, we present the remainder of the thesis in five parts. Chapter 2 attempts a general description of distributed processing systems, or at least those aspects which are relevant to the subsequent discussions. A model of a program to be executed in a distributed environment is given, together with a discussion of the cost factors which will be considered in determining how the program is to be distributed to best take advantage of available resources. Chapter 2 also includes a brief survey of previous research in this and related areas.

Chapter 3 presents one method of finding the optimum program execution distribution, through the use of commodity
flow networks. The method is quite elegant and efficient, but will be shown to lack sufficient generality. An attempt to avoid these deficiencies in generality leads us to consider a totally different approach, which we describe in Chapter 4. This approach is through the use of a dynamic programming algorithm. Although it is successful in overcoming the inadequacies of the commodity flow algorithm, it pays a high price in computational efficiency and storage requirements. It is shown, however, that the average space complexity for this algorithm is much less than worst case.

The inefficiency of the dynamic programming algorithm leads us to consider what improvements might be possible if the program were to exhibit some degree of predictability in its behavior. A program to be executed in a distributed processing system will be divided into a fixed number of disjoint parts or modules, representing the indivisible atoms of the program as far as distribution is concerned. We consider the possibility of the sequence of module executions exhibiting behavior similar in aspect to that which has been observed in the sequence of page references in a paged memory system (the working set concept). By this we mean that the sequence of module executions should show clustering of modules in time, and the clustering should change in an evolutionary manner during program execution. This forms the basis for the discussion in Chapter 5, where we show that if a program does have this characteristic, a modification
of the dynamic programming algorithm can lead to dramatic improvements in the expected time required for large problems. Chapter 6 summarizes the results of the preceding four sections and indicates potentially fruitful areas for further investigation.
CHAPTER 2
DISTRIBUTED PROCESSING SYSTEMS
AND DISTRIBUTED PROGRAM EXECUTION

2.1. Introduction

In this section we give a more precise definition to what is meant by a distributed processing system, especially in those areas which are most pertinent to the idea of distributed program execution. We then proceed to describe a model for a program which is to be executed in the distributed system. This model must incorporate those aspects of program behavior which will have significant impact on the determination of optimal program distribution. A number of recent publications have dealt with related problems in the area of distributed processing and distribution of program results. Several of these works are described briefly.

2.2. Distributed data processing systems

Any discussion of distributed data processing systems is immediately hampered by the lack of any general agreement on what constitutes such a system. The confusion resulting from the absence of a consistent definition of distributed processing is quite apparent. The recent workshops at Brown University in 1976 and 1977 brought this issue sharply into focus. A working definition of distributed processing systems has been proposed by Enslow (ENS78), and consists of five parts.
To be a distributed data processing system, the system must have

1) a multiplicity of general-purpose assignable
   (i.e., not fixed function) resources, not necessarily
   homogeneous,

2) physical distribution of resources which interact
   through a communications subnetwork,

3) high level executive control for the integration of
   individual components into a single system,

4) system transparency, in that services may be requested
   by name only, without regard to or knowledge of the
   resources which will provide that service, and

5) cooperative autonomy among all physical and logical
   resources.

For the purposes of this thesis, the first four points of the above definition are sufficient and necessary to characterize a distributed processing system. A DPS (distributed processing system) must have a number of processing components, each capable of providing a general purpose computational service on a dynamic basis. These processing elements may (and in general will) have widely diversified capabilities, not only in terms of computing facility but also in terms of local resources providing similar services. For instance, two processors may each have secondary store, but for one this might be bulk memory while the other may have a relatively slower sequential access device such as a disk or even magnetic tape.
An example of a system having two processors with quite different capabilities is the ICOPS distributed system at Brown University (FOL76). The ICOPS system is illustrated in Fig. 2.1. Note that there are actually 3 processors; however, the second Meta 4 processor merely serves as a driver for the Vector General displays and hence has no assignable function. Because of the fixed functionality, the second Meta 4 is not relevant in the categorization of ICOPS as a DPS.

Other systems include a number of identical processors. Among these are Honeywell's HXDP (JEN78), the DCS at the University of California at Irvine (FAR72), the ARPA network (KLE76), and the DCN of the University of Maryland (AGR76). Some of these systems incorporate sets of identical processors along with other dissimilar processors.

The second requirement that a DPS must fulfill is a geographical distribution of resources, which must interact through a communications network. It is assumed that the physical dispersion of components is sufficient to preclude memory access-speed communications between processors, so that not only do the disjoint primary memory address spaces mean much slower data transfer rates, but also a higher degree of interprocessor latency than among tightly coupled multiprocessors. It is this restriction which prohibits systems such as C.mmp (WUL72) and C.m* (FOR78) from being considered as DPS's.
Figure 2.1.

Brown University ICOPS, a distributed processing system for interactive graphics.
The ICOPS systems has its two processors connected via a 50 kbps link. The DCN is much more loosely coupled, with communication between processors typically at 1200 baud.

The actual communication media, network topology, message protocol, and other implementation features of the communication subnetwork are irrelevant in this discussion. Obviously these and other factors will influence average communication rates because of switching delays, message queuing, protocol overhead, and network capacity limitations, but we choose to ignore these effects or consider them implicitly incorporated in the system model.

The integration of physical and logical elements of the system by a single high-level distributed executive control is often acknowledged to be the major distinguishing characteristic of a DPS (ECK78). This must be accomplished in the absence of global, centralized state information. Each processor within the system will have its own local operating system, but the kernel of such a local system should be minimized to avoid conflict or contention with the policies of the system-wide executive.

Many authors require that in a DPS the control exercised by the overlying operating system be dynamic in that there should be no permanent binding of control processes or global state information to a single resource. The dynamic distribution of executive control is certainly one of the most desirable
characteristics of a truly distributed data processing system for the obvious impact that it will have on the issues of reliability, integrity, and what Jensen calls "extensibility" (JEN78). However, these issues per se are not of concern in this paper, and therefore we also admit systems in which the loci of control are in fact totally or partially fixed. It should be emphasized that we do not intend to support such a system, but the state of development in distributed DPS executive control leads us to the conclusion that many of the problems in implementing the operating system in general have not been solved and hence we will be faced with more restricted systems for some time.

The fourth, and for our purposes the most important, point in the characterization of a DPS, is intimately related to the concept of an overall executive control (as indeed are the first two points). The DPS should be completely transparent to the user. The DPS appears to the user as a single virtual uniprocessor. The virtualization may include resources from any of the components within the system and may in fact be dynamic during runtime. The user interacts with the system on a network level and requests services by name only. The location of the resources which will provide those services is unknown and may change as the system workload changes with time.

It is with the issue of dynamic resource allocation in a distributed environment that this thesis is concerned.
Generally, the single most important service provided by any computer system is computational power. We wish to be able to manage a user's request for execution of his program by automatically allocating those computational resources best suited to his requirements. The resources provided a single user can and should change with time as the user's requirements change and system load varies. The techniques for determining an optimal resource allocation which are presented in chapters 3, 4, and 5 are applicable to a system which at a minimum satisfies the four points mentioned above.

2.3. Program model

Because we shall be concerned with the distribution of programs in a DPS or, equivalently, the allocation of the DPS computational resources to user tasks, we need to have a fairly precise idea of what the structure of such a program should be. This in turn will necessarily reflect the structure of a DPS as defined in section 2.1. The actual computation performed by a program is of no interest in this context. Specifically we must deal with three aspects of a program:

1) program partitioning,
2) the flow of control among the various part of the program, and
3) the nature of the interactions among the parts of the program.
A program will be assumed consist of a set of modules, \( \{A, B, \ldots, N\} \). Each module may be executed many times during a single execution of the program, but the execution of modules is strictly sequential; that is, we allow no module to be executed concurrent with the execution of any other module of the program. Each execution of a module will be called an instance of that module. The program partitioning is done on an ad hoc although hopefully logical (i.e., structured) basis by the user. Allowing the operating system or a compiler to automatically partition a program to make its distribution more effective or advantageous is an open topic and is beyond the scope of this research.

The most important facet of the program model is the nature of the interactions between modules of the program. Any instance of any module of the program will be able to communicate only with the module instances immediately preceding it and succeeding it in the program execution. No global, shared variables are allowed. Intermodule communication will be solely through the means of a CALL-like mechanism with associated parameter passing. Parameter passing is by value only. These restrictions are for a number of reasons. First, the overhead due to latency in interprocessor communications makes it more desirable to minimize the frequency with which modules which potentially may reside on different processors must communicate. Secondly, the difficulty in attempting to
control processes in a distributed environment mitigates against extensive intermodule communication. Enslow (ENS78) strongly makes the point that interprocess communications should employ a message-type protocol, just as interprocessor communications, and that all information shared between two processes must be passed over well-defined and rigidly enforced interfaces. Eckhouse (ECK78) points out many of the problems in interprocess communication, and suggests that in a nonhomogeneous distributed environment, interprocess communications must be simplified.

A third reason for imposing these restrictions on a program which is to be executed in a distributed system is due to the fact that a DPS, because of the multiplicity of resources, should be multiprogrammed to maximize resource utilization. By this we mean that the DPS should support a number of virtual processors to enable users to obtain simultaneous access to system services. Not only does this mean that different processors may be servicing different users at the same time, but also individual processors may themselves be multiprogrammed. In a rapidly fluctuating environment of this nature, a module which wishes to reference a global variable bound to another module may discover that there will be additional overhead in accessing that module if it is currently residing on a multiprogrammed machine but is not active, thus degrading system performance as well.
These restrictions are not as severe from the programmer's point of view as one might think. The tenets of structured software development support the construction of a program as a number of modules, each having a single entry point and a single exit point and communicating with each other via the transfer of control (i.e., calls and returns) with explicit parameter passing. For large programs this modularization of a program into totally disjoint subprograms is necessary to make the program understandable, verifiable, and maintainable. Nor is this idea new in DPS's. One distributed processing system in existence today, the ICOPS at Brown University, allows interprocess communication only through the mechanism of CALL's with associated parameter passing (FOL76).

Programs which we will consider for execution in a DPS environment, then, should satisfy three requirements. They should be constructed modularly, the modules should be executed sequentially (with repetition and in any order), and intermodule communication should be limited to a CALL-like interface with parameter passing.

2.4. Program costs

The purpose of this research is to determine how one may optimally distribute such a program on a dynamic basis without the necessity of determining the relative merits of all possible different program distributions. We do this by assigning costs to the program based on how it might be distributed, and then
selecting that distribution which minimizes the cost. Costs fall into three types: module execution costs, intermodule communication costs, and module reassignment costs. Each category will be considered separately.

2.4.1. Execution costs

Each instance of every module of the program will have associated with it a number of execution costs, one execution cost for each processor in the system. Each such cost will be the cost of executing that module instance on the associated processor. In general, one instance of a module will have a different execution cost for each processor, because of different capabilities provided by the processors. Also, different instances of the same module may have unequal costs on the same processor, because each instance of a module may not perform precisely the same actions as other instances. Since different modules will have dissimilar functions, the diversification in processor capabilities may affect the execution costs of two modules in opposite ways. For instance, module A may have a lower execution cost on processor 1 than on processor 2, while the reverse might be true for module B.

2.4.2. Communication costs

In a tightly coupled multiprocessor system in which all processors have overlapping primary memory address space, communication between processes executed on different processors
may take place through the shared memory, with consequently little or no overhead and at memory access speeds. In a DPS, however, processes which are residing at different sites within the system must interact over the communications sub-network, thus incurring the overhead of setting up a communications link, transforming and transferring the message according to the necessary protocols, and possibly translating the message if the source and destination processors have different internal structures (such as character sets, word lengths, floating point formats, etc.). Also, the message transferral rate will be limited by the capacity of the link (either physical or virtual) between the two processors; the capacity is assumed to be far less than primary memory bandwidth.

As a result, modules on different processors in a DPS may incur a significant cost if they communicate. (Modules on the same processor will be assumed to have negligible communication cost). The cost may be symmetric with respect to the source and destination processors in that if the source and destination were reversed, the cost would remain the same, or it may be asymmetric. Even if the cost were symmetric, it might be nonuniform throughout the system. By this we mean that the cost might well be a function of the particular processors involved in the exchange, rather than simply a function of the amount of information transferred. A good example of where this might be the case is in the ring-structured DCS (FAR72).
Since the ring is unidirectional, the latency between two processors is related to their positions on the ring and the direction in which information in the ring circulates.

2.4.3. Reassignment costs

In a DPS in which modules of a program are allowed to move from one processor to another during execution, a third type of cost, associated with this movement, may be incurred. When a module instance is executed on a particular processor, we say that the instance is assigned to that processor. If successive instances of the same module are executed on different processors, the module is said to be reassigned, and there will be a cost for reassigning the module.

What happens when we reassign a module? It would be very undesirable for a module reassignment to entail the physical transfer of the code of the module from one processor to another. Not only would this create additional overhead and increase congestion on the communications subnetwork, but in a system with heterogeneous processors, the code as well as data would have to undergo a translation process to make it compatible with its new host. It seems more reasonable to require that every processor which is capable of having a module assigned to it maintain a copy of that module, either in its primary memory or more likely in secondary store. Reassigning a module then becomes a matter of activating a copy of the module on the new processor, communicating the necessary state
information from the previously active copy, and possibly translating the state information. As is the case with the communications costs, the reassignment cost may be symmetric or asymmetric, uniform or nonuniform.

2.4.4. Cost interpretation

Thus far we have identified three costs which must be taken into account in attempting to determine an optimal dynamic assignment for all of the module instances in a program. However, we have said nothing about how these costs are determined or what we are actually trying to optimize. The assignment of costs will depend upon the criterion for optimization which the user wishes to employ. Two such criteria have been suggested by Stone (ST077). One may wish to minimize the real time (the turn-around time) required for program execution. Costs are assigned on the basis of the actual time required by each of the possible actions involved in executing the program for any of the possible assignments of the module instances in the execution of the program. Alternately, one might assign dollar costs to each of the three cost categories and attempt to minimize the overall dollar cost of executing the program. For instance, the execution cost of a module instance would be based on the cost of CPU time, for each of the processors in the DPS.

How do we determine costs? One possible method is to have the programmer supply all necessary information, but this
implies a degree of knowledge which is in all likelihood impossible. A more feasible method would be to monitor previous executions of the program and use past behavior as a basis for estimating future program costs. In the remainder of this paper we will adopt the method of omniscience and assume that we know precisely all pertinent aspects of program execution.

2.5. Previous work

This section presents an overview of recent research and significant results in the area of optimal assignments in DPS's. We can see four basic topics that have been examined:

1) static assignments of modules,
2) the influence of processor multiprogramming on optimal static assignments,
3) limited memory effects on optimal static assignments, and
4) dynamic assignments of modules.

Each of these topics is discussed briefly below.

2.5.1. Static assignment

Stone (ST077) considers the problem of determining an optimal assignment for a program in which each module is constrained to be assigned to a single processor for the entire program execution. Reassignment costs are therefore eliminated. Attention is focussed on two processor systems with symmetric communications. Under these conditions the optimal assignment can be computed very efficiently. Stone uses the aggregate module execution costs and intermodule communications costs to
to construct a processor flow graph. The processor flow graph contains a node for each module, a node for each processor, and an edge for every module node-processor node pair. There is also an edge for each module node-module node pair if two instances of the corresponding modules communicate during the program lifetime. The edges are weighted appropriately with execution and communication costs, and the processor flow graph is then interpreted as a commodity flow network. By employing a max-flow min-cut algorithm, one may determine the optimum static assignment for the program in a very efficient manner. (See Chapter 3 for explanations of commodity flow networks and max-flow and min-cut algorithms).

Although it is possible to construct a processor flow graph for systems which contain more than two processors, such a graph can no longer be interpreted as a simple commodity flow network, and consequently it becomes more difficult to make efficient use of the graph as the number of processors increase. Stone (ST077a) presents an algorithm for the case of three processors. He determines the optimal assignment by finding the minimum cost tricutset partition of the three-processor flow graph using a minimum of two and a maximum of three applications of a max-flow min-cut algorithm. The three-processor algorithm may not find the optimal solution, in which case it gives a solution together with a bound on how far from optimum the solution may be. Nonoptimal solutions obtained
from this method appear to be rare. However, the method does not seem to generalize to more than three processors. It is doubtful that there exists an efficient algorithm for solving the optimal static assignment problem for more than three processors since this problem has been shown to be NP-complete (BOK77a).

Another method of determining the optimal assignment which is not limited in the number of processors is one which involves the use of shortest path algorithms (BOK77a). Its usefulness is limited to those programs whose intermodule communications structure is tree-like. By this we mean that any module (except the "root" module) is called by exactly one other module, and may in turn call any number of other modules. Under this constraint, the optimal static assignment may be computed efficiently by constructing an assignment graph and using Dijkstra's method to find as many shortest paths as there are leaves in the communications structure tree. It is also a simple task to incorporate asymmetric communications costs using this method.

2.5.2. Critical load factors

In the optimal static assignment problem considered above, we implicitly assumed that all parameters within the DPS were fixed. This is certainly unrealistic. In particular, since processors within the DPS may be multiprogrammed, we should ask ourselves how the degree of multiprogramming of a
processor will affect the optimal assignment. Stone (ST077b) considered this problem for the case of two processors, only one of which is multiprogrammed. It should be undesirable to have to run a max-flow min-out algorithm every time the load on the multiprogrammed processor changed. Stone showed that this is not necessary. Let $f$ be the fraction of CPU time that a module would receive if executed on processor $P_1$, the multiprogrammed processor. Then $f$ varies from 0 to 1 as module execution speed (in system time) goes from zero to maximum. Stone showed that every module $M$ which may be assigned to either processor has a critical load factor $f_M^*$ such that if $f$ is less than $f_M^*$, then $M$ will be assigned to $P_2$ (the second processor), but if $f$ is greater than $f_M^*$, $M$ will be assigned to $P_1$ under the optimal assignment. The load factors can be efficiently computed prior to program execution (MIC77) and then the load on $P_1$ can be monitored to allow automatic migration of modules as $f$ varies.

2.5.3. Limited memory scheduling

In the Brown University system described earlier, a relatively small minicomputer is connected to an IBM 360/67. The smaller storage capacity of the minicomputer suggests another parameter of a DPS which may be of importance in determining how a program's execution is to be distributed among the processors. In Stone's initial paper (ST077) it was assumed that each processor had more than enough memory to hold the
modules assigned to it. In a system in which one processor has limited memory, the modules assigned to a processor by the optimal static assignment may not fit in the memory of that processor. Rao, et al. (RA077) considered this problem for a two processor system in which one processor has virtually infinite memory capacity and the other has a limited memory size. A feasible assignment of modules for this system is one in which the combined sizes of all the modules assigned to the limited memory processor is less than or equal to the memory capacity of that processor. Finding an optimal feasible assignment for even this simplest instance of the limited memory problem can be shown to be NP-complete, indicating that it is likely that no algorithm exists for solving the problem which is in worst case better than exhaustive enumeration of all possible assignments.

The principle result of Rao's work is the construction of an inclusive Cuts Graph (ICG) which has three properties:

(i) all feasible assignments can be represented in the ICG;

(ii) there is a partial ordering imposed on the nodes of the ICG which can simplify the process of finding all feasible assignments; and

(iii) the ICG may be significantly smaller than the associated processor flow graph due to node condensation.
By searching for the optimal feasible assignment in the ICG rather than in the processor flow graph, the amount of work required may be greatly reduced. Empirical studies (GON78) on a class of random processor flow graphs indicate that substantial savings in computer time are common using the ICG.

