This is to certify that the

dissertation entitled

THE MODELING AND ANALYSIS OF

PARALLEL ALGORITHM DESIGN

presented by

Gerald William Cichanowski

has been accepted towards fulfillment
of the requirements for

PhD degree in Computer Science

Dr Robert G. Reynolds
Major professor

Date July 27, 1983

THE MODELING AND ANALYSIS OF PARALLEL ALGORITHM DESIGN

By

Gerald William Cichanowski

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Department of Computer Science

1983

ABSTRACT

THE MODELING AND ANALYSIS OF PARALLEL ALGORITHM DESIGN

By

Gerald William Cichanowski

Parallel algorithms contain information which describes process synchronization and intercommunication. This information is only trivially present in sequential algorithms. The impact of this added information on the design process for parallel algorithms for multiprocessor systems is studied.

In this research, the metrics of Halstead's software science are extended to describe the information content of parallel algorithms. The extended metrics are used to show that parallel algorithms may have higher information contents than their sequential counterparts. Part of this increase is due to the sequential component supporting the activities of the parallel component. We call this support subsidization.

Given that parallel algorithms may contain additional information over that for their sequential counterparts, this research examines the effects of this added information on the parallel algorithm design process. To do this, a simulation model of the algorithm design process was developed. The model is based on the information content of an algorithm, and the information which is used to make

decisions during the design process. Two representative algorithms were chosen for experiments based on the simulation model; a parallel root finding algorithm and a parallel bin packing algorithm. Both algorithms have a single localized parallel segment, but they have other structural differences.

In the experiments, it was seen that the type of information which was used by design decisions impacted the number of steps which were required to design the two algorithms. It also impacted the rate at which information was added to an algorithm, while it was being designed. Also, the performance of a given type of information differed when used for each of the two algorithms. These results are then used to suggest ways in which parallel algorithms can be effectively developed.

ACKNOWLEDGEMENTS

I would like to thank the members of my committee, Drs. Carl Page, Herman Hughes, Jacob Plotkin, and Robert Reynolds, for their assistance in creating this work and for directing my studies. I am specially grateful to my advisor, Dr. Reynolds, for his many hours of consultation and assistance. He had a major influence on the creation of this dissertation.

TABLE OF CONTENTS

List of Tables..... vii

List of Figures..... ix

1.0 Introduction.....1

1.1 The Algorithm Design Process.....4

1.2 Organization of the Dissertation.....8

2.0 Metrics for the Description of Algorithm Information Content.....12

2.1 Metrics for Sequential Algorithms.....13

2.1.1 An Example of Halstead's Metrics for a Sequential Algorithm.....14

2.1.2 Potential Volume.....17

2.2 Metrics for Parallel Algorithms.....19

2.2.1 Comparison of a Sequential and a Parallel Root Finding Algorithm.....23

2.2.2 Component Metrics and Potential Conceptual Volume.....29

2.3 Summary.....31

3.0 The Graph Model of Parallel Algorithm Structure.....33

3.1 The Graph Language.....34

3.2 Relation of Graph Language to the Metrics.....38

3.3 Metrics for Partial Designs.....39

3.4 Summary.....45

4.0 Algorithm Transformation and Subsidization.....46
4.1 Algorithm Transformation.....47
4.2 Subsidization.....49
4.3 Summary.....53
5.0 A Parallel Bin Packing Algorithm.....55
5.1 The Bin Packing Problem.....57
5.2 The Best-Fit With Replacement Rule.....60
5.3 Implementation of the BFR Rule as a Parallel Algorithm.....63
5.4 The MBFR Algorithm Simulation.....66
5.5 The Computational Complexity of the MBFR Algorithm..75
5.6 Summary.....78
6.0 The Algorithm Design Model.....80
6.1 A Design Step.....81
6.2 The Placement Process.....85
6.3 The Elaboration Process.....93
6.4 The Procedural Model.....99
6.5 The Simulation Model of Algorithm Design.....102
6.5.1 The Model's Environment.....103
6.5.2 Design Methodologies in the Simulation Model....106
6.5.3 The Design Process.....108
6.6 An Illustration of the Design Step Process.....112
6.7 Summary.....118
7.0 The Experiments.....120
7.1 The Sample Algorithms.....121
7.2 Design Methodologies Used in the Experiments.....126
7.3 The Experiments.....128

7.4 Structural Differences Between Algorithms ROOTF and MBFR.....134
7.5 Effects of the Algorithm's Structure on the Elaboration Process.....136
7.6 Relationship Between the Placement Process and the Rate Information is Added to an Algorithm's Design.138
7.7 Correspondence Between the Model's Results and Human Designers.....142
7.8 Summary.....149
8.0 Conclusions.....151
Appendix A The Bin Packing Algorithm.....156
Bibliography.....160

LIST OF TABLES

2.1 Operator counts for the polynomial program.....16
2.2 Operand counts for the polynomial program.....16
2.3 Operator counts for the bisection algorithm.....26
2.4 Operand counts for the bisection algorithm.....26
2.5 Operator counts for the multiprocessor algorithm.....27
2.6 Operand counts for the multiprocessor algorithm.....28
3.1 Element counts and conceptual component volumes for the algorithm in Figure 3.2.....40
3.2 Element counts for the algorithm of Figure 3.3.....44
3.3 Element counts for the algorithm of Figure 3.4.....44
4.1 Component lengths and vocabularies for the two root finding algorithms.....49
4.2 Usage indicies for the two root finding algorithms...51
5.1 Packings for each simulation list length for each of the 30 random permutations.....68
5.2 A summary of the performance of the MBFR algorithm for each list length (L).....69
6.1 Element counts and conceptual volume for step one of the abstract model illustration.....117
6.2 Element counts and conceptual volume for step two of the abstract model illustration.....117

7.1 The design methodology probability mixes used in the parallel algorithm design experiments.....127
7.2 The average number of design steps which were required for each design mix, for algorithms MBFR and ROOTF.....129
7.3 The average information (bits of conceptual volume) added to the algorithm by a design step, for each design mix.....131
7.4 Structural differences between algorithms ROOTF and MBFR.....135

LIST OF FIGURES

1.1 Kung's matching criteria for hardware and algorithms..5
3.1 The structures of the graph language.....35
3.2 A sample graph language algorithm.....37
3.3 A partial algorithm design.....42
3.4 The algorithm of Figure 3.3 after the next design step.....43
4.1 Transformations for converting a sequential algorithm into a parallel algorithm.....48
4.2 The placement of subsidization for the four algorithm transformation types.....52
5.1 A comparison of the BF, BFR, BFD, and optimal packing for the example list.....62
5.2 Plot of the average packing as a ratio of the optimal packing for the MBFR simulation.....70
5.3 Plot of the worst case packing as a ratio of the optimal packing for the MBFR simulation.....71
5.4 Plot of the maximum number of excess bins as a ratio to the optimal packing for the MBFR simulation.....72
5.4 Worst case, average, and optimal packings with the maximum and average number of processors, required by the algorithm, for the MBFR simulation.....73
6.1 Abstract model of the algorithm design process.....100

6.2 Simulation model's representation of the design environment.....104
6.3 Simulation model's representation of a design methodology.....107
6.4 An overview of the designer component of the simulation model.....113
6.5 Procedure for choosing selection criteria and next chunk.....113
6.6 An overview of the elaboration procedure for the simulation model.....114
6.7 A sample algorithm to illustrate the flow of the algorithm design model.....115
7.1 Graph language representation of the ROOTF algorithm.....122
7.2 Graph language representation of the MBFR algorithm.....123
7.3 The details of the PARFOR block for the MBFR algorithm of Figure 7.2.....124
7.4 Average design time, in minutes, for algorithms ROOTF and MBFR as a function of the design mix.....133
7.5 The producer and consumer processes in full detail..146
7.6 The producer and consumer processes with the synchronization and process intercommunication details removed.....147
7.7 The producer and consumer processes with all parallel details removed.....148

CHAPTER 1

Introduction

During the past decade, many advances have been made in the realm of computer architecture. While many of these advances have dealt with traditional Von Nuemann architectures, alternate models of computer systems have begun to appear [1-2]. These systems include pipeline processors, array processors, multiprocessors, and data flow computers [2]. Although these non-traditional models of computer systems differ from each other, the common thread which unites them is that they each potentially support some concurrency within an algorithm.

Since these non-traditional architectures support parallelism, if they are to be effectively used, algorithms must be developed to expolit this capability. This research will examine the software design process for parallel algorithms.

Two approaches have been taken for the development of software for such non-traditional architectures. The first approach is to start with a sequential algorithm and then to translate it into a form which is suitable for execution on a parallel computer. Although these transformations are made to the algorithm's code, since the code is a model of the algorithm, the transformations are, in essence, made to the algorithm. These translations may be made by hand, or by an automated means, such as a compiler. Kuck, [3], discusses this approach. He describes procedures for the automatic transformations of programs written in sequential languages to a form suitable for vector processors. These transformations include techniques such as tree height reductions, vectorizing DO LOOPs, simultaneous evaluation of alternate conditional paths, etc. This approach is promising primarily for achieving speedups to existing sequential software.

The approach of translating sequential algorithms taken above, can also take advantage of the extensive research which has been done in the area of sequential software design. Since the algorithm is first created as a sequential algorithm and then translated into parallel, standard sequential design techniques can be used to create the sequential algorithm. Then a simple transformation can be used to inject parallelism into the design. Starting in the late 60's with Dijkstra's famous letter to the editor

[4], through much of the 70's many successful sequential design methodologies have been identified and formalized. [5-10] These methodologies include several hierarchical design methods; top-down programming, bottom up programming, stepwise refinement of program abstraction, and modular programming. Many tools and strategies have been developed to take advantage of these methodologies; structured programming, HIPOs, Nassi-Sniederman charts, etc. [5].