2.5.4. Dynamic assignment

If we allow modules to be reassigned during program execution, the cost of reassignment will influence the cost of the optimal dynamic assignment. Bokhari (BOK77) uses an extension of Stone's original algorithm for the static case in considering this problem. A similar approach will be used in Chapter 3 of this thesis. However, Bokhari includes a fourth cost category, that of module residence costs. The idea is that even when a module is inactive during program execution, it is incurring a cost that is a function of the processor to which it is currently assigned. This may be the processor on which the previous instance of that module was executed, the processor on which the next instance is to be executed, or (in theory at least) some other processor in the system. Furthermore, an active module at any stage or phase of program execution may communicate with any other module(s) in the program. Inactive modules may be reassigned between any successive module instances. The additional complexity that is introduced by this expanded model makes the resulting dynamic processor flow graph somewhat unwieldy, perhaps unnecessarily so. Bokhari indicates that
simplifying some of the original assumptions concerning program
behavior and costs will greatly reduce the overall complexity.
The dynamic algorithm suffers from the same difficulties as
the static flow algorithm.

2.6. Thesis direction

This thesis will deal with the general problem of finding
the optimal dynamic assignment for a program execution.
Chapter 3 follows an approach similar to that described in
(BOK77) in that it is based on an extension of the optimal
static assignment algorithm of Stone. However, the algorithm
described here deals with problems which are significantly
different from those considered by Bokhari. This algorithm
will suffer from similar limitations in attempting to generalize
it to more than two processors. Chapters 4 and 5 represent the
results of an attempt to overcome these limitations, using a
dynamic programming approach. Although the relatively straight-
forward dynamic programming algorithm succeeds in greatly expand-
ing the domain of solvable problems over the flow algorithm,
it pays a high price in worst case computational time and space
complexity. Section 4.4 shows that under relatively mild
assumptions, the expected space complexity of the dynamic pro-
gramming algorithm is comparable to that of the flow algorithm.
Chapter 5 deals with a modification to the dynamic programming
algorithm which emphasizes its equivalence to a shortest path
algorithm, similar in some respects to the shortest path algorithm
proposed by Bokhari for the static assignment problem. It is shown that the modified algorithm can take advantage of "predictability" in program behavior to greatly reduce the expected time complexity, while maintaining the same range of applicability as the original dynamic programming algorithm.
CHAPTER 3

DYNAMIC ASSIGNMENT WITH MAX-FLOW MIN-CUT ALGORITHMS

3.1. Introduction

It is possible to extend Stone's static case algorithm to the dynamic case (ST077) (BOK77). This section describes an algorithm which uses the same commodity flow-based approach for determining optimal dynamic module assignments. First we describe the constraints on the DPS under which the dynamic algorithm will be formulated. A very brief discussion of commodity flow networks follows, and the max-flow min-cut theorem is presented. We then show how the dynamic assignment problem may be represented graphically by constructing a dynamic processor flow graph. The max-flow min-cut theorem is used to provide us with a means of efficiently computing the optimal assignment through its application to the flow graph. Finally, the limitations of this approach are discussed.

3.2. Restricted DPS model

The overall cost of executing a program in a distributed environment is a combination of execution costs, communication costs, and reassignment costs. Execution costs for each module instance are simply a function of the processor to which the instance is assigned. Execution costs may vary from instance to instance of the same module on the same processor. Communications costs for the purpose of this section will be assumed
to be symmetric and uniform. Two module instances which communicate while assigned to different processors incur the same cost of communication, regardless of the two processors to which they are assigned. Reassignment costs, on the other hand, are assumed to be asymmetric. The cost of reassigning a module from processor X to processor Y will not in general be the same as if the module were reassigned from Y to X. For didactic purposes, reassignment costs will also be assumed to be uniform. Except for the trivial case of two processors, however, a DPS with uniform but asymmetric communication or reassignment costs seems highly unlikely.

Initially we will also restrict our attention to two-processor systems. As will be seen shortly, this limitation is not an arbitrary one for the algorithm to be presented in the remainder of Chapter 3, but is a natural consequence of the assumption of asymmetric reassignment costs. However, in order to postpone the discussion of this, we will constrain the problem domain. Of course, the constraint of only two processors makes the assumption of uniform communications costs unnecessary.

To summarize, the salient features of the DPS which we will be considering are:

1. two processors,
2. asymmetric reassignment costs, and
3. symmetric communications costs.

Also recall that we assume that no two modules of the same program may be executed concurrently.
3.3. **Commodity flow graphs**

The algorithm to be presented in this chapter is based on the theory of single-commodity flow networks. This section provides a brief introduction to the subject of commodity flows and the results of which we will be making use. For more complete treatments of the subject see (HU69) (EVE73) (FOR62).

Commodity flow problems arise in situations where the manufacturer (or distributor) of a product (the commodity) wishes to know how much of the commodity he may transport from the commodity source or sources to one or more destinations. There are a number of routes over which the commodity may be moved, but in general there is a maximum amount of commodity that may be shipped between any two points in the transportation network.

The graphical model of a commodity transportation network is called a **commodity flow network**. The commodity flow network is a directed or undirected graph which consists of a set of nodes \( N \), a set of edges \( E \), and a set of edge capacities. The set \( N \) consists of three mutually disjoint subsets representing the three classes of distinguished nodes.

\[
N = S \cup T \cup V
\]

where

\[
S = \{ w \in N \mid w \text{ is a source node} \}
\]

\[
T = \{ w \in N \mid w \text{ is a sink node} \}
\]

\[
V = \{ w \in N \mid w \text{ is an interior node} \}
\]
Source nodes in the commodity flow network correspond to commodity supply centers, which are assumed to be capable of supplying an infinite amount of the commodity. Sink nodes represent commodity demand centers, which are assumed to have a infinite capacity for absorbing the commodity. Interior nodes correspond to transshipment points in the commodity transportation network. There is an edge \((A,B)\) in the graph if there is a direct transportation link between the points represented by \(A\) and \(B\). If the commodity may be shipped in only one direction along a transportation link, then the corresponding edge of the graph is directed. The capacity of \((A,B)\), written as \(c(A,B)\), is the maximum amount of commodity which it is possible to transport from \(A\) to \(B\) along the link represented by \((A,B)\). Fig. 3.1 shows a typical commodity flow network.

A commodity flow is represented in the network as a set of weighted arrows associated with the edges of the graph. The direction of an arrow indicates the direction of commodity flow along its associated edge, while the weight of the arrow is the amount of commodity which is being transported along that edge. A feasible commodity flow is one in which

(i) the flow along any edge is less than or equal to the capacity of that edge,

(ii) the net flow into (out of) any interior node is zero, and

(iii) all sink nodes have nonnegative flow in, and all source nodes have nonnegative flow out.
Figure 3.1.

An undirected commodity flow network with single source $s$ and single sink $t$. The remaining nodes $a$, $b$, $c$, and $d$ are interior nodes.
Fig. 3.2 shows a feasible commodity flow for the network of Fig. 3.1.

The *value of a commodity flow* is simply the net flow into $T$ (for a feasible flow, this will be the same as the net flow out of $S$). A commodity flow whose values is maximum among all possible feasible flows is called a *maximum flow*. The maximum flow for the network of Fig. 3.1 is indicated in Fig. 3.3.

A *cutset* of a commodity flow network is a set of edges $E' \subseteq E$ such that

(i) if every edge in $E'$ is removed from the network, there is no path from any source node to any sink node (ignoring edge directions), and

(ii) no proper subset of $E'$ is also a subset.

The *capacity of a cutset* (for a commodity flow network with only undirected edges) is the sum of the capacities of the edges in the cutset.

It should be fairly obvious that the capacity of the minimum capacity cutset of a commodity flow network is an upper bound on the value of the maximum flow in the network, since the minimum capacity cutset forms a sort of bottleneck for commodity flow between the source nodes and sink nodes. Ford and Fulkerson (FOR62) were able to show an even more intimate relationship between the two. This relationship is expressed in the Max-Flow Min-Cut Theorem, which states:
Figure 3.2.
A feasible flow in a commodity flow network. The value of the flow is 12.
Figure 3.3.

Maximum flow in the commodity flow network of Fig. 3.1. The value of the maximum flow is 13. The minimum capacity cutset of the network is shown as the broken line.
The value of the maximum flow in a commodity flow network is equal to the capacity of the minimum capacity cutset of the network.

Based on this result, Ford and Fulkerson were able to develop an algorithm for the efficient determination of the maximum flow in a commodity flow network. Their algorithm was subsequently improved upon by Edmonds and Karp (EDM72). Recently Dinic and Karzanov constructed an even more efficient algorithm whose time complexity is bounded by $O(|N|^3)$ (EVE76). The latest improvement, due to Galil, has a complexity of $O(|N|^{5/3}|E|^{2/3})$, which for dense graphs is $O(|N|^3)$, but for sparse graphs is only $O(|N|^{2-1/3})$ (GAL78).

3.4. Optimal dynamic assignment using commodity flow networks

3.4.1. Introduction

Following the approach used by Stone (ST077) in solving the static assignment problem, we will make use of the above algorithms for determining maximum flows in commodity flow networks to find the optimal dynamic assignment of a program. We will construct a graphical representation of the dynamic assignment problem and establish a relationship between cutsets of the graph and solutions of the problem. In addition, we will examine some of the limitations inherent in this approach.
3.4.2. Dynamic module interconnection graph

In the static case algorithm, the first step is to construct a module interconnection graph, which explicitly incorporates all possible intermodule communications costs. For the dynamic case, we similarly construct a dynamic module interconnection graph which includes all possible inter-module instance communication costs. The graph will contain additional edges with edge capacities that are related to module reassignment costs.

Assume that the program to be distributed is partitioned into n modules, and there are m module instances during the program's execution. If \( A \) is an arbitrary module of the program, then \( A_i \) is defined as the ith instance of module \( A \). As a matter of notational convenience, the subscript of the last instance of module \( A \) will be denoted by \( \max(A) \). The last instance of \( A \) can be written as \( A_{\max(A)} \) or, unambiguously, simply as \( A_{\max} \).

We begin the construction by creating m nodes, labeled with the m module instance names. There will be an undirected edge between two nodes labeled \( A_i \) and \( B_j \) if the ith instance of module \( A \) immediately precedes the jth instance of module \( B \) during program execution. The capacity of \((A_i, B_j)\) will be the cost of communication between \( A_i \) and \( B_j \) if they are assigned to different processors. Table 3.1 describes some possible costs for a hypothetical program. Fig. 3.4 shows the partially
(a) Module instance execution costs

<table>
<thead>
<tr>
<th>Module Instance</th>
<th>P₁</th>
<th>P₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₁</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>B₁</td>
<td>12</td>
<td>8</td>
</tr>
<tr>
<td>A₂</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>C₁</td>
<td>20</td>
<td>8</td>
</tr>
<tr>
<td>B₂</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>A₃</td>
<td>10</td>
<td>20</td>
</tr>
</tbody>
</table>

(b) Intermodule communication costs

<table>
<thead>
<tr>
<th>Communicating Instances</th>
<th>Cost*</th>
</tr>
</thead>
<tbody>
<tr>
<td>A₁ - B₁</td>
<td>2</td>
</tr>
<tr>
<td>B₁ - A₂</td>
<td>7</td>
</tr>
<tr>
<td>A₂ - C₁</td>
<td>12</td>
</tr>
<tr>
<td>C₁ - B₂</td>
<td>1</td>
</tr>
<tr>
<td>B₂ - A₃</td>
<td>7</td>
</tr>
</tbody>
</table>

* if modules are assigned to different processors

(c) Module reassignment costs

<table>
<thead>
<tr>
<th>Module</th>
<th>To P₂</th>
<th>To P₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>B</td>
<td>6</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 3.1
Figure 3.4.

Incomplete dynamic module interconnection graph for the hypothetical program given in Table 3.1. Only communication costs have been incorporated.

Figure 3.5.

Complete dynamic module interconnection graph based on the costs given in Table 3.1.
constructed dynamic module interconnection graph in which all communication costs have been included.

In addition, there will be an undirected edge between every module instance \( A_i \) and the next instance \( A_{i+1} \) of the same module (provided that there is more than a single instance of module \( A \)). Let \( r_{j,k}(A) \) be the cost of reassigning module \( A \) from processor \( j \) to processor \( k \). Then we define

\[
(1) \quad c(A_i, A_{i+1}) = \frac{1}{2} [r_{1,2}(A) + r_{2,1}(A)] \quad , 1 \leq i \leq \text{max}(A)-1
\]

Note that if \( r_{1,2}(A) = r_{2,1}(A) \) as in the case of symmetric re-assignment costs, then \( c(A_i, A_{i+1}) \) is simply the cost of reassigning \( A \). There are no other nodes or edges in the dynamic module interconnection graph. Fig. 3.5 is the completed graph for the example of Table 3.1.

3.4.3. Dynamic processor flow graph

From the dynamic module interconnection graph we construct a dynamic processor flow graph. First, add two nodes labeled \( P_1 \) and \( P_2 \) for processors 1 and 2, respectively. For every node \( A_i \) of the interconnection graph we add undirected edges \( (A_i, P_1) \) and \( (A_i, P_2) \). Let \( e_j(A_i) \) be the cost of executing module instance \( A_i \) on processor \( j \). Then the capacities of the 2m new edges are computed as follows:

\[
(2) \quad c(A_i, P_2) = e_1(A_i) \quad , 1 \leq i \leq \text{max}(A)
\]

\[
(3) \quad c(A_i, P_1) = e_2(A_i) + \frac{1}{2} [r_{2,1}(A) - r_{1,2}(A)] , \text{max}(A) \neq 1
\]

\[
(4) \quad c(A_{\text{max}}, P_1) = e_2(A_{\text{max}}) + \frac{1}{2} [r_{1,2}(A) - r_{2,1}(A)] , \text{max}(A) \neq 1
\]

\[
(5) \quad c(A_i, P_1) = e_2(A_i) \quad , 1 < i < \text{max}(A) \text{ or } 1 = i = \text{max}(A).
\]
The lack of symmetry in these equations with respect to \( P_1 \) and \( P_2 \) is explained by the fact that the set of equations given above does not constitute a unique valid assignment of edge capacities but simply one of many possibilities. For an explanation of the derivation of these equations and other possible valid capacity assignments see Appendix A. The dynamic processor flow graph thus consists of \( m+2 \) nodes and \( 4m-(n+1) \) edges. Fig. 3.6 is the dynamic processor flow graph for the hypothetical example.

3.4.4. Optimal dynamic assignment using dynamic processor flow graphs

The dynamic processor flow graph contains all of the information relevant to the optimal dynamic assignment problem. The remaining step of the algorithm is to consider the dynamic processor flow graph as a commodity flow network, with \( P_1 \) as a single source node and \( P_2 \) a single sink node. Then using a max-flow min-cut algorithm, find the minimum capacity cutset of the network.

Theorem: The minimum capacity of any cutset of the dynamic processor flow graph is equal to the cost of the minimum cost dynamic assignment of the program.

Proof:

First we establish the convention that for a given cutset of the dynamic processor flow graph, a module instance \( A_i \) is assigned to processor \( j \) if and only if the edge \((A_i, P_j)\) is not
Figure 3.6.

Dynamic processor flow graph for the program described by Table 3.1.
in the cutset. Clearly there is a one-to-one correspondence 
between the set of all possible dynamic assignments for the 
program and the set of all possible cutsets of the graph. Then 
we claim that the capacity of any cutset of the graph is equal 
to the cost of the corresponding dynamic assignment. To prove 
this assertion, we will examine each of the costs separately.

**Module instance execution costs:**

If module instance $A_i$ is assigned to $P_1$ by the cutset, 
then $c(A_i, P_2)$ is included in the cutset capacity. $c(A_i, P_2)$ 
is the cost of executing $A_i$ on processor 1. If $A_i$ is 
assigned to $P_2$, then $c(A_i, P_1)$ is part of the cutset capacity, 
not $c(A_i, P_2)$. $c(A_i, P_1)$ includes the cost of executing $A_i$ on 
processor 2.

**Module instance communication costs:**

If two module instances $A_i$ and $B_j$ are assigned to different 
processors by the cutset, any edge $(A_i, B_j)$ if it exists must 
be in the cutset. Otherwise, there would exist a path from 
$P_1$ to $P_2$ over which flow could be increased. The edges in the 
path would be either $(P_1, A_i)$, $(A_i, B_j)$, and $(B_j, P_2)$, or 
$(P_1, B_j)$, $(B_j, A_i)$, and $(A_i, P_2)$. The existence of such a path 
contradicts the assumption of maximum flow. Hence, if $A_i$ and 
$B_j$ communicate, there will be an edge between the corresponding 
nodes in the graph, and the capacity of the edge will be included 
in the capacity of the cutset. $c(A_i, B_j)$ is the cost of communica-
tion between $A_i$ and $B_j$. 
Module reassignment costs:

To verify that module reassignment costs are properly accounted for by the cutset capacity, we could use the same technique described in Appendix A in deriving the reassignment cost contributions to individual edge capacities. However, we will use a somewhat more straightforward approach in dealing with these costs. For each module A in the program there are four cases to be considered.

(i) $A_1$ assigned to $P_1$, $A_{\text{max}}$ assigned to $P_1$ -

Each time A is reassigned to $P_2$, it must subsequently be reassigned to $P_1$. If A is reassigned to $P_2$ $x$ times, then there will be $2x$ edges between successive instances of A in the cutset. Since each of these edges has capacity $\frac{1}{2}[r_{1,2}(A) + r_{2,1}(A)]$, the total contribution of A's reassignment costs to the capacity of the cutset is

$$2x \cdot \frac{1}{2}[r_{1,2}(A) + r_{2,1}(A)] = xr_{1,2}(A) + xr_{2,1}(A)$$

as required.

(ii) $A_1$ assigned to $P_2$, $A_{\text{max}}$ assigned to $P_2$ -

If A is reassigned to $P_1$ $x$ times, then there will be $2x$ edges between successive instances of A in the cutset, with a combined capacity of $xr_{1,2}(A) + xr_{2,1}(A)$, as in (i). The contributions to cutset capacity from $(A_1, P_1)$ and $(A_{\text{max}}, P_1)$ due to reassignment costs cancel each other.
(iii) $A_1$ assigned to $P_1$, $A_{\text{max}}$ assigned to $P_2$ -
If $A$ is reassigned to $P_2$ from $P_1$ $t$ times, it is
reassigned $x-1$ times from $P_2$ to $P_1$. The contribution to cutset
capacity due to edges between successive instances of $A$ will
be $[x + (x-1)] \cdot \frac{1}{2}[r_{1,2}(A) + r_{2,1}(A)]$. In addition there will
be a contribution of $\frac{1}{2}[r_{1,2}(A) - r_{2,1}(A)]$ from $c(A_{\text{max}}, P_1)$,
resulting in a total $xr_{1,2}(A) + (x-1)r_{2,1}(A)$ being added to
the cutset capacity due to reassignments of $A$.

(iv) $A_1$ assigned to $P_2$, $A_{\text{max}}$ assigned to $P_1$ -
This case is similar to (iii), but the additional
contribution is from $c(A_1, P_1)$ instead of $c(A_{\text{max}}, P_1)$, and will
be $\frac{1}{2}[r_{2,1}(A) - r_{1,2}(A)]$. The total contribution to cutset
capacity due to $x$ reassignments of $A$ from $P_2$ to $P_1$ and $x-1$
from $P_1$ to $P_2$ will be $(x-1)r_{1,2}(A) + xr_{2,1}(A)$.

QED

Corollary: The optimal dynamic assignment for the two processor
case can be computed in $O(m^{2-1/3})$ time.

Proof:

Construction of the dynamic processor flow graph requires
$O(m)$ time since there are $4m + (n-1)$ edges and $m > n$. The Galil
max-flow min-cut algorithm will require
$O(|V|^{5/2} \cdot |E|^{2/3}) = O(m^{2-1/3})$
time to find the minimum capacity cutset.

The minimum capacity cutset and optimal dynamic assignment
for the dynamic processor flow graph of Fig. 3.6 shown in Fig. 3.7.
Figure 3.7.

Dynamic processor flow graph with minimum capacity cutset shown in broken line. The cost of the optimal dynamic assignment is equal to the capacity of the mincut, which is 52. Note that the optimal dynamic assignment assigns $A_1$ and $A_3$ to $P_1$, but $A_2$ is assigned to $P_2$. 
Note that the minimum capacity cutset and hence the optimal
dynamic assignment are unique for the example, but in general
this need not be true. For the single objective optimization
which we are considering, any minimum capacity cutset is
acceptable. It is possible that we might wish to optimize
some secondary criterion, such as minimal interprocessor
communication cost, given a minimum cost assignment. We could
then choose from among all minimum cutsets of the graph to
satisfy the supplementary objective.

In the example of Fig. 3.7, both A and B are reassigned
before each instance. To verify that this does indeed represent
an improvement over the optimal static assignment, we use the
information in Table 3.1 to construct a static processor flow
graph as described in Stone (ST077). Reassignment costs are
ignored. There is a node for each module rather than module
instance. Edges between module nodes have capacity equal to
the sum of communication costs between all instances of the
two modules. (Communication costs are therefore implicitly
symmetric in Stone's analysis). The capacity of a processor
node-module node edge is the total cost of executing all
instances of the module on the other processor. The static
processor flow graph and its minimum cutset are shown in Fig. 3.8.
We find that the optimal dynamic assignment does have less cost
than the optimal static assignment. Of course, there is some
overhead required to compute each of these assignments. The
(a) Static module interconnection graph for the dynamic assignment problem previously considered. (b) Static processor flow graph. The minimum capacity cutset, shown in a broken line, indicates that the optimal static assignment has a cost of 54, while the optimal dynamic assignment has a cost of 52 (see Fig. 3.6.)
difference in the amount of overhead determines whether or not it is cost effective to use a dynamic assignment algorithm. We feel that for many programs, the cost of the optimal dynamic assignment will be sufficiently less than that of the optimal static assignment to justify the increase in overhead.

Before concluding the discussion of the flow graph algorithm, we need to resolve one potentially troublesome point in the algorithm as presented above. It is obvious that equation (3) or (4) in section 3.4.3 under the proper circumstances will yield a negative edge capacity. Max-flow min-cut algorithms allow only nonnegative edge capacities. However, a simple solution to this difficulty exists. A negative edge capacity is possible only on a processor node-module instance node edge. For each module instance node \( A_i \), exactly one of the edges \((A_i, P_1)\) and \((A_i, P_2)\) will be in any cutset of the graph. Suppose one of these edges has a negative edge capacity \(-r\) as determined by equation (3) or (4). Then add \(+r\) to both \(c(A_i, P_1)\) and \(c(A_i, P_2)\). Now both edges will have nonnegative edge capacities. Furthermore, every cutset of the graph has had its capacity increased by \(r\). We can now use a max-flow min-cut algorithm to determine the minimum capacity cutset, which still represents the optimal dynamic assignment, since the cost of each possible assignment has been biased by the same amount \(r\). After the minimum capacity cutset is found, the cost of the optimal assignment is the cutset capacity, less \(r\).
3.5. Limitations of the flow algorithm

In this section we will examine some of the limitations of the flow algorithm. Some of the restrictions imposed before presenting the algorithm were for the purpose of simplifying the analysis, while others are represent major problems for the commodity flow approach. In particular expansion to three or more processors presents difficulties.

One restriction made for the sake of simplicity was that of symmetric communication costs. There is a great deal of similarity between communication costs and reassignment costs. Both are costs related to pairs of module instances. Both are zero if the related module instances are assigned to the same processor. Clearly, asymmetric communication costs in the two processor problem may be taken into account in exactly the same manner as asymmetric reassignment costs.

The restriction of the problem to the two processor case has hidden a major stumbling block which becomes apparent when we attempt to expand the flow algorithm to three or more processors. With \( p > 2 \) we introduce the possibility of communication and reassignment costs which may be nonuniform as well as asymmetric. It is possible to construct a dynamic (or static) processor flow graph for \( p > 2 \) if these costs are uniform and symmetric. If they are not, it does not appear possible to incorporate them into the framework of a processor flow graph formulation of the static or dynamic assignment problem.
The most serious difficulty we encounter in attempting to use this approach for three or more processors is not in the construction of the processor flow graph, however, but in finding the optimal assignment using the graph. As was mentioned in section 2.4.1, there is no known method for finding a minimum capacity multi-partition cutset of a graph which in the worst case is any better than exhaustive enumeration. The problem has been shown to be NP-complete for \( p > 3 \), so that it is doubtful that we will ever have a polynomial time algorithm which will give us the optimal solution in every instance of the problem. It is the efficiency of the max-flow min-cut algorithm which renders the processor flow graph formulation so attractive for two processors; without this efficiency, such an approach for more than two processors may have little to offer.

3.6. Summary of dynamic processor flow algorithm

The problem of determining an optimal dynamic assignment for an execution of a program has, under certain restrictions, been shown to be capable of efficient solution using commodity flow networks and max-flow min-cut algorithms. The assignment problem is given a graphical construction as a dynamic processor flow graph. When this graph is considered as a commodity flow network, it has the essential property that the capacity of the minimum capacity cutset of the graph is equal to the cost of the optimal dynamic assignment, and there exist efficient algorithms for finding the minimum capacity cutset. However,
limitations of the flow graph approach, especially in problems with more than two processors, make it unsuitable or unusable in many cases. An effort to overcome these limitations using a totally different approach is described in the next chapter.
CHAPTER 4

AN APPLICATION OF DYNAMIC PROGRAMMING
TO THE OPTIMAL DYNAMIC ASSIGNMENT PROBLEM

4.1. Motivation for considering dynamic programming

Because of the lack of generality in the processor flow graph for the algorithm described in the previous section, we are encouraged to consider other methods for solving the optimal dynamic assignment problem in those instances of the problem which do not lie within the domain of the flow algorithm. Dynamic programming has a much wider scope of applicability, allowing the inclusion of nonuniform, asymmetric reassignment and communications costs, although it is computationally much less desirable, as we shall see.

Dynamic programming is appropriate because the optimal dynamic assignment problem obeys Bellman's Principle of Optimality (BEL62), which states:

An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.

In the dynamic assignment problem, the state at any particular decision stage might be defined as the assignments for all module instances which precede the instance whose assignment we are deciding. However, the number of possible states would grow
exponentially with the number of module instances $m$. We would essentially be creating a decision tree of depth $m$ and having $p^m$ leaves, where $p$ is the number of processors. The construction of the tree would require $O(p^m)$ time instead of the $O(mp^m)$ time for brute force exhaustive enumeration, but the exponential growth of the time complexity renders this approach impractical for all but the smallest problems.

Fortunately, the maximum number of states at any decision stage in the dynamic programming algorithm is far less than $p^m$. Suppose we have the module instance sequence $A_{i-1}...B_jA_i...$. Then the cost of assigning $A_i$ to a particular processor $P_k$ is a function of only two preceding module instance assignments, those of $A_{i-1}$ and $B_j$. Furthermore, as soon as the assignment of $A_i$ is determined, the assignment of $A_{i-1}$ becomes irrelevant in selecting an optimal assignment for any subsequent module instances. This means that the state at any given decision stage is not the set of assignments of all previous module instances, but the set of assignments for the most recent instances of all preceding modules. The computations necessary at each stage will be independent of the number of module instances previously considered. It is this limitation in the total number of possible states which makes a dynamic programming approach more feasible.
4.2. The dynamic programming algorithm

4.2.1. Relation to shortest path algorithm

The dynamic programming algorithm which will be presented in the remainder of this section can be fruitfully thought of as a shortest path algorithm, albeit for a particularly simple class of graphs in which all possible paths to any node have the same number of edges. Visualizing the dynamic programming algorithm as constructing a directed graph and then finding a tree of shortest paths within the graph will be particularly useful in allowing us to present the algorithm and its proof of correctness in a very straightforward and (hopefully) understandable manner.

Conceptually, we will build a dynamic assignment graph. For each module instance, there will be $p^n$ nodes, one for each possible state at that instance (actually, the number of nodes may be less than this, as will be described in section 4.2.2.). There will be $p$ possible assignments for the next module instance. These potential assignments are represented by $p$ directed edges from each node related to the current instance to $p$ nodes belonging to the next instance. The weight of an edge is the cost of making the corresponding assignment, given the state represented by its source node. The state of the destination node will be the state of the source node, altered appropriately by the edge assignment. Since there will be $p^n$ states for each instance, there will be $p$ paths converging on
each node, also. After constructing this graph for all module instances, the shortest path to a node belonging to the last module instance will represent the minimum cost dynamic assignment.

Actually, the dynamic assignment graph is never constructed by the algorithm. Since all paths are of equal length, we may consider each module instance separately, and build only the tree of shortest paths to the nodes for each instance. This tree will be called the dynamic assignment tree. As a consequence the expected space complexity of the algorithm will be far less than the size of a complete dynamic assignment graph. This is shown in section 4.3.

The interpretation of the dynamic programming algorithm as a shortest path-finding approach is also useful in considering the modified algorithm of section 5. The modified algorithm has the potential for significantly reducing the expected amount of work for very large problems.

4.2.2. Description of algorithm

The computation of an optimal dynamic assignment will be divided into a series of stages, one stage \( s_i \) for each module instance \( i \). In addition there is an initial stage \( s_0 \). \( s_0 \) consists of a single start node which has a value of 0. In any intermediate stage \( s_i \) there will be \( p^{\lambda(i)} \) nodes in that stage, where \( p \) is the number of processors and \( \lambda(i) \) is the number of distinct modules which have instances among the first \( i \) module instances.
in the program execution. \(\lambda(i)\) is a monotonically nondecreasing function of \(i\), with \(\lambda(1) = 1\) and \(\lambda(m) = n\), with \(n\) being the number of modules in the program and \(m\) being the number of module instances during the lifetime of the program. \(\lambda(0)\) will be defined to be 0. At the conclusion of processing the \(i\)th stage, for each node \(v\) of \(s_i\) there will be exactly one path (of length \(i\)) from the start node to \(v\). \(v\) will uniquely represent one of the \(p^{\lambda(i)}\) possible states or combinations of most recent assignments for the \(\lambda(i)\) distinct modules encountered through module instance \(i\). The value of \(v\) will be the cost of the optimal dynamic assignment for the first \(i\) instances, given the state represented by \(v\).

The algorithm proceeds as follows:

(1) \(i = 0; s_0\) contains a single node called the start node, which has value 0; \(\lambda(0) = 0\).

(2) \(i = i + 1;\) if \(i > m\) go to 5.

(3) If \(\lambda(i) = \lambda(i-1)\) go to 4;

else for each node \(v\) in \(s_{i-1}\):

/* the \(i\)th module instance is the first instance of a module */. Create \(p\) nodes in \(s_i\), with an edge from \(v\) to each; the edges are labeled with the \(p\) choices of assignment for module instance \(i\). The value of a node \(w\) in \(s_i\) with an edge from \(v\) is the value of \(v\) plus the cost of assigning the \(i\)th instance to the processor with which \((v,w)\) is labeled, assuming that
the \( \lambda(i-1) \) modules which have been encountered through
\( s_{i-1} \) are assigned according to the state represented
by \( v \). Go to 2.

(4) /* \( \lambda(i) = \lambda(i-1) \); the ith module instance is an instance
of a module which has a prior instance */
Create \( p^{\lambda(i)} \) nodes in \( s_i \). Since \( \lambda(i) = \lambda(i-1) \), there
are also \( p^{\lambda(i)} \) nodes in \( s_{i-1} \). Let \( A_j \) be the ith module
instance. We partition the nodes of \( s_{i-1} \) into \( p^{\lambda(i)-1} \)
disjoint subsets of \( p \) nodes each. The partitioning
is done such that the nodes of each subset terminate
paths in which the most recent assignments of the
\( \lambda(i-1) \) modules differ only in the assignment of \( A_{j-1} \).
Let us call these subsets partial state sets or p-sets.
For each p-set in \( s_{i-1} \), there will be a p-set also of
\( p \) nodes, in \( s_i \) corresponding to the same partial state.
(Note that partial states at each stage are defined
by the module whose instance occurs at that stage).

Let \( w \) be a node in a p-set of \( s_i \). Then every node in
the corresponding p-set of \( s_{i-1} \) will initially have
an edge to \( w \). Each of these edges represents the same
choice of assignment for \( A_j \) and will be labeled with
that processor's name. The weight of an edge \((v,w)\)
from a node \( v \) of \( s_{i-1} \) to \( w \) is the cost of assigning
\( A_j \) to the processor with which \((v,w)\) is labeled, given
the state corresponding to \( v \).
A representative $p$-set of $s_{i-1}$, with the corresponding $p$-set of $s_i$. There are $p$ nodes in each, where $p$ is the number of processors. The $p$ edges from each node of $s_{i-1}$ to $s_i$ represent the $p$ possible choices of assignment for the $i$th module instance. All lines to a node of $s_i$ represent assignment to the same processor.
There are \( p-1 \) other nodes in the same \( p \)-set as \( w \), each having \( p \) edges incident on it initially from the same \( p \)-set of \( s_{i-1} \) as \( w \) does. These \( p-1 \) nodes represent the remaining \( p-1 \) choices of assignment for \( A_j \), all edges to a node being labeled with the same processor name and having the appropriate weights. Fig. 4.1 illustrates two related \( p \)-sets of successive stages and the initial interconnections between them.

As a result of the initial construction of \( s_i \), each node in \( s_i \) has \( p \) paths from the start node converging on it. The length of the path from \( s_0 \) to \( w \) passing through \( v \) is the value of \( v \) plus the weight of \((v,w)\). For each node \( w \) of \( s_i \), choose the shortest path from \( s_0 \) to \( w \). Remove all edges incident on \( w \) which are not on the shortest path, and let the value of \( w \) be the length of the shortest path to \( w \).

Go to 2.

(5) Select the node in \( s_m \) with minimum value.

End.

Theorem: At the termination of the above algorithm, the minimum value of a node in \( s_m \) is equal to the cost of the minimum cost dynamic assignment for the program execution. Furthermore, the minimum cost assignment is indicated by the processor labels on the edges of the unique path from the start node to the minimum value node of \( s_m \).
Proof:

It is only necessary to show that at the end of execution of step 3 or 4 for an arbitrary stage \( s_i \), there is a unique node in \( s_i \) for each of the \( p^{\lambda(i)} \) possible states and that the value of each node in \( s_i \) is the minimum cost of assignment for the first \( i \) module instances, given the state (the most recent assignments of the \( \lambda(i) \) modules) represented by that node. Proof of this hypothesis will be by induction on \( i \).

Basis step: \( i = 1 \)

Since \( \lambda(1) > \lambda(0) \), step 3 is invoked and the hypothesis is trivially true.

Induction step:

Assume that the hypothesis is true for \( i = k-1 \). That is, assume there are \( p^{\lambda(k-1)} \) nodes in \( s_{k-1} \), one for each of the \( p^{\lambda(k-1)} \) possible states, and there is a unique path from the start node to each node \( v \) of \( s_{k-1} \) such that \( v \) has minimum value, given the state represented by \( v \). Then for \( s_k \) we have two possibilities.

(1) If \( \lambda(k) \neq \lambda(k-1) \), the hypothesis will obviously be true for \( i = k \), since step 3 of the algorithm will be invoked, appending \( p \) edges onto the end of each of the already existing \( p^{\lambda(k-1)} \) paths. There is one edge appended to each path for each of the \( p \) choices of assignment for the \( k \)th module instance. Therefore, all paths to nodes in \( s_k \)}
are unique. Each node \( w \) in \( s_k \) represents an "expansion" of the state of its predecessor \( v \) in \( s_{k-1} \) to include the assignment of the new module whose first instance is the \( k \)th module instance. Since \( v \) has minimum value, \( w \) will also have minimum value.

(2) If \( \lambda(k) = \lambda(k-1) \), step 4 is used to determine paths to node in \( s_k \). The edge to a node \( w \) in \( s_k \) is chosen from among a set of edges which

(a) come from the same p-set in \( s_{k-1} \), and

(b) represent the same choice of assignment for \( A_j \), the \( k \)th module instance.

Hence, we initially have \( p \) paths converging at \( w \), each path having identical edge labels except for the edge corresponding to the choice of assignment for \( A_{j-1} \). All paths converging at \( w \) therefore correspond to the same state, since the state at \( w \) includes the assignment of \( A_j \), not \( A_{j-1} \). A path to a node \( v \) of \( s_{k-1} \) corresponds to the optimal assignment of the first \( k-1 \) instances, given the state represented by \( v \). Hence the path to \( w \) through \( v \) corresponds to the optimal assignment of the first \( k \) instances, given the state represented by \( w \) and the assignment of \( A_{j-1} \) included in the state represented by \( v \). Since the p-set to which \( v \) belongs covers all possible assignments of \( A_{j-1} \), one of the initial paths to \( w \) will contain the optimal assignment of \( A_{j-1} \). Therefore, choosing
the edge to \( w \) which minimizes the value of \( w \) insures that the path to \( w \) corresponds to the optimal assignment of the first \( k \) instances, given the state of \( w \), and the hypothesis is true for \( i = k \).

QED

Figs. 4.2 and 4.3 illustrates the application of the dynamic programming algorithm to the example used in the previous section. In Fig. 4.2 we indicate all of the edges which are considered by the algorithm. Fig. 4.3 shows only the tree consisting of those edges not discarded in step 4 of the algorithm. The optimal dynamic assignment is indicated by the bold lines in Fig. 4.3.

It should be clear that we can easily incorporate asymmetric, nonuniform communication and reassignment costs in this algorithm. This is because the algorithm examines all possible states for each module instance in arriving at an optimal solution.

4.2.3. Time and space complexity

The penalty that we must pay for the wider applicability of the dynamic programming algorithm over the processor flow algorithm is increased time and space complexity. The flow algorithm time complexity is bounded by \( O(m^{2-1/3}) \) and its space requirements are \( O(m) \). (Remember that this is only for the two processor case).

The dynamic programming algorithm complexity, on the other hand, will obviously be a function not only of the number of
Figure 4.2.

All possible paths considered by the dynamic programming algorithm for the two-processor problem of Table 3.1. By convention the upper edge from each node to the next stage represents the choice of assignment to processor 1.
Figure 4.3.

Unpruned assignment tree for the example of Table 3.1. The optimal dynamic assignment is shown in bold lines.
module instances, but also of the number of modules \( n \) and the number of processors \( p \). For a typical stage there are \( p^n \) nodes. Each node originally has \( p \) edges to it. To calculate the value of a node requires us to select the minimum of \( p \) possible values, which can be done in \( O(p) \). The overall time complexity of the algorithm is therefore \( O(mp^{n+1}) \). (Since there are only \( p^3 \) possible different costs of assignment for any module instance independent of the number of modules, these calculations can be ignored in determining upper bounds).

The space complexity is essentially the storage requirements for the tree constructed by the algorithm. In the worst case, reassignment costs are so large that if we assume that the first instance of a module is assigned to a particular processor, all subsequent instances of that module will be assigned to the same processors. Each of the paths to the \( p^n \) nodes of \( s_m \) will be for the most part distinct from all other paths, as shown in Fig. 4.4. Consequently we must save \( O(mp^n) \) edges before we can select the single path representing the optimal assignment.

Neither of these two complexity bounds is particularly appealing. However, all is not as bleak as it may seem. Section 4.3 will indicate that under what are likely to be conservative assumptions, the average space complexity is considerably less than worst case. As far as time complexity goes, the average case and worst case for this algorithm are the same. At least the dynamic programming algorithm time complexity is linear in \( m \),
Figure 4.4.