The second approach for the design of parallel algorithms is to design the algorithms from scratch as parallel. This has been done by basing the design on languages which have been specifically designed or extended for parallel machines. Languages such as VAL [11] for data flow computers, and Concurrent Pascal [12] for multiprocessors are examples of such languages. The strength of using such languages is that a programming language is essentially a model of the computer which is being programmed. As such, any program developed in that language should be directly supported by the computer in question, and should not require extensive transformations to take advantage of the machine's capabilities. The weakness of this approach is that the strong foundation of design methodologies which have been developed for sequential algorithms, may not necessarily be applicable to software for non-Von Nuemann architectures. In light of these potential differences in design technqiues for

different architectures, we will now examine the algorithm design process.

1.1 The Algorithm Design Process

The design process for algorithms is essentially an information gathering process. To successfully match an algorithm to hardware, Kung [13] has suggested three pieces of required information or matching criteria; process granularity (size), process control, and communication between processes (Figure 1.1). These criteria correspond to information which a designer must have to successfully develope an algorithm. The ability to support these criteria differ significantly between sequential and parallel architectures. In a sequential algorithm, process synchronization and interprocess communication can be ignored, since there is only a single process. On a parallel machine, potentially several processes must be controlled. Thus, synchronization and communication between them becomes necessary. Differences also exist between algorithms for different types of parallel architectures. For example, an algorithm which is designed for a multiprocessor will differ from its counterpart for a data flow processor. On a data flow processor, granularity is small, an individual operation. On a multiprocessor, granularity is normally larger, and may differ between the

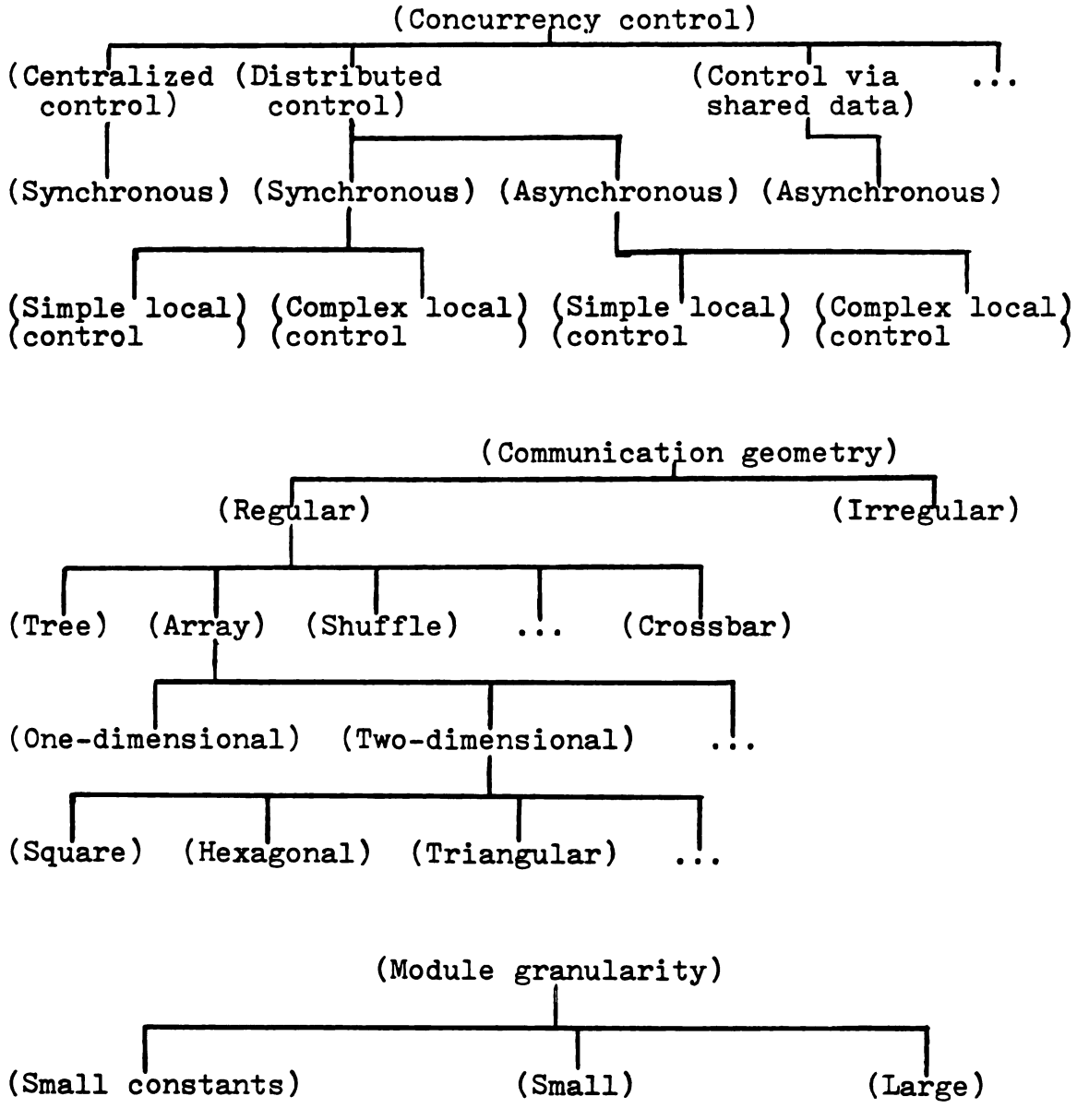


Figure 1.1 Kung's matching criteria for hardware and algorithms.

processes which make up the algorithm.

Another way of viewing the differences in information requirements between sequential and parallel algorithms is through the process of correctness proofs. Owicki and Gries [14] discuss correctness proofs for parallel algorithms. In their discussions, they show that a need exists to go beyond partial correctness and termination, the standard properties required for sequential algorithms, to demonstrate correctness. Partial correctness is shown by demonstrating that a program's output set is generated from its input set, using assertions based on the program's variables and rules describing the effects of the program's statements. Owicki and Gries show that additionally, non-interference between concurrent processes and the absence of deadlock must be proven. These additional properties correspond directly to the additional information that is required to design parallel algorithms.

Therefore, we can see that the information requirements for designing algorithms differs when the algorithm is designed for a sequential or parallel architecture. Indeed, the information requirements vary between different types of parallel architectures. Therefore, it is important to identify what types of parallel architectures will be considered. This research will restrict itself to a subclass of parallel architectures. The parameters which describe the architectures of interest are; asynchronous

process control, local process intercommunication, and varying process size. This describes multiprocessors when the granularity is large, and data flow computers when granularity is small.

The purpose of this research is to examine the design process for parallel algorithms, in light of these additional information requirements. This will be done to find out if a different design approach is required for parallel algorithms. To do this, the algorithm design process will be modeled in terms of the information content of an algorithm. An algorithm is constructed by making a sequence of design decisions. Each decision causes objects to be added, deleted, or modified in an algorithm. Thus, each decision has an impact on the information content of an algorithm. Normally, a programmer has a set of rules which govern what information may be used by a design decision, and what portions of an algorithm may be altered as a result of these decisions. These rules represent a <u>design methodology</u>. The result of any design decision, with the exception of the last decision, is a partially completed algorithm. Thus, the sequence of design decisions leads to a sequence of <u>partial algorithm designs</u>. By examining the information content of each of the partial designs, we can observe the effects of each design decision on the information in the current partial design.

1.2 Organization of the Dissertation

The first section suggests that parallel algorithms may contain additional information regarding process synchronization and intercommunication and thus may have higher information contents than their sequential counterparts. Is the amount of this additional information enough to suggest the need for modifications to traditional design strategies? In chapter two, an attempt is made to address this question by developing a set of metrics which will be used to show that indeed parallel algorithms may have a higher information content than their sequential counterparts. This will be done by examining a sequential root finding algorithm and a parallel version which was derived from it.

The metrics of chapter two can be used to quantify the information content of sequential and parallel algorithms. They were derived from those suggested by Halstead's software science [16]. The metrics as envisioned by Halstead, are normally used to describe completed algorithms. In Chapter three, it will be shown that our extensions to Halstead's metrics are also useful for describing the information content of partial designs. Thus, we will be able to quantify the effects of a design

decision, by comparing the information contents of suceeding partial designs. A convenient means of representing algorithms and their partial designs will also be presented in chapter three. A graph language based on the abstraction of an algorithm's information content will be used to demonstrate how design decisions change the information content of an algorithm.

Given that parallel algorithms may contain a significant amount of additional information, how is that added information distributed throughout an algorithm? There are two possible extremes for the placement of this added information.

1. The added information may be localized in one small section of an algorithm.

2. The added information may be distributed throughout an algorithm.

It is hypothesized that the concentration and placement of the added information may affect the approach one needs to take when designing a parallel algorithm. Chapter four examines the placement of this additional information, by examining a sequential and parallel version of a root finding algorithm. There it will be shown that for the parallel root finding algorithm, the added information is concentrated in a small, local portion of the algorithm.

The parallel root finding algorithm is an example of an algorithm whose parallel information is localized. Also, the parallel component of the algorithm is small. Chapter five will present a second parallel algorithm which also has localized parallel information. However, the parallel component of this algorithm is much larger. The algorithm solves the bin packing problem and is designed for multiprocessor systems. Thus, we will have examples of two algorithms with localized parallel information. The root finding algorithm has a small parallel component, and the bin packing algorithm has a large parallel segment. The two algorithms will be used to investigate the type of information which should be used to design varying sized algorithms with localized parallel components.