An assignment tree with worst case space complexity. Reassignment costs are sufficiently large to overpower other cost considerations, so that no module is reassigned. Consequently no pruning is possible.
so that for "large" problems \((4/3 \log_2 m \geq n+1)\) it should compare favorably with the flow algorithm for a two-processor DPS. Chapter 5 describes a modification of the above algorithm which may allow considerable improvement in expected time complexity.

4.3. *Expected space complexity*

4.3.1. *Pruning*

Although the worst case space complexity of the dynamic programming algorithm is \(O(mp^n)\), we would expect, on the average, far less storage to be necessary. At any stage of the algorithm, we would most likely see some paths terminating while others fan out. By terminating we mean that the path ends in a node which has no edges to the next stage. As a consequence, the terminated path contains a number of edges which are irrelevant to a final solution, and these edges may be discarded from the tree. We call this process of discarding edges on terminated paths of the tree *pruning*.

Intuitively, one would expect that two nodes in a stage \(s_i\) which represent similar states will have their values similarly influenced by the assignments of those module instances among the first \(i\) which are not in the state at \(s_i\). This increases the likelihood that the paths to each of the nodes will share edges for many of the module instance assignments. If the paths agree on all of the current module assignments at any stage,
then they do so for all preceding stages as well. It is this
effect that we attempt to quantify in the next section.

4.3.2. Derivation of pruning probability recurrence relations

4.3.2.1. Assumptions

Pruning, as indicated above, is the discarding of those
edges of the tree constructed by the dynamic programming algorithm
which belong to subtrees none of whose leaves are in \( s_m \). In
order to determine an upper bound on the expected number of
edges which ultimately must be retained, we make the following
assumption:

The choice of an edge to a node is independent
of the choice of an edge for any other node in any
stage, and all possible edges are equally likely.

This assumption may appear to be somewhat strong, but we will
show that in fact it seems to be quite reasonable and if anything
somewhat conservative.

In attempting to justify the assumption, we will examine
three different situations. In the first we consider two nodes
which are in the same stage \( s_i \) but belong to different p-sets.
Then they will have no edges from the same node in \( s_{i-1} \), since
the corresponding p-sets of \( s_{i-1} \) are disjoint. Any correlation
between the choice of edges for the two nodes must be derived
from edge choices made in \( s_{i-2} \) or earlier, and the effects of
such a choice should be minimal.
The second case concerns two nodes which belong adjacent stages \( s_{i-1} \) and \( s_i \). Let the \((i-1)\)st module instance be \( A_j \), and the \(i\)th instance be \( B_k \), \( A \neq B \). In selecting edges to nodes in \( s_{i-1} \), the \( p \)-sets are formed with respect to the most recent assignment of module \( A \); in dealing with \( s_i \) the \( p \)-sets are with respect to \( B \). Each node in \( s_i \) will therefore initially have exactly one edge from each of the \( p \)-sets formed in considering \( s_{i-1} \). This is shown graphically in Fig. 4.5, assuming three processors in the DPS. When we are considering \( s_{i-1} \), the partitioning of the nodes of \( s_{i-1} \) includes the \( p \)-sets \( \{1', 2', 3'\}, \{4', 5', 6'\}, \) and \( \{7', 8', 9'\} \), with the corresponding \( p \)-sets of \( s_{i-2} \) being \( \{1, 2, 3\}, \{4, 5, 6\}, \) and \( \{7, 8, 9\} \), respectively. The figure shows all of the possible edges between the corresponding \( p \)-sets of \( s_{i-2} \) and \( s_{i-1} \).

When forming the \( p \)-sets for \( s_i \), however, one of the \( p \)-sets in the partition of \( s_i \) will be \( \{1'', 4'', 7''\} \), with the corresponding \( p \)-set in \( s_{i-1} \) being \( \{1', 4', 7'\} \). Because of this shuffling of nodes in \( p \)-sets of \( s_{i-1} \), a node in \( s_i \) such as \( 1'' \) will initially have an edge from three different former \( p \)-sets of \( s_{i-1} \). Since nodes of \( s_{i-1} \) in different \( p \)-set have their edges selected independently, it seems reasonable to assume that there will be very little correlation between edges to the same node of \( s_i \). Consequently, the choice of an edge to a node \( w \) in \( s_i \) should be independent of the choices of edges for \( s_{i-1} \).
Figure 4.5.

 Portions of three successive stages of the dynamic programming algorithm for a three-processor problem. Nodes 1', 2', and 3' are in the same p-set of $s_{i-1}$ but belong to three different p-sets when considering $s_i$. 
We finally consider two nodes belonging to the same p-set of \( s_i \). The assumption of independent choice and equal likelihood is a conservative one in this situation. To determine the number of edges which we expect to remove, we intend to find the probability that a node terminates the path to that node. This is the probability that none of the possible edges from that node to the next stage are chosen. Equal likelihood means that any of the possible interconnection patterns between successive p-sets is possible. However, consider the p-sets \{1', 4', 7'\} and \{1'', 4'', 7''\} of Fig. 4.5. If 4' has value significantly lower than 1' and 7', the edges (4', 1''), (4', 4''), and (4', 7'') are much more likely to be chosen than any of the others; if 4' has higher value than 1' and 7', the same edges are less likely to be chosen. (There seems to be no reason to assume that the value of a node will significantly affect the costs corresponding to the edges from it to the next stage). In either case, we increase the probability of paths being terminated. The assumption of equal likelihood and independence should therefore result in less edges being pruned than perhaps would might actually be the case.

Analysis of the dynamic programming algorithm's storage requirements based on this assumption should give us a reasonable upper bound on the average space complexity of the algorithm. The next section describes how pruning will be done, and section 4.3.2.3 presents at least a partially quantitative analysis of pruning.
4.3.2.2. Dynamic pruning

It would be worse than useless to wait until the tree was completely constructed before pruning it. Also, if pruning were unduly difficult, increasing the algorithm complexity, it would probably not be worthwhile considering it. This section indicates how pruning may be done dynamically as the algorithm proceeds in a very simple manner.

At the completion of the computation for any stage $s_i$, each node in $s_{i-1}$ is examined for edges to $s_i$. If a node has no edges to $s_i$, we removed the edge to it from $s_{i-2}$ and examine the node of $s_{i-2}$ from which the edge originated to see if it has any remaining edges to $s_{i-1}$. If not, we remove the edge to it from $s_{i-3}$. We continue in this manner until we reach a node which has one or more edges remaining to the next stage. At this point we return to $s_{i-1}$ and begin the process again with a new node. Thus, before we proceed to $s_{i+1}$ we have removed all edges from paths which terminate before $s_{i+1}$. Fig. 4.6 shows the tree of Fig. 4.2(b) after it has been pruned.

4.3.2.3. Quantitative analysis of pruning

Let us assume that the algorithm has just completed $s_k$, and that $\lambda(k) = n$. Then there are $p^n$ nodes of $s_k$ and consequently $p^n$ edges from $s_{k-1}$ to $s_k$ out of the $p^{n+1}$ initial edges (assuming $\lambda(k-1) = n$, also.) This means that $p^{n+1} - p^n = p^n(p-1)$ edges are removed between $s_{k-1}$ and $s_k$. This also holds true for the number of edges removed initially between any two previous
Figure 4.6.

The assignment tree of Fig. 4.3 with all edges which may be pruned from the tree shown as dotted lines.
stages. For the sake of simplicity in the analysis, we assume 
\( \lambda(i) = n \forall i \leq k \).

There are \( p \) ways of reaching all \( p \) nodes in a \( p \)-set of 
\( s_k \) from exactly one of the \( p \) nodes of the corresponding \( p \)-set of 
\( s_{k-1} \); the probability that a node of \( s_{k-1} \) has no edges to \( s_k \) 
for this case is \( \frac{p-1}{p} \).

Two nodes of \( s_{k-1} \) can be chosen in \( (p)_2 \) ways. \( p \) nodes 
of \( s_k \) can be partitioned into two nonempty groups in \( S(p,2) \) ways, 
where \( S(i,j) \) is Stirling's number of the second kind for \( i \) 
unlike objects and \( j \) like cells (RI058). Therefore, there are 
\( (p)_2 \cdot S(p,2) \) ways of reaching all \( p \) nodes in \( s_k \) through exactly 
two nodes of \( s_{k-1} \). The probability of a node in \( s_{k-1} \) having no 
edges to \( s_k \) is \( \frac{p-2}{p} \).

We can generalize the above relations for the case exactly 
\( i \leq p \) nodes in a \( p \)-set of \( s_{k-1} \) having edges to the corresponding 
\( p \)-set of \( s_k \). There are \( (p)_i \cdot S(p,i) \) ways in which this can 
occur, and in each case the probability that a particular node 
of \( s_{k-1} \) has no edges to \( s_k \) is \( \frac{p-i}{p} \).

By the above analysis there are 
\[
N_T = \sum_{i=1}^{p} (p)_i \cdot S(p,i)
\]
total possible ways of reaching all \( p \) nodes in one \( p \)-set of \( s_i \). 
This number is equal to \( p^P \), as we expect.

Now the probability that a node in \( s_{k-1} \) has no edges 
to \( s_k \) is: 
\[
\Pr \{ \text{no edges} \} = \frac{N_k}{N_T}
\]
<table>
<thead>
<tr>
<th>p</th>
<th>f(p)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.250000</td>
</tr>
<tr>
<td>3</td>
<td>0.296296</td>
</tr>
<tr>
<td>4</td>
<td>0.316406</td>
</tr>
<tr>
<td>5</td>
<td>0.327680</td>
</tr>
<tr>
<td>10</td>
<td>0.348678</td>
</tr>
<tr>
<td>20</td>
<td>0.358468</td>
</tr>
<tr>
<td>50</td>
<td>0.364170</td>
</tr>
<tr>
<td>100</td>
<td>0.366032</td>
</tr>
</tbody>
</table>

**Table 4.1**

f(p) is the initial pruning probability for each stage of a dynamic assignment tree for a p-processor system.

\[ f(p) = q_{m-1} = \left(\frac{p-1}{p}\right)^p \]
where

\[ N_x = \text{the total number of ways of reaching all} \]
\[ p \text{ nodes in } s_k \text{ in which a particular node of} \]
\[ s_{k-1} \text{ has no edges to } s_k \]

This is because all ways of reaching the p nodes in a p-set of \( s_k \) are equally likely, according to the assumption that we made in section 4.3.2.1.

\[
N_x = \sum_{i=1}^{p} \frac{p-1}{p} (p)_i S(p,i)
\]

\[
= \sum_{i=1}^{p} (p-1)_i \cdot S(p,i)
\]

\[
= (p-1)^p
\]

Hence,

\[
Pr [\text{no edges}] = \frac{(p-1)^p}{p^p} = \left( \frac{p-1}{p} \right)^p, \quad p \geq 1.
\]

Table 4.1 shows how \( \left( \frac{p-1}{p} \right)^p \) behaves as \( p \) gets very large. The function is bounded and even for small \( p \) is close to the upper limit. We can obtain the limit as follows:

\[
\lim_{p \to \infty} \left( \frac{p-1}{p} \right)^p = \lim_{p \to \infty} \left( 1 - \frac{1}{p} \right)^p
\]

\[
= \lim_{p \to \infty} \exp \left( p \cdot \ln \left( 1 - \frac{1}{p} \right) \right)
\]

\[
= \exp \lim_{p \to \infty} \left( p \ln \left( 1 - \frac{1}{p} \right) \right)
\]
\[
\lim_{p \to \infty} \exp \left( p \left( - \frac{1}{p} + \frac{1}{2p^2} - \frac{1}{3p^3} + \ldots \right) \right) = \exp (-1) = e^{-1} \approx 0.367879.
\]

Therefore, the probability that a node of \( s_{k-1} \) has no edges to nodes in \( s_k \) after the initial removal of edges between \( s_{k-1} \) and \( s_k \) is equal to \( \frac{1}{e} \), in the limit \( p \to \infty \).

Two processor case:

The \( p \)-sets in each stage contain two nodes. There are four possible ways of reaching the nodes in a \( p \)-set of \( s_i \) from \( s_{i-1} \). These are illustrated in Fig. 4.7. In each of the four ways, there is exactly one edge to each node in \( s_i \). A node in \( s_i \), on the other hand, may have no edges to nodes in \( s_{i+1} \).

Then since no path for an optimal solution can pass through that node, the edge to it from \( s_{i-1} \) may be pruned.

Let \( q_i \) be the probability that a node of \( s_i \) has no edges to \( s_{i+1} \). Then this is also the probability that we may prune any of the edges which remain between \( s_{i-1} \) and \( s_i \). We wish to calculate \( q_{i-1} \), the probability that a node of \( s_{i-1} \) will have no edges to \( s_i \) after all possible branches have been pruned from between \( s_{i-1} \) and \( s_i \).

Since each of the four ways of reaching nodes in \( s_i \) illustrated in Fig. 4.6 are equally likely, the expected value \( E_{i-1}[N_i] \) of the number of nodes in one \( p \)-set of \( s_{i-1} \) which will
All possible ways of reaching the nodes of an arbitrary p-set of $s_i$ from the corresponding p-set of the previous stage (two processor case).
have no edges to $s_i$ after pruning is

$$E_{i-1}[N_T] = \frac{1}{4} \left\{ 1 \cdot q_i \cdot (1-q_i) + 1 \cdot q_i \cdot (1-q_i) + 2 \cdot q_i^2 \right\} \quad (i)$$

$$+ \frac{1}{4} \left\{ 1 + l \cdot q_i^2 \right\} \quad (ii)$$

$$+ \frac{1}{4} \left\{ 1 + l \cdot q_i^2 \right\} \quad (iii)$$

$$+ \frac{1}{4} \left\{ 1 \cdot q_i \cdot (1-q_i) + 1 \cdot q_i \cdot (1-q_i) + 2 \cdot q_i^2 \right\} \quad (iv)$$

$$= \frac{1}{2} (q_i + 1)^2$$

Then

$$q_{i-1} = \frac{1}{2} \cdot E_{i-1}[N_T]$$

$$= \frac{1}{4} \cdot (q_i + 1)^2$$

The recurrence relation derived above, together with the initial value $q_m = 0$, completely defines the value $q_i$ for every stage $s_i$. Fig. 4.9 illustrates the behavior of $q_{m-i}$ for $m \gg i > 0$. Recall that these are the probabilities for pruning those edges which remain after eliminating the $p-1$ edges to each node during step 4 of the algorithm. Fig. 4.9 tells us that, for two processors, we expect to be able to eliminate over 90% of these remaining edges by pruning from $s_{m-34}$.

The recurrence relation for $q_i$ indicates that $q_{m-i} \rightarrow 1$ as $m \rightarrow \infty$ and $i \rightarrow m$.

$$q_{i-1} = \frac{1}{4} (q_i + 1)^2$$

$$\frac{q_{i-1}}{q_i} = \frac{1}{4} \left( q_i + \frac{1}{q_i} \right) + \frac{1}{2}$$

$$> 1$$
since \( f(x) = x + \frac{1}{x} \) has a minimum at \( x = 1 \) and

\[
f(1) = 2.
\]

Hence \( q_{m-i} \) is a monotonically increasing sequence for increasing values of \( i \). However, at some value of \( i \) which depends on \( n \), the probability of a node having an edge to the next stage becomes less than one over the number of nodes per stage. Let the first value of \( i \) for which this occurs be called \( i_2(n) \). Since there must always be a path from the start node to each of the nodes of \( s_m \), there can never be less than one edge between any two stages. For two processors, \( i_2(n) \) is found by

\[
1 - q_{m-i_2(n)} < \left( \frac{1}{2} \right)^n.
\]

\( 1 - \left( \frac{1}{2} \right)^n \) forms an upper bound on \( q_i \). The recurrence relation which actually defines \( q_{m-i} \) for two processors is

\[
q_{m-i} = \min \left[ 1 - \left( \frac{1}{2} \right)^n, \frac{1}{4} \cdot (q_{m-(i-1)} + 1)^2 \right]
\]

Section 4.3.3. gives some numerical results based on this recurrence relation.

**Three or more processors**

It is possible to use similar methods for \( p > 2 \) in generating recurrence relations, but the derivation of \( E_{i-1} [N_T] \) from \( q_i \) rapidly becomes overwhelming using the same technique as for \( p = 2 \). Fortunately there are shortcuts which may be employed to relieve the tedium of the calculations. Appendix B informally describes
a simple and easy-to-automate algorithm for generating these relations. Using this algorithm, we have generated the recurrence relation equations given in Fig. 4.8 for \( p = 3 \) through \( p = 7 \). Actually, these equations are only partially correct, since for each value of \( p \), \( q_i \) is subject to the upper bound \( 1 - \left( \frac{1}{p} \right)^n \), just as in the case \( p = 2 \). Again, numerical results are presented in section 4.3.3.

In this section we have shown that the expected number of edges which remain between stages decreases as the distance from the final stage increases. We developed techniques for generating recurrence relations which we expect to retain between successive stages. The first six recurrence relations were given, for the number of processors ranging from 2 to 7.

4.3.3. Numerical results

Each of the recurrence formulas for the pruning probabilities \( q_i \) has its highest order term of order \( p \), and \( p \geq 2 \). This makes further investigation of the space complexity for the dynamic programming algorithm by analytical means extremely difficult. In this section we examine the algorithm storage requirements numerically, basing our calculations on the recurrence formulas of Fig. 4.8.

The probabilities \( q_i \) are independent of \( n \), except that for each \( p \), \( q_i \) has a maximum value which is a function of \( n \). Let us define \( i_p(n) \) to be the smallest value of \( i \) for which
Recurrence formulas for determining $q_{i-1}$ for the number of processors ranging from 2 to 7. $q_i$ is the probability that a node of $s_i$ will have no edges to nodes in $s_{i+1}$ after pruning is completed.
<table>
<thead>
<tr>
<th>n</th>
<th>p=2</th>
<th>p=3</th>
<th>p=4</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>11</td>
<td>23</td>
<td>39</td>
</tr>
<tr>
<td>3</td>
<td>26</td>
<td>73</td>
<td>166</td>
</tr>
<tr>
<td>4</td>
<td>58</td>
<td>237</td>
<td>677</td>
</tr>
<tr>
<td>5</td>
<td>121</td>
<td>723</td>
<td>2725</td>
</tr>
<tr>
<td>6</td>
<td>248</td>
<td>2180</td>
<td>10916</td>
</tr>
<tr>
<td>7</td>
<td>503</td>
<td>6553</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1015</td>
<td>19675</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>2038</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>4085</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>8181</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>16372</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2
<table>
<thead>
<tr>
<th>p</th>
<th>n</th>
<th>E[edges left]</th>
<th>E[edges pruned]</th>
<th>E[edges left]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>total edges</td>
<td>nπn</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>21</td>
<td>0.76145</td>
<td>2.624</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>63</td>
<td>0.84895</td>
<td>2.618</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>170</td>
<td>0.90840</td>
<td>2.656</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>427</td>
<td>0.94486</td>
<td>2.669</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>1030</td>
<td>0.96755</td>
<td>2.682</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>2414</td>
<td>0.98125</td>
<td>2.694</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>5538</td>
<td>0.98934</td>
<td>2.704</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>12493</td>
<td>0.99401</td>
<td>2.711</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>27825</td>
<td>0.99667</td>
<td>2.717</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>61328</td>
<td>0.99817</td>
<td>2.722</td>
</tr>
</tbody>
</table>

| 3  | 2  | 58           | 0.90587        | 3.247        |
|    | 3  | 263          | 0.95733        | 3.243        |
|    | 4  | 1054         | 0.98170        | 3.252        |
|    | 5  | 3963         | 0.99248        | 3.261        |
|    | 6  | 14288        | 0.99700        | 3.257        |
|    | 7  | 50071        | 0.99884        | 3.271        |
|    | 8  | 171837       | 0.99956        | 3.274        |

| 4  | 2  | 118          | 0.95266        | 3.692        |
|    | 3  | 707          | 0.98336        | 3.632        |
|    | 4  | 3773         | 0.99456        | 3.684        |
|    | 5  | 18277        | 0.99831        | 3.687        |
|    | 6  | 90646        | 0.99949        | 3.688        |

Table 4.3

Expected space complexity for pruned assignment tree, from \( s_{m-i_p(n)} \) to \( s_m \).
\[ q_{m-i} > 1 - \left( \frac{1}{p} \right)^n. \] Table 4.2 gives \( i_p(n) \) for \( p = 2, 3, \) and \( 4, \) and for varying values of \( n. \) It is apparent that \( i_p(n) \) increases as \( p^n, \) and for even moderate values of \( p \) and \( n \) becomes quite large, much larger than we would expect \( m \) to be.