For algorithms, such as our two sample algorithms, with localized parallel information, how does the size of the parallel component affect the type of information which should be used by a design decision? In order to answer this question, we need to model design decisions in terms of the information which they use. In chapter six, a model of the algorithm design process will be presented. The model will focus on the use of spatial, temporal, and functional information for the design of parallel algorithms with localized parallel components. It will examine how the structure of the algorithm effects the kind of information which is needed to efficiently design it. The model will

simulate the design of an algorithm by reconstructing a graph language representation of it. The order in which objects are added to a design will be controlled by exploiting relationships which are based on the use of different types of information. By monitoring the number of steps it takes to complete the algorithm, and the information content of each of the partial designs, we will be able to observe the effects of the use of spatial, temporal, and functional information in the algorithm design process. An implementation of the model, in Pascal, for the Cyber 750 will also be discussed.

Chapter seven will describe a set of experiments which were conducted to examine the effects of using methodologies which differ in terms of their use of spatial, temporal, and functional information on algorithms with local parallel components. The use of this information will be examined relative to the granularity of an algorithm's parallel component. The root finding and bin packing algorithms will be the subjects of the experiments.

Finally, chapter eight will summarize the findings of this research. It will discuss the correspondence between the type of information used during the design process, and the size and placement of the parallel component of the algorithms. Suggestions will be offered for the development of parallel algorithms based upon this correspondence.

CHAPTER 2

Metrics For The Description of Algorithm Information Content

In order to effectively model and compare design strategies, it is necessary to have an objective criteria regarding the information content of an algorithm, and to have a means of monitoring the growth of information of an algorithm, as it moves from its conception through the design process. Several sets of metrics have been proposed for this purpose [15]. The metrics proposed by Halstead [16] seem to dominate and have led to the field of "Software Science".

Halstead uses the term software science to describe a set of metrics which are used to describe and compare algorithms. The metrics are based on counts of the elements which compose the algorithm. The metrics represent the number of bits which are required to uniquely encode an algorithm. As such, they represent the information content

of an algorithm.

The metrics as presented by Halstead, have been used to describe sequential algorithms. In [17] they were extended to allow their use with parallel algorithms. A brief description of Halstead's metrics will be followed by their extensions into the parallel realm.

2.1 Metrics For Sequential Algorithms

Halstead has proposed the following basic metrics for sequential algorithms.

n_1 = number of unique or distinct operators in an algorithm's implementation.

n_2 = number of unique or distinct operands in an algorithm's implementation.

N_1 = total count of the appearance of all operators in an implementation.

N_2 = total count of the appearance of all operands in an implementation.

$F_{1,j}$ = number of occurences of the jth most frequently used operator, where $j = 1, 2, \ldots, n_1$.

$F_{2,j}$ = Number of occurences of the jth most frequently used operand, where j = 1, 2, ..., n_2.

Also it is evident that:

$$N_1 = \sum_{j=1}^{n_1} F_{1,j}$$

and

$$N_2 = \sum_{j=1}^{n_2} F_{2,j}$$

As can be seen from the descriptions above, the metrics are based on simple element counts of a particular implementation of an algorithm. It should be clear that implementations of an algorithm in different languages will potentially yield different values for these metrics, showing that the information content of an algorithm is dependent on the language in which it is implemented.

2.1.1 An Example of Halstead's Metrics for a Sequential Algorithm

As an example of these metrics, an algorithm from [18] which evaluates a polynomial $p(x)=a_n x^n + a_{n-1}x^{n-1}+\ldots+a_1 x+a_0$ will be examined.

```
P := A[0];
POWERX := 1;
FOR I := 1 TO N DO
 BEGIN
   POWERX := X * POWERX;
   P := P + A[I] * POWERX
 END;
```

The operator and operand counts for the algorithm are given in Tables 2.1 and 2.2. Note, that the counts are based on a static description of the code, and that they ignore the number of references made to a particular piece of code, unlike the more traditional measures such as an algorithm's asymptotic behavior.

Using the basic metrics more sophisticated measures can be formed. The vocabulary of an algorithm is the sum of the number of unique operators, n_1, and unique operands, n_2. For the example algorithm, the vocabulary, n, is:

$$n = n_1 + n_2 = 15$$

The implementation length, N, of an algorithm is based on the number of occurences of the elements in the algorithm's vocabulary. Thus:

$$N = N_1 + N_2 = 32$$

The length of an algorithm and its vocabulary are often combined to form the volume metric. Volume represents the number of bits which are required to encode an algorithm. Thus, volume requires a component to uniquely represent each

Table 2.1 Operator counts for the polynomial program.

Operator	j	$F_{1,j}$
:=	1	5
;	2	4
*	3	2
[]	4	2
+	5	1
BEGIN...END	6	1
FOR...TO...DO	7	1
	n_1=7	N_1=16

Table 2.2 Operand counts for the polynomial program.

Operand	j	$F_{2,j}$
POWERX	1	4
P	2	3
A	3	2
I	4	2
1	5	2
0	6	1
X	7	1
N	8	1
	n_2=8	N_2=16

member of the vocabulary, $\log_2 n$. Plus, it needs to include the number of occurences (length) of those elements. Therefore, volume is expressed as:

$$V = N \log_2 n.$$

For the example algorithm this yields:

$$V = 32 \log_2 15 = 125.02 \text{ bits.}$$

2.1.2 Potential Volume

As mentioned earlier, the language in which an algorithm is implemented has an effect on the metrics. Because of this, a basis is needed to compare different implementations of a given algorithm. Potential volume, V*, is the metric used for this purpose. This represents the minimal information needed regarding an algorithm. This is comparable to calling a predefined procedure. The information which is needed are the input and output operands. The operators become assignment operators to the output operands, plus the function which is the purpose of the procedure. Thus, if we let n_1^* and n_2^* represent the minimal vocabulary and N_1^* and N_2^* represent the minimal length, the minimal or potential volume becomes:

$$V^* = (N_1^* + N_2^*) \log_2 (n_1^* + n_2^*)$$

For the polynomial algorithm this is:

$$V^* = (2+4)\ \log_2(2+4) = 15.5 \text{ bits.}$$

We assume three input operands (A, N, and X) and one output operand (P). The operators are the polynomial evaluation function and the assignment operator for the result operand.

Knowing that the most succint form of our algorithm is 15.5 bits, it can now be compared to the acutal implementation volume of 125.02 bits. The resulting metric is called the implementation level, L.

$$L = \frac{V^*}{V} = \frac{15.5}{125.02} = .124$$

Note that the closer L is to one, fewer details regarding the algorithm are required. If L equals one, the algorithm must be present as a component of the language which is being used. As L becomes smaller, the implementation volume must increase. Since implementation volume represents information, one would suspect such an algorithm required more design steps, with a resultant increase in the design time and the number of expected design errors. L can also be used to compare two implementations of an algorithm. The version which has the smaller value for L can be assumed to be more complex, and require more design effort.

This concludes the discussion of metrics for sequential algorithms. In [16] other uses for these metrics are given. It is shown there that the metrics are useful for predicting

development and maintenance time and effort. The next section will describe extensions to these metrics for use with parallel algorithms.

2.2 Metrics For Parallel Algorithms

The metrics described in [17] for parallel algorithms are direct extensions of their sequential counterparts. As such, they are also based on element counts. However, there is a difference in how the elements are to be counted. In chapter one, it was argued that the design of parallel algorithms require information above their sequential counterparts. This information includes knowledge of process intercommunication, insuring mutual exclusion where required, and so forth. This new information was not required in designing sequential algorithms. As a result of this, for our purposes in analyzing an algorithm's information content, we will view an algorithm as being composed of two components; a sequential component, and a parallel component. Thus, the element counts become:

$n_{1,p}$ = a count of those unique operators that represent the activities of parallel processes, their synchronization, and access to shared data.

$n_{1,s}$ = a count of those unqiue operators that represent the basic sequential activities performed by an algorithm and are not invoked be a parallel operator.

$n_{2,p}$ = a count of those unqiue operands used by parallel operators including synchronization and resource sharing operators. This includes all operands associated with the invocation of a parallel operator.

$n_{2,s}$ = a count of those unqiue operands that are used in the algorithm's basic sequential computational activities, excluding those invoked by a parallel operator.

When dealing with these elements, the designer must be cognizant of the domain in which the elements are found (sequential or parallel), since they are treated seperately in each domain. Thus, the simple element counts will be superceded by counts of operator-context and operand-context pairs. Thus, we now include <u>conceptual</u> differences between sequential elements and possible parallel counterparts. Therefore, if an element is found in both a sequential and a parallel portion of the algorithm, for purposes of the vocabulary, it is counted twice. It is counted as belonging to the sequential component, and also as belonging to the

parallel component. The vocabulary is not inflated, because now the vocabulary measures unique operator/operand-context pairs.

If we return to the polynomial algorithm of the last section, we notice that it is composed of only a sequential component. Thus, we can easily see that the sequential metrics of the first section are a special case of the parallel metrics, with $n_{1,p}$ and $n_{2,p}$ both equal to zero. Since sequential algorithms can be described in terms of the new metrics, the metrics will be useful in comparing a parallel algorithm to its sequential counterparts, as well as for comparing two parallel algorithms.

As before, the total number of occurences of element-context pairs yield the metrics shown below.

$N_{1,p}$ = the total count of parallel operators and those sequential operators invoked by them.

$N_{1,s}$ = the total count of those sequential operators not invoked by a parallel operator.

$N_{2,p}$ = total number of occurences for operands used by parallel operators. This includes operands involved in sequential actions invoked by a parallel operator.

$N_{2,s}$ = total number of occurences of operands used in sequential activities not invoked by parallel operators.

$F^j_{1,p}$ = number of occurences of the jth most frequently used operator in this class.

$F^j_{1,s}$ = number of occurences of the jth most used sequential operator.

We can define operands $F^j_{2,s}$ and $F^j_{2,p}$ in the same fashion. It follows that:

$$N_{1,s} = \sum_{j=1}^{n_{1,s}} F^j_{1,s}$$

and

$$N_{1,p} = \sum_{j=1}^{n_{1,p}} F^j_{1,p}$$

$N_{2,s}$ and $N_{2,p}$ can be similarly defined.

The total number of operators and operands will be designated N_1 and N_2 respectively, where:

$$N_1 = \sum_{i=s,p} \sum_{j=1}^{n_{1,i}} F_{1,i}^{j}$$

$$N_2 = \sum_{i=s,p} \sum_{j=1}^{n_{2,i}} F_{2,i}^{j}$$

Also, the number of unique operators and operands are given as n_1 and n_2. They are defined as:

$$n_1 = \sum_{i=s,p} n_{1,i}$$

$$n_2 = \sum_{i=s,p} n_{2,i}$$

The sum of the number of unique operators and operands in each category for an algorithm is the vocabulary of the parallel algorithm. We again designate it as n, where:

$$n = n_1 + n_2$$

Likewise, the length, N, of a parallel algorithm equals:

$$N = N_1 + N_2$$

2.2.1 Comparison of a Sequential and a Parallel Root Finding Algorithm

To illustrate the use of the new metrics in comparing two algorithms, a sequential and a parallel version of a

root finding algorithm will be presented and analyzed using the new metrics. The sequential algorithm taken from [19] is shown below. The algorithm assumes that the function, which is to be evaluated, is monotonically increasing and has a root in the interval [LOW, HIGH].

```
real procedure bisect(LOW, HIGH, E)
begin
 real Z, X, LOW, HIGH, E, Y;
 while (HIGH - LOW) > 2 * E  do
   begin
    X := (LOW + HIGH) / 2;
    Y := F(X);
    if Y < 0
         then HIGH := X
         else LOW  := X
   end;
 Z := (LOW + HIGH) / 2
end;
```

The element-context pair counts are shown in Tables 2.3 and 2.4. Since it is a sequential algorithm, all parallel metrics equal zero, and the metrics correspond to the standard sequential versions.

Next, a parallel version of the algorithm is given. The parallel component is that block which composes the "PARFOR" statement and the definitions of the shared variables.

```
real procedure rootf(LOW, HIGH, N, E)
begin
 integer shared N, J;
 integer I;
 real shared DELTA, LOW, Y[1..N];
 real E, Z, HIGH;
 while ABS(HIGH - LOW) > E do
  begin
   DELTA := ABS(HIGH - LOW) / (N + 1);
```

```
    parfor J := 1 until N do
     begin
      Y[J] := F(LOW + (J * DELTA))
     end;
    LOW := LOW + DELTA;
    I := 1;
   while 0 > Y[I] and I <= N do
     begin
      LOW := LOW + DELTA;
      I := I + 1
     end;
   HIGH := LOW + DELTA
  end;
  Z := (LOW + HIGH) / 2
end;
```

Tables 2.5 and 2.6 give the element-context counts for algorithm ROOTF.

Using the counts given in the tables, it can be seen that the vocabularies of the respective algorithms are:

$$n = n_{1,p} + n_{2,p} + n_{1,s} + n_{2,s}$$

BISECT	0 +	0 +	13 +	8	= 21
ROOTF	10 +	6 +	15 +	11	= 42

The lengths of the two algorithms are:

$$N = N_{1,p} + N_{2,p} + N_{1,s} + N_{2,s}$$

BISECT	0 +	0 +	32 +	29	= 61
ROOTF	16 +	15 +	50 +	39	= 120

Thus, we can immediately observe that the parallel algorithm has twice the vocabulary and length of its sequential counterpart.

As with the vocabulary and length metrics, there is a new counterpart to the volume metric. Now volume refers to

Table 2.3 Operator counts for the bisection algorithm.

Operator	rank (i)	$F^i_{1,p}$
:=	1	8
begin...end, (...)	2	7
;	3	5
+	4	2
/	5	2
-	6	1
>	7	1
*	8	1
while...do	9	1
if...then...else	10	1
F	11	1
real	12	1
real procedure bisect	13	1
	$n_{1,s}$=13	$N_{1,s}$=32

Table 2.4 Operand counts for the bisection algorithm.

Operand	rank (i)	$F^i_{2,s}$
HIGH	1	6
LOW	2	6
X	3	5
E	4	3
Y	5	3
2	6	3
Z	7	2
0	8	1
	$n_{2,s}$=8	$N_{2,s}$=29

Table 2.5 Operator counts for the multiprocessor algorithm.

Parallel Operator	rank (i)	$F^i_{1,p}$
;	1	3
begin...end, (...)	2	3
[]	3	2
:=	4	2
+	5	1
*	6	1
parfor...until...do	7	1
F	8	1
real shared	9	1
integer shared	10	1
	$n_{1,p}$=10	$N_{1,p}$=16

Sequential Operator	rank (i)	$F^i_{1,s}$
:=	1	11
;	2	8
begin...end, (...)	3	9
+	4	6
while...do	5	2
>	6	2
-	7	2
ABS	8	2
/	9	2
≤	10	1
[]	11	1
integer	12	1
real	13	1
real procedure rootf	14	1
and	15	1
	$n_{1,s}$=15	$N_{1,s}$=50

Table 2.6 Operand counts for the multiprocessor algorithm.

Parallel Operand	rank (i)	$F^i_{2,p}$
J	1	4
N	2	3
LOW	3	2
DELTA	4	2
Y	5	2
1	6	2
	$n_{2,p}=6$	$N_{2,p}=15$

Sequential Operand	rank (i)	$F^i_{2,s}$
LOW	1	9
I	2	6
HIGH	3	6
DELTA	4	4
1	5	3
E	6	3
N	7	3
Z	8	2
Y	9	1
2	10	1
0	11	1
	$n_{2,s}=11$	$N_{2,s}=39$

the number of bits required to encode conceptual discriminations which an algorithm's designer must make. This new metric, which is called the conceptual volume, $\underline{V}$, is:

$$\underline{V} = (N_s + N_p) \log_2(n_s + n_p).$$

As with the other metrics, the sequential metrics presented earlier are a special case with N_p and n_p equal to zero.

For our example root finding algorithms, $\underline{V}$ is shown below.

$$\text{BISECT } \underline{V} = (61 + 0) \log_2(21 + 0) = 267.93$$
$$\text{ROOTF } \underline{V} = (89 + 31) \log_2(26 + 16) = 647.07$$

2.2.2 Component Metrics and Potential Conceptual Volume

At times, it will be useful to examine the parallel or sequential portion of an algorithm only. The resulting metrics are known as component metrics. Thus, the component conceptual volume for the parallel portion of an algorithm is calculated as below.

$$\underline{V}_p = (N_{1,p} + N_{2,p}) \log_2(n_{1,p} + n_{2,p})$$

Thus, for ROOTF:

$$\underline{V}_p = (16 + 15) \log_2(10 + 6) = 124$$
$$\underline{V}_s = (50 + 39) \log_2(15 + 11) = 418.34$$

This can be used to see that the bulk of the information resides in the sequential portion of the ROOTF algorithm.

Next, the concept of potential volume will be examined in the parallel context. The reader should recall that the potential volume represented the minimal expression of an algorithm, equivalent to calling a predefined procedure. Thus, one would not expect there to be a difference in the potential volume whether the underlying procedure was sequential or parallel, since all that is seen is the procedure call. A procedure call would be the same for procedures which are either sequential or parallel. In either case, we assume the parameters passed and two operators (the function and assignment operators) belong in the sequential domain. Finally, the potential conceptual volume, $\underline{V}^*$ can be expressed as:

$$\underline{V}^* = (N^*_s + N^*_p) \log_2(n^*_s + n^*_p)$$

$$= (N^*_s) \log_2(n^*_s)$$

Which for both BISECT and ROOTF is:

$$\underline{V}^* = (2 + 4) \log_2(2 + 4) = 15.51 \text{ bits.}$$

We assume Z, LOW, HIGH, and E are the parameters, and as before there are two operators.

2.3 Summary

This chapter began by reviewing Halstead's metrics which describe the information content of sequential algorithms. Extensions were introduced to allow the metrics to describe the information content of parallel algorithms as well. Two algorithms, BISECT and ROOTF, were used to show that the extended metrics can be used to compare sequential and parallel algorithms. By comparing the two algorithms, it can be seen that the information content of the parallel algorithm is greater than that of its sequential counterpart. Thus the prediction, which was made in chapter one, that parallel algorithms may need to include additional information to handle process synchronization, process intercommunication, etc., is supported. Although we have only examined a single algorithm, one of our basic questions is answered. That is, a parallel algorithm may have an increased information content over its sequential counterparts. If an increase in information is seen for an algorithm as simple as the root finder, one would suspect that more complex algorithms would show an even larger increase in their information contents. The remainder of this research will examine whether this increase in an algorithm's information content has an impact on the design

process for such algorithms.

CHAPTER 3

The Graph Model of Parallel Algorithm Structure

In chapter two, metrics were presented for describing parallel algorithms. These metrics were based on counts of the elements of the language in which the algorithm was expressed. Thus, the metrics represent the information content of a particular implementation of an algorithm. As designed, the metrics include counts of non-active elements, i.e., BEGIN/END pairs, semi-colons, etc.. In seeking to abstract out only those algorithm features which actively contribute to and describe the information needed to design an algorithm, a graphical model of parallel algorithm structure was developed.