Nevertheless, we would like to know the total expected number of edges which will remain in stages \( s_{m-i_p(n)} \) through \( s_m \) after pruning. (In stages \( s_1 \) through \( s_{m-(i_p(n)+1)} \) there will be exactly one edge remaining in each stage, in expectation.) Table 4.3 summarizes the results of these calculations. For each of the indicated values of \( p \) and \( n, \) we give the expected number of edges left in \( s_{m-i_p(n)} \) through \( s_m, \) together with the ratio of the expected number of edges pruned to the number of edges in the unpruned assignment tree. The latter figures clearly the improvement in average storage requirements that pruning the assignment tree offers.

The third column of Table 4.3 is very suggestive concerning what the expected space complexity of the dynamic programming algorithm with pruning might be for values of \( p \) and \( n \) not included in the table. For each pair of values \( p \) and \( n \) we have divided the expected number of edges which remain by \( np^n. \) The somewhat unexpected result is that the ratio appears to rapidly converge to a constant for fixed \( p \) as \( n \) increases, indicating that for a given value of \( p, \) the expected space complexity is \( O(np^n). \) Although not explicitly included in Table 4.3, the behavior of the average number of edges remaining for fixed \( n \) as \( p \) increases
was also examined. Since only three values of $p$ were studied, any inferences we might draw are suspect; however, it does appear that for fixed $n$, the average space complexity may also be bounded by $O(np^n)$. If this is true, and if $i_p(n)$ is $O(p^n)$, then the ratio of expected complexity to worst case complexity is $O(n/m)$. We emphasize again that this is conjectural and further analysis is needed in this area.

The discussion of the above paragraph hinges upon $i_p(n)$ being less than $m$. As we have seen, $i_p(n)$ appears to increase as $p^n$ for a given $p$ with increasing $n$. Clearly many if not most problems we might wish to consider will be for module instance execution strings of much shorter length, especially for systems having more than two processors. The above results on space complexity provide a lower bound on the storage requirements of the algorithm for arbitrary $m$. Module instance execution strings shorter than $i_p(n)$ will require more space per stage, because $q_{m-i}$ is a monotonically increasing function of $i$. However, $q_{m-i}$ increases very rapidly when $i$ is small. Recall that $q_{m-i}$ is the ratio of expected number of edges pruned to the number of edges before pruning for $s_{m-(i+1)}$. Figure 4.9 shows how rapidly $q_{m-i}$ approaches 1 initially. The abscissa $i$ is plotted on a logarithmic scale. As $i$ increases from 1, the curves for $p = 2$ and $p = 7$ are almost linear out to approximately 10, testifying to the nearly exponential behavior of $q_{m-i}$ in this region. Even for $p = 2$, the probability that any given edge will be deleted
from $s_{m-34}$ is above 90%, and by $i = 100$, the probability is above 96%. The curves for $p = 3$ through $p = 6$ have similar shapes and lie between the two shown. Fig. 4.9 indicates that for even relatively short module instance execution strings (short with respect to $p^n$), pruning will have a tremendous impact on the expected space complexity of the algorithm.

4.3.4. *Summary of expected space complexity*

The dynamic programming algorithm has been shown to have an average space complexity that is considerably less than worst case if we alter the algorithm slightly to dynamically prune useless edges from the assignment tree being constructed. As more and more edges are pruned from successive stages as we proceed back towards $s_0$, we eventually reach the point at which we expect to find only a single path from the start node. Probability recurrence relations for several different numbers of processors have been derived and used to numerically investigate the expected space complexity. Asymptotically, for any fixed $p$, the average storage needs for the algorithm appear to be $O(m + np^n)$ rather than the worst case complexity of $O(mp^n)$. This requires that $m > i_p(n)$, and $i_p(n)$ increases as $p^n$ for the values of $p$ and $n$ studied. Even for shorter module instance execution strings, however, the expected space saving is very great, due to the rapid increase of $q_{m-i}$ as $i$ increases from 1.
4.4. Summary of dynamic programming algorithm

The algorithm presented in this chapter uses a dynamic programming approach which is equivalent to a shortest path algorithm for problems in which all paths to any node have the same number of edges. This algorithm is completely general, allowing asymmetric reassignment and communication costs as well as arbitrary number of processors. The price of generality is unpleasantly large upper bounds on space and time complexities. However, average space complexity has been shown to be significantly less than worst case. In the next chapter we will examine a modification of the algorithm which has the potential of substantially reducing expected time complexity as well.
CHAPTER 5
MODIFIED DYNAMIC PROGRAMMING ALGORITHM

5.1. Introduction

In section 3.4.3., we showed that the dynamic flow algorithm time complexity is bounded by $O(m^{2-1/3})$ but independent of $n$, for the two processor case. The dynamic programming algorithm, on the other hand, is linear in $m$ but exponential in $n$. For small values of $n$ this is perhaps acceptable, but the algorithm quickly becomes unwieldy as $n$ increases. Furthermore, the dynamic programming algorithm has an expected time complexity that is equal to its worst case complexity; since for each module instance we are essentially considering every possible assignment for the most recent instances of the $n$ modules. The number of possible assignments or states is eventually $p^n$ for every stage.

It has been shown (APE77) (APE78) that for some classes of flow networks the expected number of phases executed by the Dinic-Karzanov algorithm is bounded by a constant, leading to an overall expected time complexity of $O(|V|^2)$, where $|V|$ is the number of vertices. Apers also conjectures that this holds true for other classes of flow networks as well. Results from a recent implementation of the Dinic-Karzanov algorithm appear to lend support to this conjecture (FOL78). The Galil max-flow min-cut algorithm uses the same layered network approach, and differs only in the execution within a phase. Consequently, the overall expected time complexity of the dynamic processor
flow graph algorithm using Galil's algorithm is likely to be only $O(m^{4/3})$.

Can we in some way modify the dynamic programming algorithm to allow improvements in expected time complexity over worst case? The answer is yes, if we assume that there is a certain kind of "predictability" in program behavior. The modified algorithm uses "clustering" of module executions to advantage by considering in each stage only those assignment states in which modules in the "cluster" have different assignments. The degree of improvement in the expected time complexity of the algorithm will depend upon the size and persistence of the clusters. We will show that the modified dynamic programming algorithm has an expected time complexity which can asymptotically approach $O(mp^3)$, independent of n.

The remainder of the chapter is presented in four principal sections. In section 5.2 we illustrate the general principles behind the algorithm by means of an example. Section 5.3 gives a somewhat informal description of the algorithm and analysis of its expected performance. We attempt to model the rather vague notions of "predictability" and "clustering" in a precise manner in section 5.4., and use this model to analyze characteristics of program behavior in which we have an interest. Finally, in section 5.5. we show how to optimize the modified dynamic programming algorithm, based on our analysis of the model in the previous section, in order to obtain our objective of minimizing expected time complexity.
5.2. An example

In this and ensuing sections we will rely heavily on the description of the dynamic programming approach as a shortest path algorithm to make the discussion as succinct and clear as possible. This is particularly important since we will not give a formal proof of algorithm correctness. The reader should, however, be able to convince himself of correctness by relating the modified algorithm to the original algorithm in chapter 4.

The modified algorithm's success relies on the fact that in dealing with the (imaginary) dynamic assignment graph, the original algorithm often duplicates much of its efforts in finding shortest paths. We illustrate the point using the portion of a dynamic assignment graph shown in Fig. 5.1. Shown are five successive stages for a 2-processor, 3 module problem. As before, the upper and lower edges from a node to the next stage mean assignments to processor 1 and processor 2, respectively.

Now let us assume that we have completed the computations for \( s_{i-1} \) and are ready to begin \( s_i \). We will begin by considering the portion of the graph outlined by the broken lines. Fig. 5.2 is an expanded picture of this portion. Our ultimate goal will be to find the edges along the shortest paths to nodes in \( s_{i-2} \). For now, suppose that we do not know the values of any of the nodes in \( s_{i-1} \). Then if we find the shortest paths from all the nodes of \( s_{i-1} \) to each node of \( s_{i+2} \) we can select the proper path if we know the value of the nodes in \( s_{i-1} \).
Figure 5.1.

Five stages of a two processor, three module dynamic assignment graph.
\[ s_i : A_j \quad s_{i+1} : B_k \quad s_{i+2} : A_{j+1} \]

\[
\begin{array}{c}
\text{a}_{i-1} & \text{a}_i & \text{a}_{i+1} & \text{a}_{i+2} \\
\text{b}_{i-1} & \text{b}_i & \text{b}_{i+1} & \text{b}_{i+2} \\
\text{c}_{i-1} & \text{c}_i & \text{c}_{i+1} & \text{c}_{i+2} \\
\text{d}_{i-1} & \text{d}_i & \text{d}_{i+1} & \text{d}_{i+2}
\end{array}
\]

**Figure 5.2.**

a: \( A - P_1 \quad B - P_1 \)
b: \( A - P_2 \quad B - P_1 \)
c: \( A - P_1 \quad B - P_2 \)
d: \( A - P_2 \quad B - P_2 \)

The state of each node includes the same assignment for C: \( C - P_1 \).
As usual, we find the shortest path from $s_{i-1}$ to $s_{i+2}$ in the subgraph by finding first all shortest paths to $s_i$, then to $s_{i+1}$, and finally to $s_{i+2}$. Each node of $s_i$ has paths from exactly two nodes of $s_{i-1}$. Since the p-sets become shuffled in $s_{i+1}$ with respect to the p-sets of $s_i$, each node in $s_{i+1}$ will have exactly four paths to it from $s_{i-1}$. There will be a path from each of $a_{i-1}$, $b_{i-1}$, $c_{i-1}$, and $d_{i-1}$.

In $s_{i+2}$, we will have a choice of two paths from each node of $s_{i-1}$ to each node of $s_{i+2}$, and we choose the shorter of the two.

The crucial point to note is that at no time in any of the shortest path calculations for the subgraph of Fig. 5.1 does the assignment of $C$ have any effect. The previous assignment of $C$ becomes important only when we wish to select the shortest paths from $s_0$ to $a_{i+2}$, $b_{i+2}$, $c_{i+2}$, and $d_{i+2}$. The shortest path to $a_{i+2}$ from $s_0$ will be one of four possible paths, the four shortest paths from $s_0$ to $a_{i+2}$ which pass through $a_{i-1}$, $b_{i-1}$, $c_{i-1}$, and $d_{i-1}$. The shortest path from $s_0$ to $a_{i+2}$ through $a_{i-1}$ is the shortest path from $s_0$ to $a_{i-1}$ followed by the shortest path from $a_{i-1}$ to $a_{i+2}$. The shortest path from $s_0$ to $a_{i+2}$ through $b_{i-1}$ is the shortest path from $s_0$ to $b_{i-1}$ followed by the shortest path from $b_{i-1}$ to $a_{i+2}$, and so forth. The same holds true for all other nodes of $s_{i+2}$ in the subgraph of Fig. 5.2.

Now let us consider the subgraph consisting of those nodes and edges in $s_{i-1}$ through $s_{i+2}$ in the dynamic assignment graph which are not included in Fig. 5.2. We illustrate this subgraph
in Fig. 5.3. If we attempt to find all shortest paths from \( s_{i-1} \) to \( s_{i+2} \) in this subgraph, we once again discover that the previous assignment of \( C \) is of no importance. But the subgraphs of Figs. 4.2 and 4.3 are equivalent. The shortest path from \( a_{i-1} \) to \( a_{i+2} \) has the same value and same module assignments as the shortest path from \( e_{i-1} \) to \( e_{i+2} \), etc. Since this is so, we may do our calculations of shortest paths from \( s_{i-1} \) to \( s_{i+2} \) on one representative subgraph, and then apply the results, together with our preknowledge of shortest paths from \( s_0 \) to \( s_{i-1} \), to find all shortest paths from \( s_0 \) to \( s_{i+1} \). Thus, we can avoid considerable duplication of work.

Suppose the module instances of \( s_{i+3} \) through \( s_{i+r} \) were all instances of modules A and B. Then the subgraph of Fig. 5.2 could simply be extended to \( s_{i+r} \). Each stage in the subgraph would still contain only four nodes, and for each node of \( s_{i+2} \) through \( s_{i+r} \), we would choose one of only two possible shortest paths from each node of \( s_{i-1} \).

What if the program consisted of more than three modules? That is, what if the subgraph of Fig. 5.1 were a subgraph of a dynamic assignment graph for a program with \( n > 3 \) modules? Then the dynamic assignment graph would have \( 2^{n-2} \) subgraphs of \( s_{i-1} \) through \( s_{i+2} \) identical to Figs. 5.2 or 5.3. There would be one of these for each of the \( 2^{n-2} \) possible combinations of assignments for the \( n-2 \) modules not occurring in \( s_i \) through \( s_{i+2} \).
Figure 5.3.

\[ e: \text{A} - P_1 \quad \text{g: A} - P_1 \]
\[ B - P_1 \quad \quad B - P_2 \]

\[ f: \text{A} - P_2 \quad \quad \text{h: A} - P_2 \]
\[ B - P_1 \quad \quad B - P_2 \]

The state of each node includes the same assignment for C: C - P_2.
Now we really begin to see the advantage of following the procedure outlined above. Increasing the number of modules has no effect on the complexity of a representative subgraph such as Fig. 5.2. Consequently, the only increase in overall algorithm complexity due to an increase in the number of modules is in the number of times we form a shortest path from $s_0$ to $s_{i+2}$ by combining a shortest path from $s_0$ to $s_{i-1}$ with a shortest path in the representative subgraph from $s_{i-1}$ to $s_{i+2}$. Since there are $2^n$ nodes in $s_{i-1}$ and $s_{i+2}$, and $2^2$ possible paths from nodes of $s_{i-1}$ to a single node of $s_{i+2}$, the overall complexity for this step is $2^{n+2}$, somewhat more than the complexity $2^{n+1}$ for each stage in the unmodified dynamic programming algorithm. However, the complexity of dealing with the representative subgraph can be drastically less than dealing with the same stages of the complete dynamic assignment graph.

It should be noted the validity of creating a single shortest path by simply joining two smaller shorter paths whose combined lengths are minimal is entirely dependent on the fact that $B_{k-1}$, the $(i-1)$st module instance, is an instance of a module which also occurs in $s_i$ through $s_{i+2}$. If it were not B but another module, for instance C, then the shortest paths from nodes of $s_{i-1}$ to nodes of $s_{i+2}$ in the representative subgraph would depend on the assignment of C. However, each of the subgraphs which are supposedly identical to the representative subgraph assume a different assignment for C. Hence, to be truly representative,
the subgraph must include an instance of the module which immediately precedes the subgraph. As we shall show in a later section, the modified algorithm will be optimized at a point at which the probability that a module immediately preceding a representative subgraph has an instance in the subgraph is extremely high as \( n \) gets large.

We will generalize the above discussion for arbitrary values of \( p \) and \( n \) in the next section. We will also give a general formula for determining the expected complexity of the modified dynamic programming algorithm based on the expected size of representative subgraphs used by the algorithm.

5.3. The modified dynamic programming algorithm-general case

5.3.1. Definitions

The ideas of the previous section for the two-processor problem can be extended in a straightforward manner to enable us to incorporate arbitrary numbers of processors and modules. We will begin by defining module clusters and representative subgraphs. A module cluster set or module cluster is a subset of modules in the program. A \( j \)-module cluster is a module cluster of size \( j \). We define a clustered sequence with respect to a module cluster as being a set of successive module instances in which every instance is an instance of a module in the cluster and every module in the cluster has an instance in the sequence. Every clustered sequence with respect to a \( j \)-module cluster \( S \)
contains a clustered sequence with respect to a \((j-1)\)-module
cluster T\&S for \(j > 1\).

A representative subgraph for stages \(s_{i+1}\) to \(s_{i+r}\) of a
dynamic assignment graph is itself defined to be a dynamic
assignment graph. We treat the \(r\) module instances of \(s_{i+1}\)
through \(s_{i+r}\) as a clustered sequence, with its associated \(j\)-module
cluster C. There is a node in each stage of the representative
subgraph for every possible combination of assignments of the
\(j\) modules in the module cluster. This means there are \(p^j\) nodes
in each stage. We have an additional initial stage with the
same number of nodes. We will call these stages by the same
names as the corresponding stages in the complete dynamic assignment
graph, \(s_i\) through \(s_{i+r}\). It should be apparent from the example
of the previous section that in the complete dynamic assignment
graph, there will be \(p^{n-j}\) subgraphs which are identical to each
other and to the representative subgraph of those stages. Each
one of these \(p^{n-j}\) subgraphs corresponds to a different combination
of assignments for the \(n-j\) modules not in the module cluster
defined by the cluster sequence of \(s_{i+1}\) through \(s_{i+r}\).

5.3.2. The modified algorithm

The modified algorithm will proceed as follows. Assume
that we have constructed the dynamic assignment tree from \(s_0\) to
\(s_i\), giving us the shortest path from \(s_0\) to each node of \(s_i\). Also
assume that we have in some way determined a value \(j\) for a module
cluster set size, and we have scanned the string of module instances
from \( s_{i+1} \) forward to determine the largest clustered sequence beginning at the \((i+1)\)st module instance which contains exactly \( j \) distinct modules. Let the last instance in this sequence be \( i + r \).

Now construct the representative subgraph for \( s_i \) through \( s_{i+r} \) as described above. Find all shortest paths from nodes of \( s_i \) to nodes of \( s_{i+r} \) in the representative subgraph. There will be \( p^j \) paths to each of the \( p^j \) nodes of \( s_{i+r} \). Since the representative subgraph is identical to the \( p^{n-j} \) subgraphs of the complete assignment graph as far as path lengths are concerned, we may then use the shortest paths of the representative subgraph to extend the assignment tree to \( s_{i+r} \).

Suppose we wish to find the shortest path to a node \( W \) of \( s_{i+r} \). First determine the assignments of the \( p^{n-j} \) modules not in \( s_{i+1} \) through \( s_{i+r} \) which are part of the state of \( W \). There are \( p^j \) nodes of \( s_i \) which have the same assignments for the \( n-j \) modules as \( W \) does. Furthermore, the assignments of these modules are for the same instances of the \( n-j \) modules, since none of the \( n-j \) modules were executed in the \((i+1)\)st through \((i+r)\)th module instances. Then the shortest path to \( W \) from \( S \) must pass through one of these \( p^j \) nodes of \( s_i \). The same is true for every other module in \( s_{i+r} \) which have the same assignments for the \( n-j \) modules as \( W \) does. Consequently, the state of \( W \) defines a subgraph in \( s_i \) through \( s_{i+r} \) which is identical to the representative subgraph.
The shortest path from \( s_0 \) to \( W \) through a node \( U \) of \( s_i \) (assuming \( U \) and \( W \) belong to the same subgraph) is the shortest path from \( s_0 \) to \( U \) coupled with the shortest path from \( U \) to \( W \). The shortest path from \( U \) to \( W \) can be determined from the representative subgraph as being the shortest path from \( U' \) in \( s_i \) to \( W' \) in \( s_{i+r} \), where \( U' \) and \( W' \) are nodes of the representative subgraph which have states in which the assignments of the \( j \) modules in the cluster are identical to the assignments of the same \( j \) modules in the states of \( U \) and \( W \), respectively. The shortest path to \( W \) from \( s_i \) is selected as the shortest of all the \( p^j \) possible paths to \( W \) obtained in this manner, and this path is added to the assignment tree. We repeat this procedure for all nodes of \( s_{i+r} \) in the complete assignment graph. At the conclusion, we have extended the assignment tree to \( s_{i+r} \).