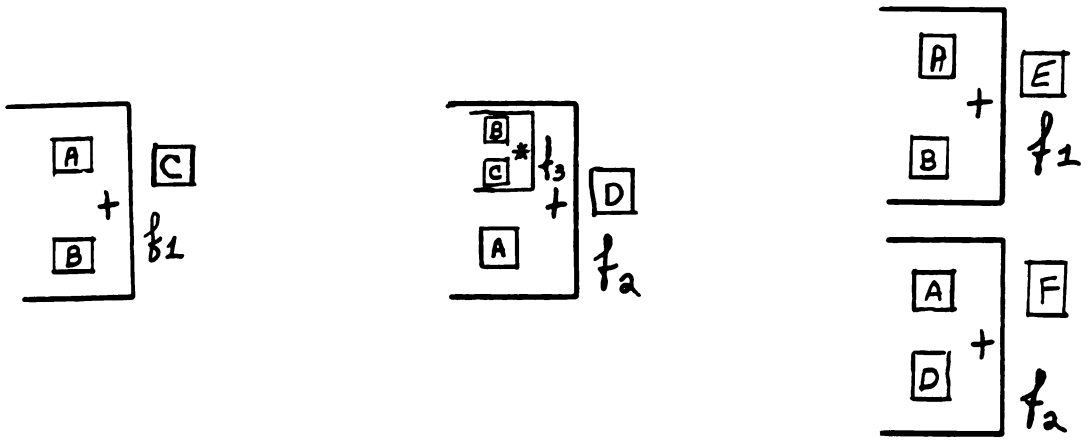
In [20] Reynolds and Chang describe the PAL (Parallel Algorithm Laboratory) system. PAL is a graph language which is used to describe parallel algorithms. PAL allows both completed algorithms and algorithms still in the design

process to be represented. In its original intent, it was to describe algorithms for VLSI array based architectures, but it has proved useful for describing algorithms designed for other parallel architectures as well. A description of PAL, as extended for this research, follows in the next section.

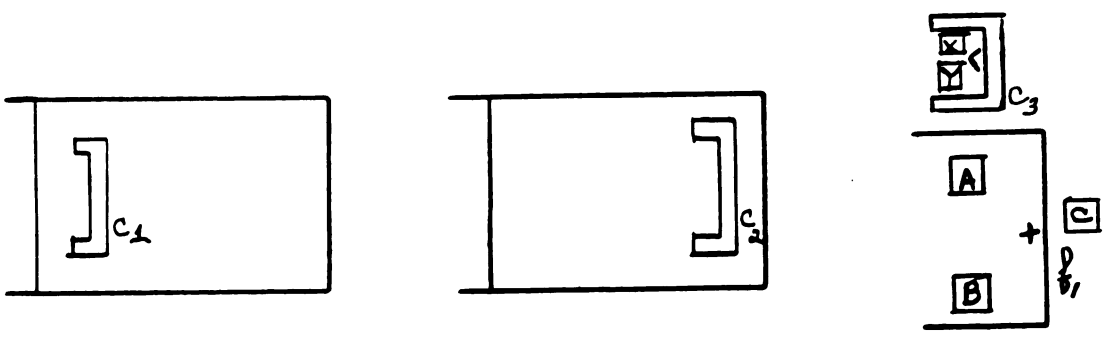
3.1 The Graph Language

The graph language is based on a functional representation of an algorithm. It consists of data objects and structures. Data objects are given a label (name) and flow where needed in the completed graph in a data-flow fashion. Data objects are represented, in the graph, by closed rectangles, Figure 3.1. Structures represent manipulation of data objects and the control of an algorithm. The simplest structure is the function, Figure 3.1a. It consists of operands (data objects) and an operator, producing a result, which optionally may be given a label or used directly as an operand, when the functions are nested, Figure 3.1b. Functions may be nested to any level. Parallelism between functions is indicated by stacking the functions vertically, as shown in Figure 3.1c. Control in the graph proceeds from left to right.

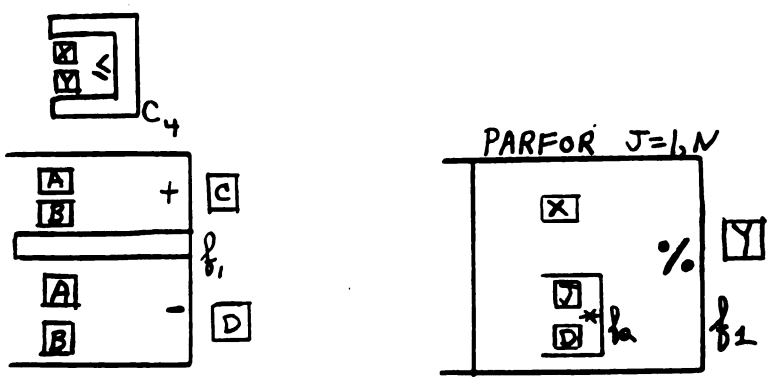
Control structures in the graph language consist of closed rectangle-like like structures as shown in Figures 3.1d and 3.1e. The backward block C represents a logical



A. B. C.



D. E. F.



G. H.

Figure 3.1 The structures of the graph language.

condition. In Figure 3.1d the condition operates as a pretest (DO WHILE), and in Figure 3.1e, it operates as a post test (REPEAT UNTIL). Any structure may be nested inside of a control block, including other control blocks.

In Figure 3.1f, another use for the condition block is shown. In this example, it serves as an IF THEN like structure. In Figure 3.1g, it is shown representing an IF THEN ELSE like structure. The use of the conditional block is not restricted to the function structures, any structure may be governed by such a conditional.

An additional control construct remains, the PARFOR structure. This structure allows an arbitrary number of copies of a structure to be placed in parallel. It is shown in Figure 3.1h. N copies of function f_1 will proceed in parallel, when this structure is encountered.

Figure 3.2 shows how the structures can be combined to describe a complete algorithm. Several features of the graph language can be seen in Figure 3.2. First, the placement and amount of parallelism in the algorithm is easily observed. Second, the graph language forces the designer to construct his algorithms in a modular form. If one attempted to construct a complex algorithm in a non-modular fashion using this notation, it would be visually confusing. For purposes of this research, global and local data are represented the same. However, the graph language could be easily extended to allow discriminations

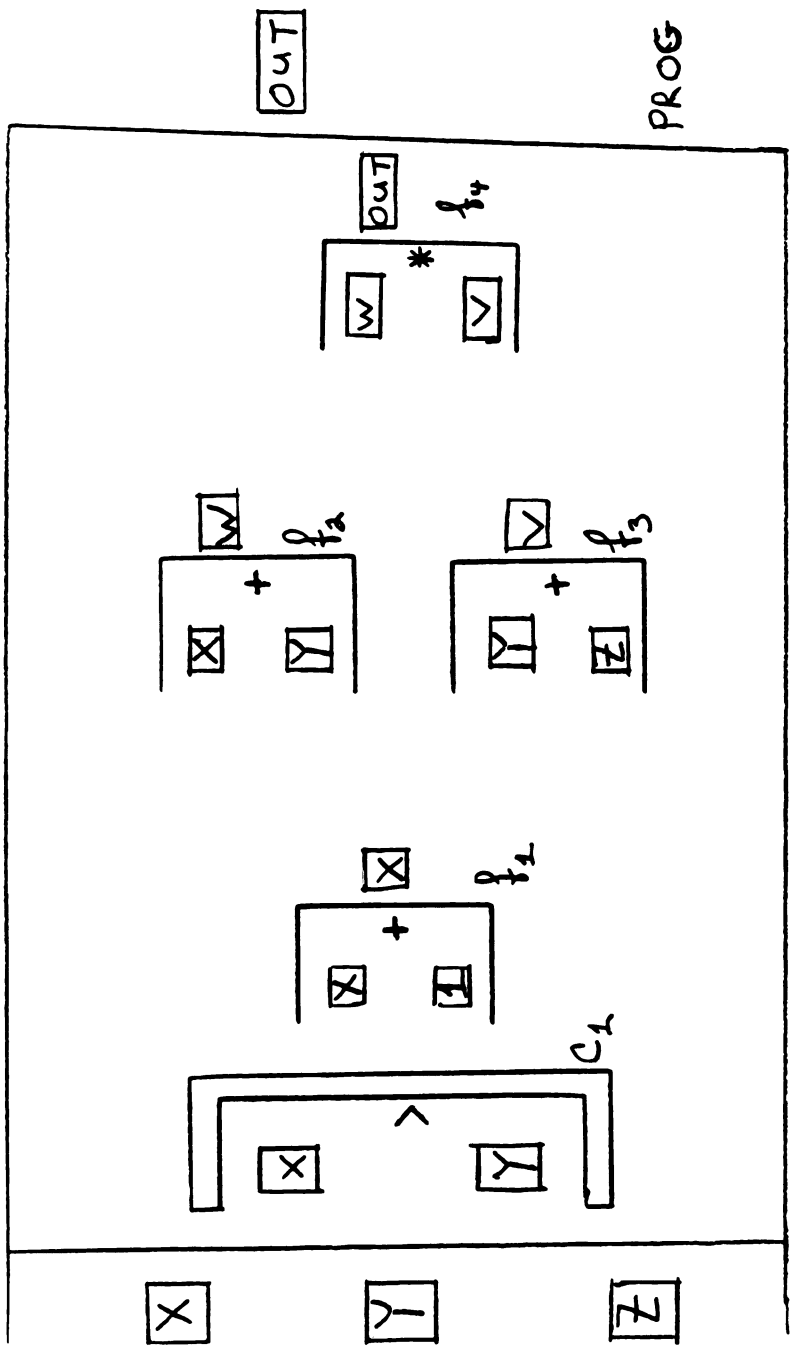


Figure 3.2 A sample graph language algorithm.

between them.

3.2 Relation of Graph Language to the Metrics

The structures of the graph language correspond to design decisions which must be made when designing an algorithm. As such, each element is assumed to be an abstraction of a piece of information which describes the algorithm. Thus, there is a correspondence between the graph elements and the metrics of chapter two, which describe an algorithm's information content.

When a designer creates an algorithm, he uses many pieces of information, and introduces much of that information into the algorithm. The first piece of information, which is required, is the realization of the need for a structure in the algorithm. This is represented in the graph language by the structure's label. Second, the need for a structure is determined because its results are needed. Finally, the designer must be concerned with those elements which make up the body of the structure, the input operands, the operator(s), and the conditionals, if they exist. This procedure is quantified, for purposes of the metrics, by counting the structure's label, results, input operands, operator, and conditionals. The counts are combined, as in chapter two, to form the conceptual volume and other metrics.

Additionally, the graph language directly supports the notion of context. There are two distinct components to any graph; those structures and operands which are strictly sequential and those which are in parallel portions of the algorithm.

Table 3.1 shows the element counts for the algorithm in Figure 3.2. It also shows how the counts are combined to form the conceptual and component volumes for the algorithm.

3.3 Metrics for Partial Designs

The metrics of chapter two are normally used to describe completed algorithms. In analyzing design methdologies, it is important to be able to determine the effects of each design decision or step. The graph language and metrics presented in chapter two, allow this to be easily done.

Since the metrics represent the information content of an algorithm, and partially completed algorithms, partially represent the description of their completed designs, the metrics can be used to describe the information content of the partial design at each stage of the algorithm design process. By comparing the information content at each stage of an algorithm's development, the growth in the information content of the algorithm during the design process can be

Table 3.1 Element counts and conceptual component volumes for the algorithm in Figure 3.2.

Sequential Elements	rank (i)	$F^i_{2,s}$		rank (i)	$F^i_{1,s}$
X	1	4	PROG	1	1
Y	2	2	C1	2	1
OUT	3	2	f1	3	1
Z	4	1	f4	4	1
1	5	1	>	5	1
W	6	1	+	6	1
V	7	1	*	7	1
	7	12		7	7

Parallel Elements	rank (i)	$F^i_{2,p}$		rank (i)	$F^i_{1,p}$
Y	1	2	+	1	2
X	2	1	f2	2	1
Z	3	1	f3	3	1
W	4	1			
V	5	1			
	5	6		3	4

$\underline{V}_s = (12+7) \log_2 (7+7) = 72.3$

$\underline{V}_p = (6+4) \log_2 (5+3) = 30.0$

monitored.

Figure 3.3 shows a partially completed algorithm. It is recognized that "PROG" has three inputs, one output, and is to be constructued as a DO WHILE like block. At this stage of the design process, the structure has an information content of 15.5 bits (Table 3.2).

Each additional element added to "PROG", will change that information content. By monitoring the metrics, we can observe the change in the information content, as the design process proceeds. The values obtained for the metrics at each stage of the design process are called <u>partial</u> metrics, PV represents partial volume, and PL represents partial length. The difference between the partial metrics at succeeding design steps are called <u>added</u> metrics, i.e., volume added or length added.

$$V_{added} = PV_n - PV_{n-1}$$
$$L_{added} = PL_n - PL_{n-1}$$

Thus, the resulting partial volume and length for the algorithm in Figure 3.4 is 25.2 bits. The volume added from the design step which led from Figure 3.3 to Figure 3.4 is 9.7 bits (Table 3.3).

Although the term 'added' implies that the volume increases with each design step, this may not always be the case. The designer may backtrack in his design, or realize that what originally appeared to be distinct objects are the

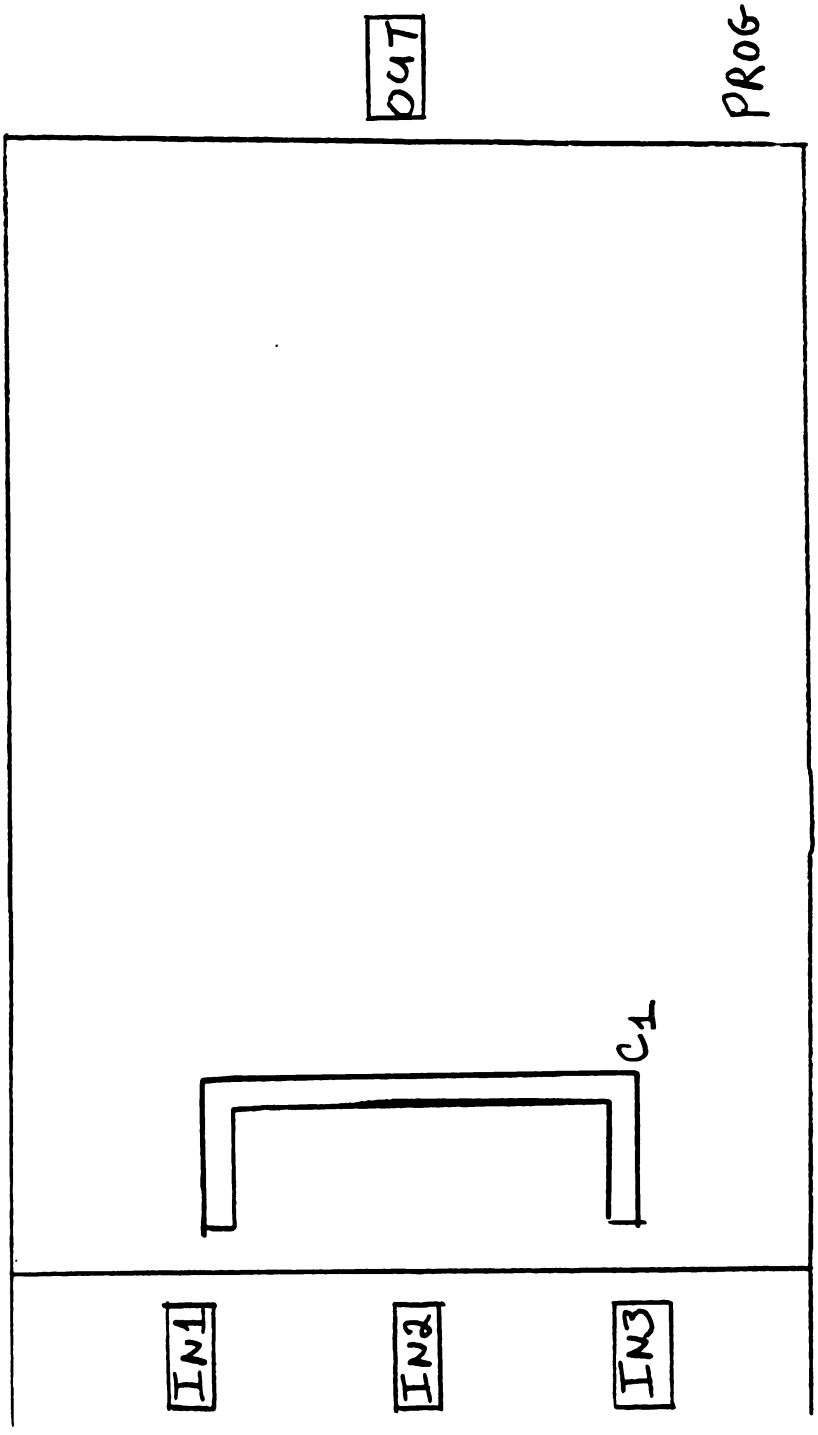


Figure 3.3 A partial algorithm design.

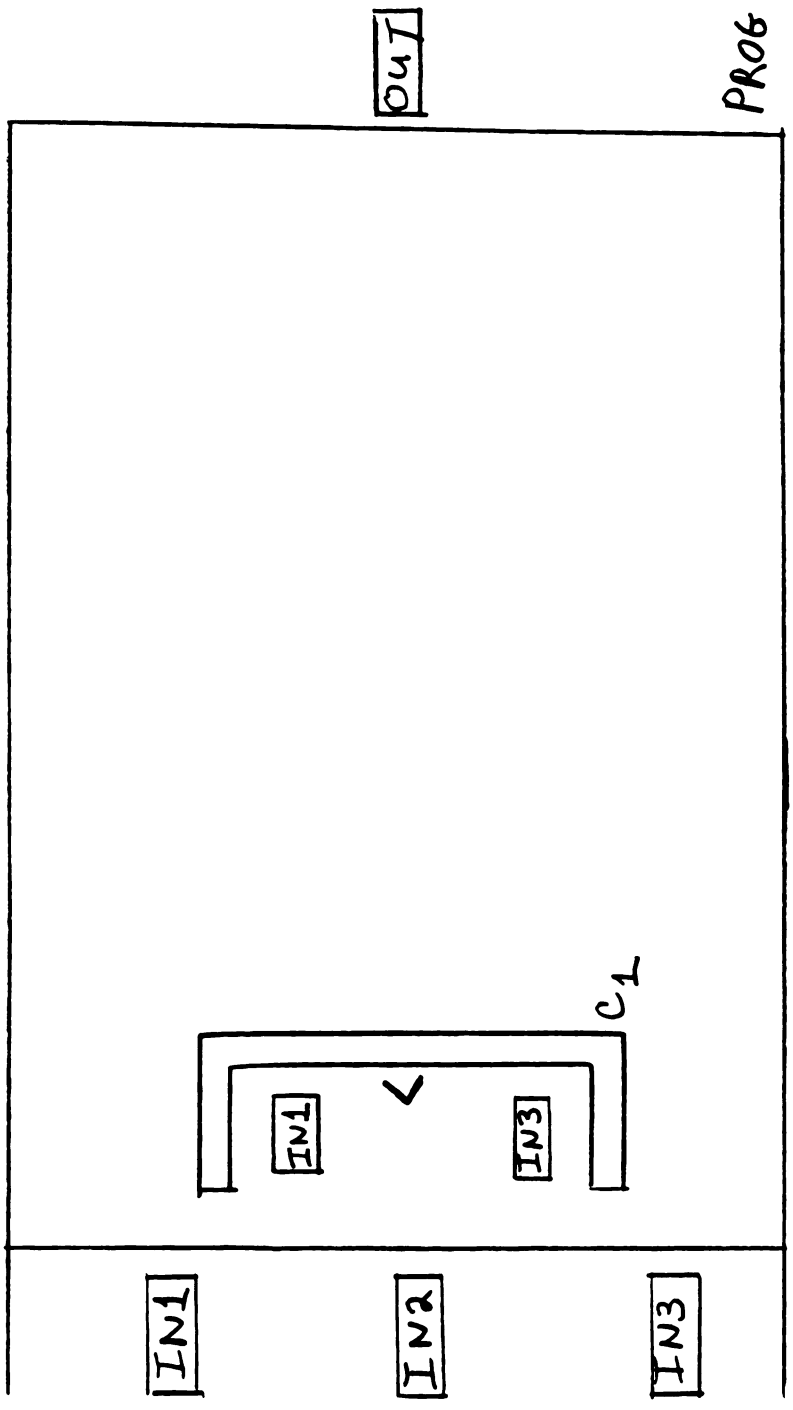


Figure 3.4 The algorithm of Figure 3.3 after the next design step.