**Theorem:** The assignment tree constructed according to the above procedure consists of the shortest paths from \( s_0 \) to all nodes of \( s_{i+r} \).

**Proof:** The proof of this theorem is a formalization of the arguments made above in the description of the algorithm.

Once again we have implicitly assumed that the module of \( s_i \) is in the module cluster corresponding to the clustered sequence of \( s_{i+1} \) through \( s_{i+r} \). In section 5.5, we will attempt to justify such an assumption by showing that for the values of \( j \) which the algorithm will use, the probability of this not being true
is extremely small. For now we will make the following policy. If the module of $s_i$ is not in the $j$-module cluster for $s_{i+1}$ through $s_{i+k}$, extend the assignment tree to $s_{i+1}$ as per the original dynamic programming algorithm. We believe that the probability of being forced to do this more than once or twice in a row is vanishingly small, and consequently such a policy should have very little effect on the overall expected time complexity.

5.3.3. General expression for expected complexity

In deriving an expression for expected algorithm complexity, we will assume that $j$, the module cluster size, is fixed and that each clustered sequence contains an instance of the module whose execution immediately precedes the sequence. Let $E[T_s | j]$ be the expected time complexity to find all shortest paths through a representative subgraph for an arbitrary clustered sequence. Also, let us define $\sigma_i(n)$ to be the expected length of a maximal length clustered sequence containing exactly $i$ distinct modules. Furthermore, we define

$$\mu_{i, i+1} = \sigma_i(n) - \sigma_{i-1}(n)$$

to be the difference in expected lengths of a maximal length clustered sequence of $i + 1$ modules and one containing one less module.

First consider nodes in the representative subgraph belonging to the first $\sigma_1(n)$ stages, starting at $s_a$. We expect there to
be only one module in the first $\sigma_1(n)$ instances. Each node can be reached from every node in its same p-set of $s_a$, but no others. This means that every node will have $p^2$ shortest paths converging on it, and we select $p$ paths at each node in each stage. This will require $p^{j+2}$ choices at each stage, so the complexity out to $s_{a+\sigma_1}(n)$ is $\sigma_1(n) \cdot p^{j+2} = \nu_{1,2} \cdot p^{j+2}$.

All nodes in $s_{a+\sigma_1}(n) + 1$ to $s_{a+\sigma_2}(n)$ may be reached from $p^2$ nodes of $s_a$, since there will be two different modules in the clustered sequence from the $(a+1)$st module instance to the $(a+\sigma_2(n))$th instance. The first occurrence of the second module will be at $s_{a+\sigma_1}(n) + 1$. The $p$-sets of this stage will be formed with respect to the second module, while the $p$-sets of the previous stage are with respect to the first module. Each node of $s_{a+\sigma_1}(n) + 1$ is reachable from $p$ nodes of $s_{a+\sigma_1}(n)$, each of which in turn is reachable from $p$ different nodes in $s_a$. The complexity for the portion of the representative subgraph from $s_{a+\sigma_1}(n)$ to $s_{a+\sigma_2}(n)$ is therefore $\nu_{2,3} \cdot p^{j+3}$.

We can quickly generalize the above discussion. For the part of the representative subgraph from $s_{a+\sigma_i}(n)$ to $s_{a+\sigma_{i+1}}(n)$, there are $\nu_{i,i+1}$ stages. Each node in each of these stages will be reachable along $p$ paths from each of the $p^i$ nodes of $\sigma_a$. The expected complexity for this part of the subgraph is $\nu_{i,i+1} \cdot p^{j+i+1}$.
Then
\[ E[T_s | j] = \sum_{i=1}^{i} \nu_{i,i+1} \cdot p^{j+i+1} \]

Let \( E[T | j] \) be the expected complexity for the entire algorithm. If there are \( m \) module instances and \( j < n \), then we compute \( \frac{m}{\sigma_j(n)} \) representative subgraphs, on the average. At the conclusion of the calculations for each subgraph, the extension of the assignment tree will require \( p^{n+j} \) calculations, since each node in the last stage of the representative subgraph will have a shortest path to it from each of the \( p^j \) nodes in the initial stage of the subgraph. The overall expected complexity of the algorithm is therefore
\[ E[T | j] = \frac{m}{\sigma_j(n)} \left[ E[T_s | j] + p^{n+j} \right]. \]

Section 5.4. will consider a model for program behavior which will allow us to quantify \( E[T_s | j] \), \( \sigma_j(n) \), and consequently \( E[T | j] \) for any values of \( j \) and the given problem parameters \( p \), \( n \), and \( m \). We will do this by introducing another parameter, one which is related to the "predictability" or module clustering tendency of the program. In section 5.5., we will use the analysis of section 5.4. to allow us to optimize the modified algorithm by selecting the value of \( j \) which minimizes the expected complexity for a given instance of the dynamic assignment problem.
5.4. Module instance execution models

5.4.1. Basis for model

In order to determine bounds on expected time complexity for the dynamic programming algorithms, it is necessary to model the sequence of module instance executions in a manner which can be related to actual program behavior in a straightforward way and can be dealt with using suitable analytical tools. The model we have chosen relies on the intuitive notion that modules which have been executed most recently in the past are more likely to be executed in the near future. A well-known and well understood model which will allow us to observe the effects of such behavior is the least recently used stack model, or LRUUSM.

The LRUUSM has been used with a great deal of success in studying page reference behavior in paged memory systems, especially in the area of predicting comparative performance characteristics for various demand paging strategies (SPI77). Its success is due to the observation that page references are generally clustered in time. At any given point of the program execution there is a "most favored" set of pages being referenced, and the constitution and size of this set changes slowly with time. One of the most successful paging strategies, in fact, is the LRU strategy, which is optimum for the LRUUSM when fixed sized memory partitions are being employed.

It is our conjecture that module instance execution behavior has a similar characteristic. In a well structured program, control
flow should be hierarchical in nature, especially for the application-oriented programs with which we are primarily concerned. The LRU SM will be used to model the string of module executions. It will be shown that the modified dynamic programming algorithm is capable of significantly reduced average computation time under this model. The following sections describe the LRU SM and examine the modified algorithm's expected time complexity under this model, for both the biased and unbiased cases.

5.4.2. The LRU SM

In the LRU SM, we maintain a stack of depth \( n \), where \( n \) is, as before, the number of modules in the program. Let \( d(k) \) be the \( k \)th result in a string of \( m \) independent trials. The sequence \( d(1)d(2)\ldots d(m) \) is called the stack distance string. The outcome of the \( k \)th trial is one of \( n \) possible results, and the probability of one particular outcome \( i \) on the \( k \)th trial is given by

\[
\Pr[d(k) = i] = a_i, \text{ for all } k, 1 \leq i \leq n.
\]

\( \{a_i\} \) is the set of stationary stack distance probabilities. We will assume \( a_i > 0 \) for all \( i, 1 \leq i \leq n \).

The stack at the \( k \)th module execution is represented by the \( n \)-vector \( \tilde{s}(k) \). \( s_1(k) \), the first element of the vector, is defined to be the top of the stack, and \( s_n(k) \) is the bottom of the stack. Assume that \( d(k+1) = i \). Then \( \tilde{s}(k+1) \) is defined by:
(1) \( s_1(k+1) = s_1(k) \)

(2) \( s_j(k+1) = s_{j-1}(k), \ 2 \leq j \leq i-1 \)

(3) \( s_j(k+1) = s_j(k), \ i+1 \leq j \leq n \)

The manipulation of the LRUSM is quite simple. If the (k+1)st trial has an outcome of \( i \), we move the \( i \)th element of the stack to the top position of the stack, and the 1st through (i-1)st elements in the stack are all shifted down one position. Updating the LRU stack is illustrated in Fig. 5.4.

Elements in the stack are module names, with each of the \( n \) modules having an entry in the stack. If we define \( e(k) \) to be the name of the module for the \( k \)th module instance in the program execution, then we can relate the sequence of module execution to the LRUSM by

\[
\begin{align*}
    s_1(k) &= e(k) .
\end{align*}
\]

Thus, the series of trials whose outcomes are given by \( d(1)d(2)...d(m) \), together with the initial stack state \( s(0) \), uniquely determines the sequence of module executions \( e(1)e(2)...e(m) \). The initial stack state is of no interest to us, and without loss of generality we may take \( s_i(0) = i, \ 1 \leq i \leq n \). We assume a one-to-one mapping from the set of integers \( \{1, \ldots, n\} \) to the set of program module names.

We are interested in one feature of the LRUSM in particular. We wish to know how long (how many module executions) before the
Figure 5.4.

Updating the LRU SM following the \( (k+1) \)st trial, which has an outcome of \( i \).
jth distinct module is first executed. This is equivalent to finding the largest value of k for which the modules in $s_1(k)$ through $s_{j-1}(k)$ have each been executed at least once, and no module in $s_j(k)$ through $s_n(k)$ has been executed yet.

We begin by constructing the imbedded Markov chain for the number of distinct modules executed thus far before each module instance. The state transition diagram for the Markov chain is shown in Fig. 5.5. State 0 is the initial state and corresponds to the fact that we have executed 0 modules as we start. Upon the execution of the first module instance, the chain has only one possible transition, to state 1. For any $i$, $1 \leq i \leq n$, we define $h(i)$ and $m(i)$ as follows:

$$h(i) = \sum_{j=1}^{i} a_j$$

$$m(i) = \sum_{j=i+1}^{n} a_j = 1 - h(i)$$

$h(i)$ is the probability that a trial to select a stack distance will have an outcome among the first $i$ stack positions; it is the probability of a hit in the top $i$ stack elements. $m(i)$ is the probability of a miss in the first $i$ stack positions.

The transitions for the imbedded Markov chained are easy to derive. For any state $i$, there are two possible transitions before executing a module instance. If the module which is to be executed is among the top $i$ modules of the stack, then the
The Markov chain transition graph for LRU stack growth if the stack is initially empty. $h(i)$ is the probability of a hit (selecting a module) in the first $i$ positions of the stack. $m(i) = 1 - h(i)$ is the probability of a miss.
next state transition returns us to state $i$. This will occur with probability $h(i)$. If the module is among the bottom $n-i$ modules of the stack, it has not been executed before. The chance for this to occur is $m(i)$, and we make a transition to state $i+1$ if it does. Obviously, $h(0) = 0$ and $m(n) = 0$. Also, all states of the chain are transition states except for state $n$ which is the single absorbing state.

The resulting chain is extremely simple. Now for any state $i+1$ let us calculate $\nu_{0,i+1}$, the expected first passage time from state 0 to state $i+1$. We begin with the system of equations

\begin{align}
\nu_{0,i+1} &= 1 + \nu_{1,i+1} \\
\nu_{1,i+1} &= 1 + h(1) \cdot \nu_{1,i+1} + m(1) \cdot \nu_{2,i+1} \\
\vdots \\
\nu_{j,i+1} &= 1 + h(j) \cdot \nu_{j,i+1} + m(j) \cdot \nu_{j+1,i+1} \\
\vdots \\
\nu_{i-1,i+1} &= 1 + h(i-1) \cdot \nu_{i-1,i+1} + m(i-1) \cdot \nu_{i,i+1} \\
\nu_{i,i+1} &= 1 + h(i) \cdot \nu_{i,i+1}
\end{align}

(5.1) 
(5.2) 
(5.3) 
(5.4) 
(5.5)

From equation (5.5) we see that

$$\nu_{i,i+1} = \frac{1}{1-h(i)} = \frac{1}{m(i)}.$$
For arbitrary \( j, 1 < j < i \), eqn. (5.3) shows

\[
\nu_{j,i+1} = \frac{1}{1-h(j)} \left[ 1 + m(j) \cdot \nu_{j+1,i+2} \right]
\]

\[= \frac{1}{m(j)} + \nu_{j+1,i} \]

By starting at eqn. (5.4), substituting in \( \nu_{i,i+1} = \frac{1}{m(i)} \), and working our way back up the sequence of equations, we have

\[
\nu_{j,i+1} = \sum_{t=j}^{i} \frac{1}{m(t)}
\]

In particular,

\[
\nu_{1,i+1} = \sum_{t=1}^{i} \frac{1}{m(t)}
\]

As a result,

\[
\nu_{0,i+1} = 1 + \sum_{t=1}^{i} \frac{1}{m(t)}
\]

\( \nu_{0,i+1} \) is very nearly the result we want. However, \( \nu_{0,i+1} \) is the expected number of module executions until we execute the \((i+1)st\) distinct module, including the execution of the \((i+1)st\) module itself. We actually want to know \( \sigma_{i}(n) \), the expected number of executions before we encounter the \((i+1)st\) new module. This is obviously \( \nu_{0,i+1} - 1 \), or

\[
\sigma_{i}(n) = \sum_{j=1}^{i} \frac{1}{m(j)} . \quad (5.6)
\]
\( \sigma_i(n) \) is identical to the \( \sigma_j(n) \) of section 5.3., for \( i=j \). We will use this general expression for \( \sigma_i(n) \) to determine specific expressions for two cases of stack distance probabilities.

5.4.3. **LRUSM stack distance probability distributions**

5.4.3.1. **Uniform distribution**

In this section we consider the set \( \{ a_i \} \) to have a discrete uniform random distribution.

\[
a_i = \frac{1}{n}, \quad 1 \leq i \leq n.
\]

Thus, the probability of a particular module being executed next at any point in the program execution is also \( \frac{1}{n} \), since no positions in the stack are favored. One might ask why we bother with a stack at all in this case. The answer is that doing so enables us to make use of the analysis in the previous section in finding an exact expression for \( \sigma_i(n) \).

The discrete uniform random distribution represents a kind of "worst case" situation for the LRUSM. In using this to model program behavior, we are saying that we expect the past history of the program to have absolutely no correlation to its present and future behavior. No matter which module is currently executing, the next module to be executed is equally likely to be any of the modules in the program. Even the most malicious programmer would have difficulty writing a program with this characteristic. Consequently, we will use the LRUSM with this uniform distribution.
of stack probabilities to provide us with an upper bound on
the expected time complexity of the modified algorithm.

Under this distribution,

\[ m(j) = \sum_{i=j+1}^{n} \frac{1}{n} = \frac{1}{n} \cdot (n-j) \]

Therefore,

\[ \sigma_j(n) = \sum_{i=1}^{j} \frac{1}{m(i)} \]

\[ = n \sum_{i=1}^{j} \frac{1}{n-i} \]

\[ = n \left[ \sum_{i=1}^{n-1} \frac{1}{i} - \sum_{i=1}^{n-j-1} \frac{1}{i} \right] \]

\[ \approx n \ln \left( \frac{n-1}{n-j-1} \right) \]

for \( n \) sufficiently larger than \( j \). \( \sigma_j(n) \) is obviously a very
slowly growing function of \( n \), with a maximum near \( n \cdot \ln[n] \) for
large \( n \).

5.4.3.2. Uniform stack bias

The set of stack distance probabilities in the previous
section minimally satisfies the "strong locality condition" (SPI77)
which requires that \( a_1 \geq a_2 \geq \ldots \geq a_n \). Here we are interested
in probabilities which also (with one exception) satisfy this
condition, but never with equality. Since this class of stack
distance probability distributions which have the strong locality
condition is too general to be of much use, we will restrict our
attention to those distributions which show a constant ratio between successive stack distance probabilities.

\[ \frac{a_i}{a_{i+1}} = b, \text{ a constant } > 1 \text{ for all } i, 1 \leq i < n-1. \]

We will call b the stack bias, and a LRUSM which has a uniform bias will be termed a constant bias LRUSM, or CLRUSM. Then for a CLRUSM,

\[ \sum_{i=1}^{n} a_i = a_1 + \frac{1}{b} \cdot a_1 + \left(\frac{1}{b}\right)^2 \cdot a_1 + \ldots + \left(\frac{1}{b}\right)^{n-2} \cdot a_1 + a_n \]

\[ = a_1 \left[ \sum_{i=0}^{n-2} \left(\frac{1}{b}\right)^i \right] + a_n \]

\[ = a_1 \left[ \frac{1 - \left(\frac{1}{b}\right)^{n-1}}{1 - \frac{1}{b}} \right] + a_n \]

\[ = a_1 \cdot \frac{b}{b-1} \left[ 1 - \left(\frac{1}{b}\right)^{n-1} \right] + a_n \]

Ideally, as \( n \to \infty \), \( a_n \to 0 \).

\[ \therefore \sum_{i=1}^{n} a_i \xrightarrow{n \to \infty} a_1 \cdot \frac{b}{b-1} = 1 \]

\[ a_1 = \frac{b-1}{b} \]

So

\[ a_i = (b-1) \left(\frac{1}{b}\right)^i, 1 \leq i < n. \]
For finite depth stacks, we choose $a_n$ to normalize the stack distance probability distribution.

$$a_n = 1 - \sum_{i=1}^{n-1} a_i = 1 - (b-1) \cdot \sum_{i=1}^{n-1} \left( \frac{1}{b} \right)^i$$

$$= \left( \frac{1}{b} \right)^{n-1}$$

Consequently,

$$\frac{a_{n-1}}{a_n} = \frac{(b-1) \left( \frac{1}{b} \right)^{n-1}}{\left( \frac{1}{b} \right)^{n-1}} = b - 1$$

Hence, $a_{n-1} \leq a_n$ if $b \leq 2$, violating the strong locality condition. However, $a_n$ is certainly less than $a_{n-2}$ for all values of $b > 1$, and is never of any concern in the calculations which follow, so we may ignore this minor imperfection in the model. Then for $1 \leq j < n$,

$$h(i) = \sum_{k=1}^{i} a_k = \sum_{k=1}^{i} (b-1) \left( \frac{1}{b} \right)^i$$

$$= (b-1) \left[ \frac{1 - \left( \frac{1}{b} \right)^{i+1}}{1 - \frac{1}{b}} - 1 \right]$$

$$= 1 - \left( \frac{1}{b} \right)^i$$

$$m(i) = 1 - h(i) = \left( \frac{1}{b} \right)^i$$
\[
\sigma_j(n) = \sum_{i=1}^{j} \frac{1}{m(i)} = \sum_{i=1}^{j} b^i
\]

\[
= \frac{1-b^{j+1}}{1-b} - 1
\]

\[
= \frac{b}{b-1} (b^j - 1)
\]

5.4.4. Summary of LRUdMH

The LRUdMH attempts to model "predictability" or clustering of module executions. It does this by insuring that there is a higher probability of executing a module which has been executed in the near past than a module whose most recent execution was earlier. We believe this to be characteristic of program behavior. Using the techniques of Markov chains, we have derived expressions for the expected number of module instance executions before executing the jth distinct module while introducing the concept of stack bias. We will make use of these and ancillary results in the next section to optimize the modified dynamic programming algorithm, given the parameters p, n, m, and the stack bias b.

5.5. Optimizing the modified dynamic programming algorithm

5.5.1. Optimal module cluster sizes

In attempting to solve the optimal dynamic assignment using the modified dynamic programming algorithm described in this chapter, there are five parameters which will affect the expected time required. These are: p, the number of processors; n, the number of modules; m, the number of module instances;
b, the stack bias for the LRUSM; and j, the module cluster set size. The first four are fixed for each problem. However, the module cluster size j is arbitrary and may be set to any value between 1 and n-1, inclusive.

Recall that the expression for algorithm complexity is

\[ E[T|j] = \frac{m}{\sigma_j(n)} \left[ E[T_s|j] + p^{n+j} \right] . \]

We see that increasing j will have two opposing effects. As j increases, \( \sigma_j(n) \) will increase, tending to decrease the expected complexity. At the same time, however, \( E[T_s|j] \) and \( p^{n+j} \) will both increase. The net effect of increasing j will depend on the other parameters of the problem, and may be either a decrease or an increase in overall algorithm complexity.

In the remaining parts of section 5.5., we will determine the optimum values of j, given the other parameters of the problem. We will also present methods of applying the algorithm to a given module instance execution string based on these results.

5.5.2. Uniform stack distance model revisited

This section deals with the model of section 5.4.3.1., in which the stack distance probabilities are all equal.

\[ \mu_{i,i+1} = \frac{1}{m(i)} = \frac{n}{n-i} , \quad 0 \leq i < n . \]
Then
\[
E[T_S|j] = \sum_{i=1}^{j} \mu_{i,i+1} \cdot p^{j+i+1}
\]
\[
= p^{j+1} \sum_{i=1}^{j} \frac{n_{-i}}{n_{i-1}} \cdot p^i
\]
\[
= np^{j+1} \left[ \sum_{i=1}^{n-1} \frac{1}{ip^i} - \frac{n-j-1}{n-1} \right]
\]

We can easily determine a lower bound for the expected time complexity of the modified algorithm under the assumption of uniform random stack distance probability distribution. Let
\[
R(n,p,j) = \frac{E[T|j]}{mp_{n+j}^j}
\]
\[
R(n,p,j) = \frac{1}{p^{n-j}} \cdot \sum_{i=1}^{j} \frac{p^i}{(n-1)\sum_{i=1}^{j} \frac{1}{ip^i}} + \frac{p^{j-1}}{j \sum_{i=1}^{j-1} \frac{1}{ip^i}}
\]
\[
\geq \frac{1}{p^{n-j}} \cdot \frac{p^{j-1}}{n-1} \left( \frac{p^j-1}{p-1} \right) + \frac{p^{j-1}}{n-1}
\]
\[
> \frac{n-j}{np_j} \cdot \left[ \frac{p}{p^{n-j}} (p^j-1) + p^j \right]
\]
\[
= \frac{(n-j)p^j}{np_j} \left[ \frac{p^j-1}{p^{n-1}} + 1 \right]
\]
\[
> \frac{(n-j)p^j}{np_j}
\]
This expression has a minimum for $1 \leq j \leq n-1$ at

$$n-j = \frac{n}{\ln(p)}$$

Substituting for $n-j$ in the lower bound gives us

$$R(n,p,j) > \frac{p^{j-1}}{j^2 \ln(p)}$$

Hence, $R(n,p,j) < 1$ for only a very limited set of values if $p > 2$, $R(n,p,j)$ is less than 1 only if $j = 1$, or $p = 3$ and $j = 3$. Furthermore, $R(n,p,1)$ converges to $\frac{n-1}{n}$ from above as $n$ increases, and $R(n,3,3)$ converges to 1, and is always greater than $\frac{7}{9}$.

Even these modest gains are misleading because for $j$ small, we are very likely to violate the assumption that a clustered sequence chosen by the algorithm must contain an instance of the module which immediately precedes the clustered sequence. The additional overhead incurred when this condition does not exist will reduce any net savings in expected time complexity to almost zero. This agrees well with our intuition. If the program has no predictability in its behavior, the modified algorithm offers no advantage at all over the original algorithm.

5.5.3. Constant bias LRUUSM revisited

The CLRUSM is defined by $n$, the stack depth, and $b$, the stack bias. Recall from section 5.4.3.2. the expressions for $m(i)$, and $\sigma_j(n)$:
\[ m(i) = \left\lfloor \frac{i}{b} \right\rfloor \]

\[ \sigma_j(n) = \frac{b}{b-1} (b^j - 1) \]

Using these equations, we can derive the expected time complexity of the modified dynamic programming algorithm for a program whose module execution sequence can be modeled by a CLRUSM.

\[
E[T_s | j] = \sum_{i=1}^{j} \frac{1}{m(i)} \cdot p^{j+i+1}
\]

\[
= p^{j+1} \cdot \sum_{i=1}^{j} \frac{1}{m(i)} \cdot p^i
\]

\[
= p^{j+1} \cdot \sum_{i=1}^{j} (bp)^i
\]

\[
= p^{j+1} \cdot \frac{bp}{bp-1} [(bp)^j - 1]
\]

\[
E[T | j] = \frac{m}{\sigma_j(n)} \left[ E[T_s | j] + p^{n+j} \right]
\]

\[
= \frac{m}{b \frac{bp}{bp-1} (b^j - 1)} \left[ p^{j+1} \frac{bp}{bp-1} [(bp)^j - 1] + p^{n+j} \right]
\]

\[
= \frac{mp^{j+1}}{b \frac{bp}{bp-1} (b^j - 1)} \left[ \frac{bp}{bp-1} [(bp)^j - 1] + p^{n-1} \right]
\]

Suppose that \( j = 1 \). Then
\[ E[T|1] = \frac{mp^2}{b} \left( \frac{bp}{bp-1} (bp-1) + p^{n-1} \right) \]

\[ = \frac{mp^2}{b} \left( bp + p^{n-1} \right) \]

\[ = mp^2 \left[ p + \frac{1}{b} \cdot p^{n-1} \right] \]

\[ \frac{E[T|1]}{mp^{n+1}} = \frac{1}{p^{n-2}} + \frac{1}{b} \]

Hence, if \( b \geq p^{n-2} \), the average complexity of the modified algorithm is \( O(mp^3) \). If \( b \) is very large, then successive trials in the CLRUSM will mostly have outcomes of choosing the top position in the stack. This means that the same module will be repeated many times consecutively, and only rarely does the module at the top of the stack change. Since the cost of an assignment depends on the processor to which the module is assigned, the processor to which the immediately preceding module instance is assigned, and the processor to which the previous instance of the same module is assigned, there is a minimum complexity for each stage of \( p^3 \), just as the above result suggests. (Of course, if we simply repeat a module, the complexity of each stage is only \( p^2 \), but the algorithm as stated does not take advantage of this fact.)

For \( j = 2 \), we can similarly find \( E[T|2] \).
\[
E[T|2] = \frac{m^3 p}{b \cdot (b^2 - 1) \cdot (bp - 1) \cdot (bp - 1)} \left\{ (bp)^2 \cdot 1 + p^{n-1} \right\}
\]

\[
= \frac{m^3 p}{b \cdot (b+1)} \left[ bp(bp + 1) + p^{n-1} \right]
\]

\[
\frac{E[T|2]}{mp^{n+1}} = \frac{1}{p^{n-3}} \cdot \frac{bp + 1}{b + 1} + \frac{1}{b} \cdot \frac{p}{b+1}
\]

\[
\approx \frac{1}{p^{n-4}} + \frac{1}{b} \cdot \frac{p}{b}
\]

If \( b \) is not large enough (i.e., \( b < p^{n-2} \)) to allow us to choose \( j = 1 \), but \( b \geq p^{(n-3)/2} \), we may optimize the modified algorithm by choosing \( j = 2 \), resulting in an overall complexity of \( O(mp^5) \).

In general,

\[
E[T|j] = \frac{m^j p^{j+1}}{b \cdot \sum_{i=1}^{j-1} b^i} \cdot \left[ \frac{bp}{bp-1} \cdot (bp-1) \cdot \sum_{i=1}^{j-1} (bp)^i + p^{n-1} \right]
\]

\[
= \frac{m^j p^{j+1}}{\sum_{i=1}^{j-1} b^i} \left[ \frac{bp}{b} \cdot \sum_{i=1}^{j-1} (bp)^i + p^{n-1} \right]
\]

and

\[
\frac{E[T|j]}{mp^{n+1}} = \frac{1}{p^{n-j-1}} \cdot \frac{\sum_{i=1}^{j-1} (bp)^i}{b^i} + \frac{p^{j-1}}{\sum_{i=1}^{j-1} b^i}
\]
\[ \approx \frac{1}{p^{n-2j}} + \frac{1}{b} \cdot \left( \frac{p}{b} \right)^{j-1} \]

As \( j \) increases, \( \frac{1}{p^{n-2j}} \) also increases, but \( \left( \frac{p}{b} \right)^{j-1} \) is decreasing if we assume that \( b > p \). The ratio of complexities will be minimized when the two terms are approximately equal

\[ \frac{1}{p^{n-2j}} = \frac{1}{b} \left( \frac{p}{b} \right)^{j-1} \]

\[ p^n = p \cdot (bp)^j \]

\[ n \log p = \log p + j \cdot \log bp \]

\[ j_{\text{opt}} \approx (n-1) \frac{\log p}{\log b + \log p} \]

At \( b \approx p \), \( j = \frac{n-1}{2} \). For \( b \gg p \), \( j \rightarrow 1 \), as we saw in the earlier discussion of \( E[T|1] \).

The expected complexity of the modified dynamic programming algorithm for \( b > p \) is approximately \( O(mp^{2j_{\text{opt}}+1}) \), where \( j_{\text{opt}} \) is a function of both the number of processors and the bias in program behavior towards module clustering. For a fixed \( p \), the algorithm complexity decreases from \( O(mp^n) \) to \( O(mp^3) \) as \( b \) increases from \( p \) to infinity. Fig. 5.6 shows the variation in \( j_{\text{opt}}/(n-1) \) with \( b \) for several different values of \( p \).

What if \( b \) is less than \( p \)? Then the complexity of the modified algorithm varies from \( O(mp^{n+1}) \) at \( b = 1 \) to \( O(mp^n) \) at \( b = p \).
Figure 5.6.
Since there is at most a factor of $p$ improvement in the algorithm, we will not concern ourselves with finding a $j^{\text{opt}}$ for this range of values of $b$, especially since the increased overhead of the modified algorithm will partially offset any improvement over the original algorithm.

The results of this section can be summarized as follows. If the program whose execution is to be optimally distributed in the DPS exhibits clustering of module executions, the modified dynamic programming algorithm offers some improvement. The amount of improvement can be related to the stack bias $b$ of the CLRUSM which models the sequence of module executions for the program. For $b > p$, the savings in expected computation time may be very large, and in the limit as $b$ increases, the expected time complexity of the modified algorithm is independent of $n$ and only cubic in $p$. The following section will describe how we may use these results to optimize the algorithm for a given instance of the optimal dynamic assignment problem.

5.4.4. Application of the modified algorithm

Let us suppose that we are given an optimal dynamic assignment problem, and we wish to determine the optimum value of $j$, the module cluster set size for a representative subgraph. We will initially examine a procedure in which we find a value for $j$ which minimizes the upper bound for the expected time complexity of the algorithm.
We begin by initializing an empty LRU stack of depth \( n \), and a list of \( n \) numbers \( \alpha(1) \) to \( \alpha(n) \), all zero. We then scan the program's module instance string. At each module instance, we examine the stack to see if the module is already in the stack. If it is not, we simply push it onto the top of the stack. If it is, we determine its depth \( i \) in the stack and increment \( \alpha(i) \), and then update the stack by moving the module to the top of the stack. After scanning all \( m \) module instances, \( \alpha(i) \) contains the number of times that exactly \( i-1 \) distinct modules are executed between consecutive instances of a module. The \( \alpha(i) \)'s will sum to \( m-n \), since we will increment some \( \alpha(i) \) for every module instance except the first instance of each of the \( n \) modules. We normalize each \( \alpha(i) \) by dividing by \( m-n \).

\( \alpha(i) \) is now the fraction of module instances in which the module executed at that instance was at depth \( i \) in the stack prior to execution.

We will use \( \alpha(i) \) as an estimator for \( a_i \), the stack distance probability for distance \( i \). For each \( \alpha(i) \), calculate \( x(i) \) such that

\[
\frac{x(i)-1}{[x(i)]^i} = \alpha(i), \quad 1 \leq i \leq \left\lfloor \frac{n}{2} \right\rfloor
\]

Using these values of \( x(i) \), find \( b(i) \), \( 1 \leq i < \left\lfloor \frac{n}{2} \right\rfloor \) such that

\[
b(i) = \min \{ [x(j)]_{1 \leq j \leq i} \}
\]

\[
= \min \{ x(i), b(i-1) \} \quad \text{for} \quad 1 < i < \left\lfloor \frac{n}{2} \right\rfloor
\]
b(i) represents a lower bound on the stack bias for stack positions 1 through i. If we choose a module cluster size \( j_{\text{opt}} = i \), then the modified dynamic programming algorithm will have an expected complexity at least as small as it would if the entire stack had a uniform bias \( b = b(i) \), since the values \( b(i+1), b(i+2), \ldots, b(n-1) \) are irrelevant in determining \( E[T_s | j_{\text{opt}} = i] \).

Obviously \( b(1) \geq b(2) \geq \ldots \geq b(n-1) \). We only want to consider those values of \( b(i) > p \). Suppose \( b(k) \geq p \) and \( b(k+1) < p \), for some \( k \). Then the final step in determining an optimum value for \( j \) is to start with \( b(1) \) and find the smallest value of \( i \) between 1 and \( k \) for which a CLRUSM with a bias of \( b(i) \) has an optimum value of \( j \leq i \). We set \( j_{\text{opt}} \) equal to this value of \( j \).

The modified dynamic programming algorithm with \( j_{\text{opt}} \) determined from the program's module instance string will have an expected time complexity bounded by \( O\left(\frac{2^i}{mp^{n+1}}\right) \). There is no other value of \( j \) which will allow us to establish a smaller upper bound, using this procedure. If there does not exist an \( i \) between 1 and \( k \) for which the optimum value of \( j \) for a CLRUSM with bias \( b(i) \) is less than or equal to \( i \), the modified algorithm might, for some value of \( i \), lead to some improvement over the original algorithm, but we cannot guarantee it. Since in this case any value of \( j_{\text{opt}} \) that we choose might lead to an expected time complexity that is worse than \( O\left(\frac{mp^{n+1}}{mp^{n+1}}\right) \), we will be better off using the original algorithm. Fig. 5.7 represents an example of this procedure. At \( j_{\text{opt}} = 4 \), the modified algorithm
<table>
<thead>
<tr>
<th>i</th>
<th>a(i)</th>
<th>a_i</th>
<th>x(i)</th>
<th>b(i)</th>
<th>j_{opt}</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>212</td>
<td>0.579235</td>
<td>2.376623</td>
<td>2.376623</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>87</td>
<td>0.237705</td>
<td>2.569922</td>
<td>2.376623</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>32</td>
<td>0.087432</td>
<td>2.676645</td>
<td>2.376623</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>14</td>
<td>0.038251</td>
<td>2.504156</td>
<td>2.376623</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>11</td>
<td>0.030055</td>
<td>2.026060</td>
<td>2.026060</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>0.027322</td>
<td></td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>0.016393</td>
<td></td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>0.005464</td>
<td></td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>0.002732</td>
<td></td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>-</td>
</tr>
</tbody>
</table>

Figure 5.7.

Application of the modified dynamic programming to a hypothetical program with 10 modules and 376 module instances during its execution on a two-processor distributed processing system. Choosing a module cluster size of 4 minimizes an upper bound on time complexity (in expected value).
will have an expected complexity of $O\left(mp^9\right)$ rather than $O\left(mp^{11}\right)$.

Although this is perhaps the simplest algorithm which
makes use of the model presented in this section, it is certainly
not the only one possible. Others which may give smaller upper
bounds on algorithm complexity are possible. For instance,
using the normalized values \{\(\alpha(i)\)\} as estimators for the stack
distance probabilities \{\(a_i\)\}, we can compute \(E[T_s | j]\) and \(\sigma_j(n)\)
for each value of \(j\) from 1 to \(n-1\). (This could be done reasonably
efficiently since

\[
E[T_s | j] = E[T_s | j-1] + \frac{1}{m(j)} \cdot p^{j+i+1}
\]

\[
\sigma_j(n) = \sigma_{j-1}(n) + \frac{1}{m(j)}, \quad 1 < j < n
\]

where \(m(j) = 1-h(j)\)

\[
= 1 - \sum_{i=1}^{j} a_i
\]

This would enable us to calculate \(E[T | j]\), \(1 \leq j \leq n\), and choose
the optimal value of \(j\). The expected complexity of the modified
algorithm with \(j_{\text{opt}}\) determined in this fashion will certainly be
no greater than if the value of \(j_{\text{opt}}\) found by the preceding method
were used, and conceivably could be much better, perhaps sufficiently
better to justify the increased overhead. Perhaps still other
procedures for determining \(j_{\text{opt}}\) could be devised to further lower
the expected complexity. One potentially fruitful area of
investigation is in considering $j_{opt}$ to be variable rather than a constant over the life of the program. This might be especially valuable for those programs whose behavior goes through several "phases," somewhat analogous to the concept of activity sets due to Madison and Batson in describing program page localities (MAD76).

Finally, we must justify an assumption made earlier. In describing the extension of the assignment tree by means of representative subgraphs, we stated that it is necessary that the module which immediately precedes the subgraph must also occur in the $j$-module cluster to which the subgraph's clustered sequence corresponds. We feel that, in the light of the discussion of $j_{opt}$, the assumption is in fact justified. Suppose $j_{opt} = 1$. Then obviously since we are beginning a new cluster sequence for a $1$-module cluster set, the module immediately preceding the cluster sequence will not be in the set, and we will have to use the techniques of the original algorithm to extend the assignment tree one stage. The next module instance, however, will almost certainly be the same one, since $a_1 = (b-1) \cdot \frac{1}{b}$, and $b \geq p^{n-2}$ for $j_{opt} = 1$.

Suppose that $j_{opt} \geq 2$. Let $M_s(i)$ be the probability that the $i$th distinct module in the clustered sequence (i.e., $j$) is not the module immediately preceding the clustered sequence.
Then
\[ M_j(i) = \frac{a_{i+1} + a_{i+2} + \ldots + a_n}{a_i + a_{i+1} + a_{i+2} + \ldots + a_n} , \quad 1 < i < j \leq \frac{n}{2} \]

and \( M_j(1) = 1 \)

Since
\[ \sum_{k=1}^{n} a_k = \left( \frac{1}{b} \right)^{i-1} \]

\[ M_j(i) = \frac{1}{b} , \quad 1 < i < j \]

The probability that none of the \( j \) modules of the clustered sequence is the module immediately preceding the clustered sequence is
\[ \prod_{i=1}^{j} M_j(i) = \left( \frac{1}{b} \right)^{j-1} \]

Consequently, as \( b \) decreases, the expected time complexity of the modified algorithm will increase, but as \( b \) gets smaller \( j_{opt} \) grows, so that for sufficiently large \( n \), the probability that the clustered sequence includes an instance of the preceding module actually increases. This probability is in fact maximum for \( j = \frac{n}{2} \) (\( b=p \)) justifying the statement that the additional overhead required by the algorithm when the assumption is not satisfied will be negligible.
5.5.5. Chapter summary

The exponential growth in complexity of the dynamic programming algorithm prompted us to consider a modification of the algorithm which relies on the assumptions that at any point in the program execution there is a set of favored modules most likely to be executed in the near future and this set changes slowly with time. We used an LRU stack model with a constant bias to enable us to model the sequence of module executions of a program. The amount of improvement of the modified algorithm over the original algorithm is a function of the relative sizes of the stack bias and the number of processors. In the limit as the bias becomes very large, the expected complexity approaches $O(mp^3)$.

We then illustrated how one may make use of the constant bias LRUUSM to develop procedures for choosing the module cluster set size which optimizes the algorithm by minimizing the upper bound on expected time complexity. We also showed in an informal way that an assumption made earlier concerning the presence of one particular module in a module cluster set seems to be justified.