Table 3.2 Element counts for the algorithm of Figure 3.3.

Sequential Elements	rank (i)	$F^i_{2,s}$		rank (i)	$F^i_{1,s}$
IN1	1	1	PROG	1	1
IN2	2	1	C1	2	1
IN3	3	1			
OUT	4	1			
	4	4		2	2

$\underline{V}_s = (4+2) \log_2 (4+2) = 15.5$

Table 3.3 Element counts for the algorithm of Figure 3.4.

Sequential Elements	rank (i)	$F^i_{2,s}$		rank (i)	$F^i_{1,s}$
IN1	1	2	PROG	1	1
IN3	2	2	C1	2	1
IN2	3	1	<	3	1
OUT	4	1			
	4	6		3	3

$\underline{V}_s = (6+3) \log_2 (4+3) = 25.2$

$\underline{V}_{s,\text{added}} = \underline{V}^2_s - \underline{V}^1_s = 9.7$

same. Either of these situations could cause the design step to reduce the partial conceptual volume for an algorithm. Thus V_{added} and L_{added} may have negative values.

3.4 Summary

This chapter introduced a graph language which will be used to represent algorithms in a model of the parallel algorithm design process. The graph elements are based on an abstraction of an algorithm's information content, and thus have a strong correspondence to the metrics discussed in chapter two. Halstead's metrics are normally used to describe completed algorithms. This chapter has shown that our extensions to them are also useful for describing the information content of algorithms during the design process. Thus, we can now observe the effects of a design decision on the information content of an algorithm.

CHAPTER 4

Algorithm Transformations and Subsidization

In chapter two, two versions of a root finding algorithm were introduced. There it was shown that the parallel version contained significant additional information, above that found in the sequential algorithm. This chapter will examine the placement of that additional information.

Parallel algorithms, such as the root finder, may be developed using several approaches [3]. One approach is to start with a sequential algorithm and make a set of simple transformations to it. Such transformations are often based on isolating and replacing a small sequential segment of the algorithm with a parallel segment. When a replacement of this nature is made, how does this impact the structural complexity of the algorithm, and are there interaction effects between the parallel and sequential components? The

nature of these interactions will be interpreted in terms of algorithm transformations.

4.1 Algorithm Transformations

Algorithm transformations can be categorized into four types (Figure 4.1), based upon changes in an algorithm's input and output sets. A type 1 transformation is the simplest. A single parallel code segment replaces a sequential segment. The parallel segment receives, as input, a single data object and produces, as output, a single data object. A type 2 transformation differs, in that it receives multiple input objects, but only produces a single output object. This is indicative of log sum reduction transformations. Many of the automatic transformations suggested by Kuck [3] are of this type. The parallel root finding algorithm is an example of a type 3 transformation. A single data object is input, the search interval, and multiple data objects are output, the function values at the interval's midpoints. Finally, a type 4 transformation involves multiple inputs as well as multiple outputs. This type of transformation would be expected with many matrix problems, such as calculating determinents.

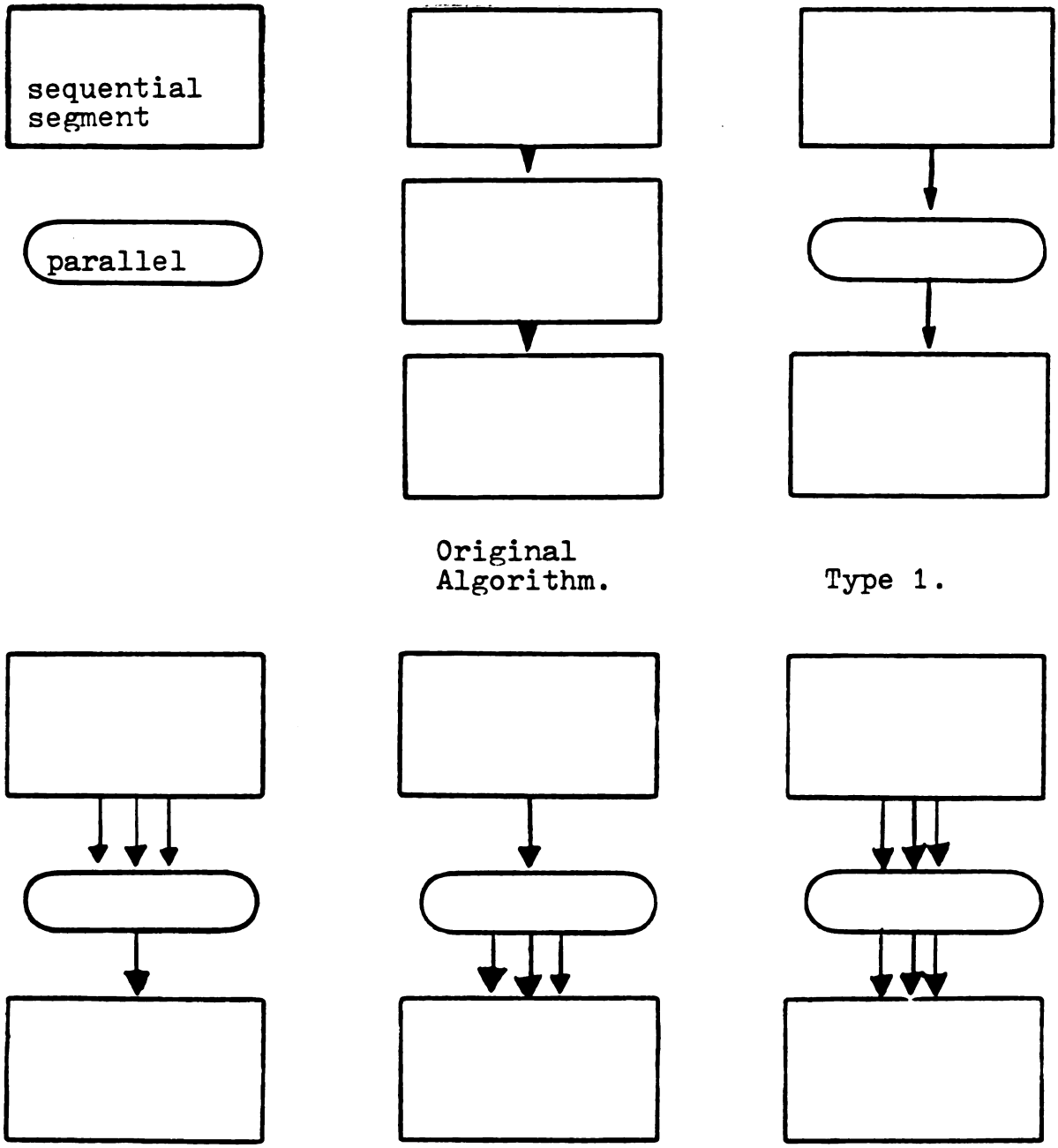


Figure 4.1 Transformations for converting a sequential algorithm to a parallel algorithm.

4.2 Subsidization

If we return to the root finding algorithms of chapter two, we can observe some features of these transformations. Recall that in creating the parallel root finding algorithm, a small sequential segment was replaced by a parallel segment. In chapter two, it was observed that the parallel algorithm had a higher information content than the sequential algorithm. We will now look at the two algorithms in an attempt to determine what portions of the parallel algorithm were responsible for this increase in conceptual volume. Table 4.1 shows the component lengths and vocabularies of the two algorithms.

Table 4.1 Component lengths and vocabularies for the two root finding algorithms.

	n_s	n_p	N_s	N_p	N	n
sequential, BISECT	21	0	61	0	61	21
parallel, ROOTF	26	16	89	31	120	42

By examining Table 4.1, it can be seen that the length, N, of the parallel algorithm is approximately double that of the sequential algorithm. As expected, much of this increase is due to the inclusion of the parallel comnponent.

N_p accounts for half of the increase in length. Since a small sequential segment of the sequential root finding algorithm was replaced, one would expect that the length of the sequential component of parallel algorithm would have decreased from that of the sequential algorithm. However, this did not happen. Not only was there no reduction in N_s, it actually increased by almost 50%. In chapter two, the conceptual component volumes were calculated for the two algorithms. For ROOTF $\underline{V}_p$ was 124 bits and $\underline{V}_s$ was 418.34 bits. Recall that the conceptual volume of BISECT was 267.93 bits and that the conceptual volume of ROOTF was 647.07 bits. While component volumes are not additive, they demonstrate that the bulk of the increase in volume was due to an increase in the sequential component of the algorithm. Thus, the increased information in the ROOTF algorithm was caused by the addition of the parallel component to the algorithm, plus by the an increase in the sequential component as well.