The modified dynamic programming algorithm may offer significant reductions in the amount of computations necessary to determine an optimal dynamic assignment for the execution of a program. Since we are able to clearly identify those problems in which it may actually prove worse in terms of time complexity
then the original algorithm, the modified algorithm has the potential to be a valuable tool in investigating real time dynamic program execution distribution strategies.
CHAPTER 6

CONCLUSION

6.1. Summary of results

Distributed processing systems have been purported to offer many desirable features: reliability, graceful degradation of performance with component failure, resource sharing and load balancing, ease of system growth or modification, increased functional capabilities, and more. This thesis addresses a problem which impacts many of these features, but which has only recently begun to receive attention.

The problem with which we are concerned is the determination of an optimal dynamic assignment for a program. Chapter 2 describes the structure of a program which is to be executed in a distributed environment, and identifies the nature of the costs which a program's execution may incur. Two cost categories, those of module execution costs and intermodule communications costs, have been considered previously in finding an optimal static assignment. The third cost category which is specific to the dynamic assignment problem is that of module reassignment costs. We also have expanded the definition of communication and reassignment costs to allow asymmetric and/or nonuniform costs.

The static assignment problem for the two processor case has been shown to be efficiently solvable using commodity flow techniques. In Chapter 3, we have extended these techniques to the two processor dynamic assignment problem. First we construct
a dynamic processor flow graph which incorporates all execution, reassignment, and communication costs. The reassignment (and communication) costs may be asymmetric. Then using a max-flow min-cut algorithm, we can determine an optimal dynamic assignment for the program execution by finding a minimum capacity cutset of the processor flow graph. The time complexity of the dynamic assignment algorithm, using the most efficient max-flow min-cut algorithm known, is $O\left(\frac{m^2}{\sqrt[3]{m}}\right)$, where $m$ is the number of module executions during the program lifetime. Finally, the limitations of the commodity flow approach were examined, indicating its lack of suitability to problems with more than two processors.

Chapter 4 presents an entirely different algorithm for finding an optimal dynamic assignment. The new algorithm is a dynamic programming approach which is equivalent to a shortest path algorithm. The major advantage of this is that it may be used for problems with arbitrary numbers of processors. The algorithm makes use of the restriction that the cost of assigning an execution or instance of a module is a function only of the processor to which it is assigned, the assignment of the module instance which immediately precedes it, and the assignment of the most recent instance of the same module. The price that the dynamic programming algorithm must pay for its wider applicability is increased time and space complexity. The worst case (which is also the average case) time complexity is $O\left(mp^{n+1}\right)$ where $p$ is the number of processors and $n$ is the number of program modules.
The worse case space complexity is $O(mp^n)$. However, in section 3 of Chapter 4 we indicated that substantial reductions in space complexity may be expected by dynamically pruning the dynamic assignment tree which consists of all of the shortest paths constructed by the algorithms. For fixed $p$, the expected space complexity with pruning appears to be $O(m+np^n)$ for very large $m$, and is much less than worst case even for small $m$.

The correctness of the dynamic programming approach is assured by the fact that, for each module instance, it considers every possible current assignment of modules. This is also the cause of the exponential growth in complexity with $n$. In Chapter 5, we have described a modification of the dynamic programming algorithm which may offer significant reductions in expected time complexity for those problems in which the program exhibits clustering in its sequence of module executions. The improvements are due to the fact that it is sometimes possible to consider only a small fraction of all possible current module assignments at each stage or module instance.

First we derived a general expression for the expected complexity of the modified algorithm. The derivation in section 5.3.3. results in the following equation:

$$E[T|j] = \frac{m}{c_j(n)} \left[ E[T_{S}|j] + p^{n+j} \right]$$

$j$ is the size of a subset of program modules whose composition is arbitrary. $c_j(n)$ is the expected length of a maximal length
module instance sequence which contain exactly \( j \) distinct modules. \( E[T_S|j] \) is the expected time complexity for finding all shortest paths through a structure called a representative subgraph of the dynamic assignment graph, given a module cluster set size \( j \). \( E[T|j] \) is the overall expected complexity of the modified algorithm given \( j \).

Both \( E[T_S|j] \) and \( \sigma_j(n) \) depend on the nature of the sequence of module executions for the program. We have modeled this part of program behavior with an LRU stack model. Under the assumption of uniform stack distance probabilities we showed that the modified algorithm is no better than the original algorithm. We then introduced the concept of a constant bias LRU stack model, or CLRUSM, which is characterized by a constant \( b > 1 \), the ratio of successive stack distance probabilities. Under this model, we were able to show that for \( b > p \), the algorithm may show significant improvements in expected time complexity. Furthermore, we may optimize the algorithm by choosing that value of \( j \) which minimizes the expected complexity for the problem. The optimum value of \( j \) depends on both \( b \) and \( p \), and is given by

\[
j_{\text{opt}} = (n-1) \cdot \frac{\ln p}{\ln b + \ln p}
\]

with the minimum expected complexity being

\[
E[T|j_{\text{opt}}] \approx O\left(m p_{\text{opt}}^{2j_{\text{opt}} + 1}\right)
\]
We concluded Chapter 5 by describing a procedure which examines the program's module execution sequence to determine a value for \( j \) which is optimal for the program.

These algorithms are unrealizable in the sense that each requires complete knowledge of the program's execution beforehand. However, they should be valuable in the comparison of various real time heuristic assignment strategies by allowing us to compare the performance of a given heuristic to the theoretical optimum. In this sense they are analogous to the OPT page replacement strategy for fixed memory partitions, which is also optimal but unrealizable (SPI77). The concepts developed in dealing with the modified dynamic programming algorithm may be of interest in themselves, particularly as we attempt to construct heuristic assignment strategies.

6.2. Future research

A number of problems related to finding an optimal dynamic assignment of a program execution should be considered. Principal among these is the need for development of real time heuristic assignment strategies which in some manner use the past history of the program's execution to predict "good" assignments in the future. Among the points which need to be studied in this area are the questions of how we obtain information of past program behavior and who makes the assignment decisions in a distributed system. Ideally, the execution of an assignment algorithm should
itself be distributed and therefore subject to the same assignment strategy which it implements.

A second major question concerns program partitioning. We have assumed that partitioning is done by the programmer on a logical (to him) basis. Is it possible to use automatic techniques of program partitioning to produce modules which are more suitable for distribution?

Program parallelism introduces another dimension in the dynamic assignment problem, particularly in a system in which processor multiprogramming, interprocessor latency, and communications subnetwork congestion may substantially influence program costs. Techniques used for optimizing assignments for concurrent executions of program modules are likely to be stochastic in nature, and therefore we shall need better models of the total activity in a distributed processing system.

These and other open questions deserve attention in the future. Only when we begin to find some answers will we be able to achieve some of the full potential of distributed processing systems. It is hoped that the ideas presented in this thesis will be of use in achieving this goal.
APPENDIX A

The edge capacities of the dynamic processor flow graph as given by equations (1) through (5) of sections 3.4.1. and 3.4.2. are not the only possible valid capacity assignments. Since the dynamic processor flow graph is, as its name suggests, interpreted as a flow network, we may use superposition of flow to allow us to consider each cost category separately.

Execution costs and (symmetric) communication costs are incorporated very simply. The capacity of an edge \((A_i, P_1)\) is just the cost of executing \(A_i\) on \(P_2\). Similarly, \(c(A_i, P_2)\) is the cost of executing \(A_i\) on \(P_1\). The capacity of an edge \((A_i, B_j)\) for \(A \neq B\) is the cost of any communication between \(A_i\) and \(B_j\) if they are assigned to different processors.

Reassignment costs, since they are assumed to be asymmetric, are not as simple to deal with. Consider the portion of a dynamic processor flow graph shown in Fig. A.1. There are four possible distinct assignments of the node pair \(A_i, A_{i+1}\), as illustrated by the four cutsets labeled I to IV. The capacities of \((A_i, P_1), (A_{i+1}, P_1), (A_i, P_2)\), and \((A_{i+1}, P_2)\) are \(x_1, x_2, x_3,\) and \(x_4\), respectively, while \(c(A_i, A_{i+1})\) is given as \(y\). (Remember that these capacities are only for the component of flow due to reassignment costs.)

Cut I assigns both \(A_i\) and \(A_{i+1}\) to \(P_2\), and consequently there should be no contribution to the capacity of cut I due to reassigning A between \(A_i\) and \(A_{i+1}\). Let \(c\) be the contribution
Figure A.1.

A portion of a two-processor dynamic processor flow graph showing the processor nodes and the nodes representing the $i$th and $(i+1)$st instances of module $A$. The broken lines show the four possible assignment combinations for $A_i$ and $A_{i+1}$. 
to I's capacity related to reassignment of A between these two instances. Then

\[ A.1. \quad c^I = x_1 + x_2 = 0 \]

Similarly,

\[ A.2. \quad c^{II} = x_2 + x_3 + y = r_{1,2}(A) \]

\[ A.3. \quad c^{III} = x_1 + x_4 + y = r_{2,1}(A) \]

\[ A.4. \quad c^{IV} = x_3 + x_4 = 0 \]

where \( r_{i,j}(A) \) = cost of reassigning module A from processor \( i \) to processor \( j \).

Adding A.3. to A.2. gives us

\[ x_1 + x_2 + x_3 + x_4 + 2y = r_{1,2}(A) + r_{2,1}(A) \]

Substituting for \( x_1 + x_2 \) and \( x_3 + x_4 \) using A.1. and A.4. results in

\[ A.5. \quad y = \frac{1}{2} [r_{1,2}(A) + r_{2,1}(A)] \]

A.5. is stated as eq. (1) in section 3.4.1.

Using this expression for \( y \), equations A.2. and A.3. become

\[ x_2 + x_3 + \frac{1}{2} [r_{1,2}(A) + r_{2,1}(A)] = r_{1,2}(A) \]

\[ x_1 + x_4 + \frac{1}{2} [r_{1,2}(A) + r_{2,1}(A)] = r_{2,1}(A) \]
or

\[ A.6. \quad x_2 + x_3 = \frac{1}{2} [r_{1,2}(A) - r_{2,1}(A)] \]

\[ A.7. \quad x_1 + x_4 = \frac{1}{2} [r_{2,1}(A) - r_{1,2}(A)] \]

We may give either \( x_2 \) or \( x_3 \) an arbitrary value, and this will completely determine the value of the other. The same holds true for \( x_1 \) and \( x_4 \). An obvious choice is to set one of each pair to zero. The inclusion of reassignment costs in the complete processor flow graph is made particularly simple by choosing either \( x_1 \) and \( x_2 \) or \( x_3 \) and \( x_4 \) to both be zero. We will arbitrarily choose \( x_3 \) and \( x_4 \) to be zero. Then

\[ x_2 = \frac{1}{2} [r_{1,2}(A) - r_{2,1}(A)] \]

\[ x_1 = \frac{1}{2} [r_{2,1}(A) - r_{1,2}(A)] \]

Assuming we make this choice, we now consider the partial dynamic processor flow graph of Fig. A.2. The edge between \( A_{i+1} \) and \( P_1 \) will also be involved in determining the cost of possible reassignment of \( A \) between \( A_{i+1} \) and \( A_{i+2} \). To simplify matters, we separate the edge \((A_{i+1},P_1)\) into two parallel edges, one with capacity \( x_2 \) as derived previously and the other with capacity \( x_1 \). The capacity of a single edge \((A_{i+1},P_1)\) will be \( x_2 + x_1 \). \((A_{i+1},P_2)\) is treated in a like manner.
Figure A.2.

Similar to Fig. A.1, but with the (i+1)st and (i+2)nd instances of A. \((A_{i+1}, P_1)\) and \((A_{i+1}, P_2)\) are separated into two parallel edges.
By requiring that the reassignment of \( A \) between \( A_i \) and \( A_{i+1} \) impact only \( x_2 \) and not \( x_1 \) (also, \( x_4 \) and not \( x_3 \)), we can carry out the analysis of Fig. A.2 independently of but in precisely the same manner as the analysis of Fig. A.1.

Thus, by setting \( x_3^1 \) and \( x_4^1 \) to zero, we can show that

\[
\begin{align*}
x_2^i &= \frac{1}{2} \left[ r_{1,2}(A) - r_{2,1}(A) \right] = x_2 \\
x_1^i &= \frac{1}{2} \left[ r_{2,1}(A) - r_{1,2}(A) \right] = x_1 \\
y^i &= \frac{1}{2} \left[ r_{1,2}(A) + r_{2,1}(A) \right] = y
\end{align*}
\]

Consequently,

\[
\begin{align*}
c(A_{i+1}; P_1) &= x_1^i + x_2 = 0 \\
c(A_{i+1}; P_2) &= x_3^i + x_4 = 0
\end{align*}
\]

We can extend this analysis to \( A_{i+3}, A_{i+4}, \text{ etc.} \). Doing so results in \( c(A_i; P_2) = 0 \) for all edges \( (A_i, P_2) \) and \( c(A_i; P_1) = 0 \) for all \( A_i \) except for \( A_1 \) and \( A_{\text{max}} \). Equations (2) to (5) of section 3.4.2 reflect this analysis.

There is obviously a similar, equally simple valid assignment of flow capacity if we choose \( x_1 = x_2 = 0 \) in Fig. 4.1 rather than \( x_3 = x_4 = 0 \). Equation (1) remains unchanged; equations (2) to (5) become

\[
\begin{align*}
(2) \quad & c(A_i; P_1) = e_2(A_i) \quad , \quad 1 \leq i \leq \text{max}(A) \\
(3) \quad & c(A_1, P_2) = e_1(A_1) + \frac{1}{2} \left[ r_{1,2}(A) - r_{2,1}(A) \right] \quad , \quad \text{max}(A) \neq 1
\end{align*}
\]
(4) \( c(A_{\text{max}}, P_2) = e_1(A_{\text{max}}) + \frac{1}{2} [r_{2,1}(A) - r_{1,2}(A)] \), \( \max(A) \neq 1 \)

(5) \( c(A_i, P_2) = e_1(A_i) \), \( 1 < i < \max(A) \) or \( i = \max(A) \)

Other valid flow capacities (an infinite number, in fact) are possible, but these lead to more complicated constructions of the dynamic processor flow graph.
APPENDIX B

The derivation of the recurrence relations for the probabilities $q_i$ by the method of section 4.3.2.3. is impractical for $p$ much larger than three, since the number of distinct ways of connecting the $p$ nodes of a $p$-set of $s_{i-1}$ to the $p$ nodes in a $p$-set of $s_i$ such that every node in $s_i$ has exactly one edge from $s_{i-1}$ grows as $p^p$. Here we describe a relatively simple algorithm for mechanically generating the recurrence relations.

For arbitrary $p$ we begin by considering every possible partition class of $p$ unlike objects (the $p$ nodes of the $p$-set in $s_i$). A partition class is a set of partitions in which the parts of the partitions have the same sizes and multiplicities as every other partition in the class. For instance, one partition class of $\{A,B,C\}$ is the set of all partitions which group two elements into one part. This partition class consists of the partitions $\{\{A,B\}, \{C\}\}$, $\{\{A,C\}, \{B\}\}$, and $\{\{B,C\}, \{A\}\}$.

We count the number of partitions belonging to each partition class. Since each part of a partition class will be uniquely paired with exactly one out of $p$ distinct objects (the $p$ nodes in the corresponding $p$-set of $s_{i-1}$), we can then easily enumerate all possible ways of partitioning nodes in a $p$-set of $s_i$ into a partition belonging to a particular partition class and then connecting all nodes of each part to a distinct node of $s_{i-1}$. 
Each of the ways in which the parts of a partition in a given class may be connected to nodes in the previous stage is equally likely by assumption. All partitions in the same class have the same expectation for the number of nodes in the p-set of $s_{i-1}$ which will not have edges to the p partitioned nodes of $s_i$. By summing the expected value over all possible partition classes, we can compute the unconditional expectation that a node of $s_{i-1}$ will have no edges to $s_i$.

An example should help to clarify the above discussion. Consider the case $p = 4$. There are five partition classes of four distinct objects. These classes are represented in the standard notation by $4, 31, 2^2, 21^2$, and $1^4$ (RI058). The edge probabilities and unnormalized weights for each partition class are given in Table B.1. Summing up the weighted expectations and normalizing by dividing by $4^4 = 256$ gives us

$$q_{i-1} = \frac{1}{256} [q_1^4 + 12q_1^3 + 54q_1^2 + 108q_1 + 8]$$

A simple check is to examine the value of $q_{i-1}$ when $q_1$ is equal to zero. In this case,

$$q_{i-1} = \frac{1}{256} [8] = \left(\frac{4-1}{4}\right)^4$$

which was derived in section 4.3.2.3. Also, if $q_1 = i$, then $q_{i-1} = 1$, since the coefficients of the $q_i$ terms sum to 1.
<table>
<thead>
<tr>
<th>partition class</th>
<th>weight (unnormalized)</th>
<th>no edges probabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>((4)_{1}[4])</td>
<td>(3 + q_i^4)</td>
</tr>
<tr>
<td>31</td>
<td>((4)_{2}[3][1])</td>
<td>(2 + q_i + q_i^3)</td>
</tr>
<tr>
<td>(2^2)</td>
<td>((1/2!)(4)_{2}[2][2])</td>
<td>(2 + 2q_i^2)</td>
</tr>
<tr>
<td>(21^2)</td>
<td>((1/2!)(4)_{3}[2][1][1])</td>
<td>(1 + 2q_i + q_i^2)</td>
</tr>
<tr>
<td>(1^4)</td>
<td>((1/4!)(4)_{4}[1][1][1][1])</td>
<td>(4q_i)</td>
</tr>
</tbody>
</table>

Table B.1
REFERENCES


APE77 Apers, P.M.G. "Average case analysis of Dinic-Karzanov network flow algorithm," Tech. Rept. HP 77-7-001, University of California, Santa Cruz, June 1977.


BOK77a Bokhari, S.H. "Multiprocessor scheduling with shortest path algorithms," Tech. Rept. ECE-CS-77-11, Dept. of Electrical and Computer Engineering, University of Massachusetts, Amherst, MA., December 1977.


FAR72  Farber, D.J., and Larson, K.C. "The system architecture of
distributed computer system - the communications
system," Proc. of the Symposium on Computer-
Communications Networks and Teletraffic, Polytechnic

FOL76  Foley, J.D. "A tutorial on satellite graphics systems,"

FOR62  Ford, L.R., Jr., and Fulkerson, D.R. Flows in Networks,

FOR78  Forsman, F., and Fuller, B. An implementation of the
max-flow min-cut algorithm of Dinic and Karzanov
(private communication), 1978.

FUL78  Fuller, S.H., Ousterhout, J.K., Raskin, L., Rubinfeld,
P.I., Sindhu, P.J., and Swan, R.J. "Multi-micropro-
cessors: an overview and working example,"

GAL78  Galil, Z, "A new algorithm for the maximal flow problem,"
Dept. of Mathematical Sciences, Tel-Aviv University,
Tel-Aviv, Isreal, May 1978.

GON78  Gonsalves, T.J. "Heuristic algorithms for distributed
processor scheduling with limited memory," Master's
Thesis, Rice University, Houston, TX, June 1978.

HU69   Hu, T.C. Integer Programming and Network Flows,
Addison-Wesley, Reading, MA, 1969.

JEN78  Jensen, E.D. "The Honeywell Experimental Distributed
Processor - an overview," Computer 11:1, January 1978,
pp. 28-37.

KLE76  Kleinrock, L. Queueing Systems Vol. II: Computer

MAD76  Madison, A.W., and Batson, A.P. "Characteristics of
program localities," Comm. ACM 19:5, May 1976,

MIC77  Michel, Jr., and van Dam, A. "Evaluation of performance
improvement in distributed processing," Brown
University, Providence, RI, August 1977.

RAO77  Rao, G.S., Stone, H.S., and Hu, T.C. "Scheduling in a
distributed processor system with limited memory,"