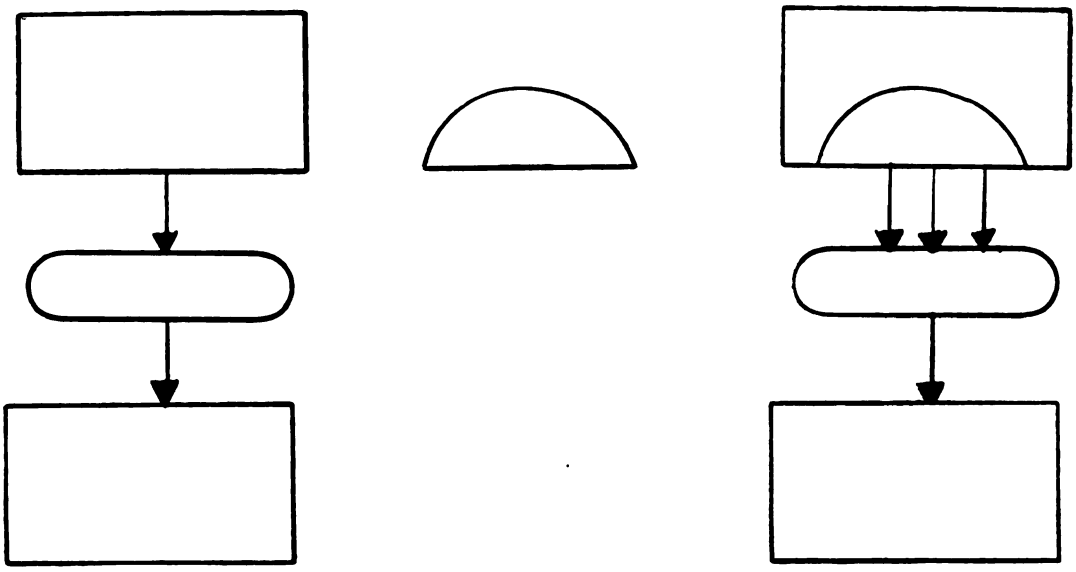
Another observation can be made by viewing the usage index, the ratio of N_i and n_i for the ith component (Table 4.2).

Table 4.2 Usage indices for the two root finding algorithms.

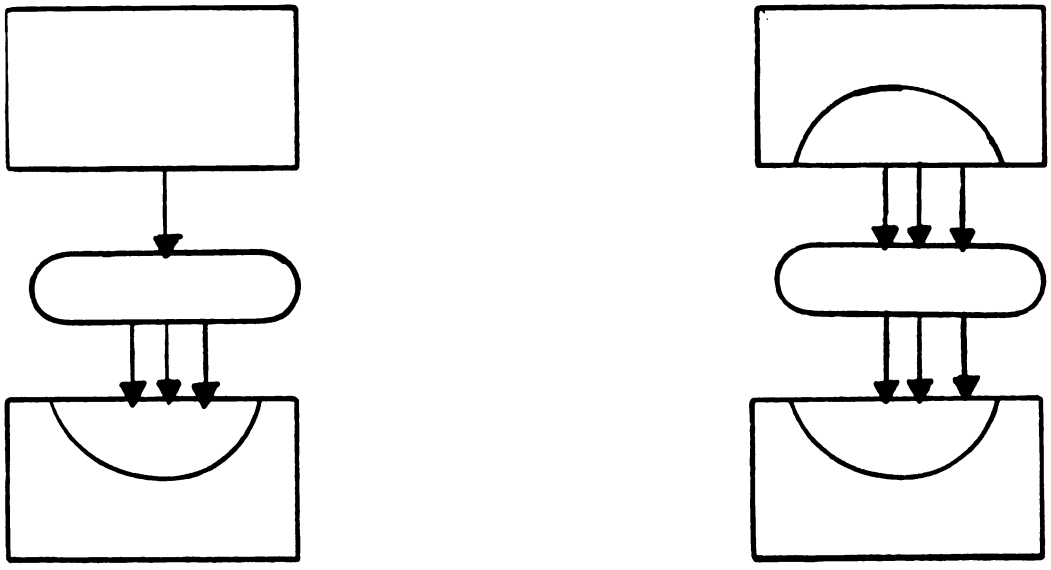
	$\frac{N_s}{n_s}$	$\frac{N_p}{n_p}$	$\frac{N}{n}$
BISECT	2.90	*	2.90
ROOTF	3.42	1.94	2.86

For both algorithms, the overall usage indices N/n are about the same. However, while the overall indices are about the same, for algorithm ROOTF, the sequential usage index, N_s/n_s has increased, and is larger than the parallel usage index, N_p/n_p. The reason for this can be seen by examining Figure 4.1. The single output data object of BISECT has been replaced by multiple outputs from ROOTF. The final sequential segment of ROOTF must collect and analyze these multiple outputs. This results in an increase in size for the sequential component. We say that the sequential component is subsidizing the parallel component.

Subsidization potentially occurs in three of the transformation types presented earlier. It is present in the sequential code and it serves to support the parallel portion of the algorithm by either preparing input for the parallel segment, or by collecting output from the parallel segment. However, the position of subsidization will vary depending on the type of transformation which took place (Figure 4.2).



Type 1. Subsidization. Type 2.



Type 3. Type 4.

Figure 4.2 The placement of subsidization for the four algorithm transformation types.

4.3 Summary

One method of transforming a sequential algorithm into parallel is to replace a sequential segment of the algorithm with a parallel segment. When this is done, we have shown that information may be added to the algorithm. Contrary to expectation, our examination of the two root finding algorithms has shown that the information content of the algorithm may increase not only because a parallel segment is added to the algorithm, but also because the sequential component of the algorithm increases in size as well. The sequential component increases in size, because an interface may be needed between the original sequential statements and the parallel component. It was seen here that when a single sequential segment was replaced by a single local parallel segment, that the sequential code which provides the interface may be localized as well. The support provided by this interface was called subsidization. The possible presence of subsidization in a parallel algorithm, suggests that a designer must pay particular attention to the interfaces between the sequential and parallel components. Further research is needed to examine algorithms with multiple parallel components. It is anticipated that subsidization effects will be seen there as well. However,

the form they take may differ. Multiple parallel segments may not need sequential subsidization segments between them, or a single subsidization segment may serve several parallel components. Whatever form subsidization takes for algorithms with multiple parallel components, one suspects its presence will effect the design process for such algorithms.

CHAPTER 5

A Parallel Bin Packing Algorithm

In the introduction we indicated that we would conduct experiments on the design process for parallel algorithms. Parallel algorithms with a single localized parallel component will be used in the experiments. The ROOTF algorithm, which has been discussed in Chapters two and four, will serve as an example of such an algorithm with a small localized parallel component. This chapter will examine the other algorithm, which we will use in our experiments. Like the ROOTF algorithm, it has a single localized parallel component. However, the size of the parallel component is larger. Thus, we will have two algorithms for our experiments, one with a small parallel component, and one with a large parallel component. The second algorithm solves a version of the bin packing problem. Like the ROOTF algorithm, it is designed for use

on a multiprocessor system.

The bin packing problem is a combinatorial problem. Combinatorial algorithms by their nature are computationally intensive. Exact solutions to them are often computationally intractable. Therefore, heuristic algorithms are generally used when confronted with such problems. One of the limiting factors in designing heuristic algorithms, is the amount of computations they involve. Therefore, combinatorial problems are good candidates for solution by parallel algorithms.

Bin packing is an example of such a problem. Bin packing in the one dimensional case, is an assignment problem [21-27]. Objects of various sizes are to be placed into a set of bins of limited capacity, such that the minimum number of bins are used. There are two versions of the bin packing problem. The first is the on-line version. In on-line bin packing the objects must be placed into the bins one at a time, as they are presented to the algorithm. This is opposed to off-line bin packing, where all items are presented before the algorithm attempts to pack the items. The off-line version allows the algorithm to condition the data before it processes it. The most common form of conditioning used is to sort the data. It has been shown that the worst case performance of off-line bin packing algorithms is significantly better than that for on-line ones [21].

The remainder of this chapter will present and discuss a parallel on-line bin packing algorithm. Its packing performance will be compared with that of sequential algorithms, both on-line and off-line.

5.1 The Bin Packing Problem

The bin packing problem consists of placing objects of particular sizes into a set of standard size bins. In the one dimensional case, the objects are characterized by one parameter. Objects can be of size between 0 and 1. For a given problem, the algorithm will be presented with a list of items, L, of length n, $0 < a_i \leq 1$, for each i, $1 \leq i \leq n$, where a_i represents the ith element of list L. The same value may occur in more than one position in the list.

The bins into which the items will be packed, are specified by a set of indices, 1, ..., m, where B_1 is called the first bin and B_m represents the last nonempty bin used. For any bin B_j the sum of a_i packed into it cannot exceed 1. The current sum of a_i stored in bin B_j is called the content of bin j, and is denoted as $c(B_j)$. The amount of unused space in a bin is called the gap, and is given as $1-c(B_j)$.

Several sequential heuristic algorithms have been proposed for on-line bin packing [22-25]. These include the

first-fit, refined first-fit, and best-fit approaches. The best-fit is of special interest to us here. In the best-fit, BF, approach a new item is assigned to a bin B_j such that the gap in bin B_j will be minimum over all currently nonempty bins. It has been shown that the worst case packing generated by the best-fit method is:

$$1.7\ L^* - 2 \leq BF(L) \leq 1.7\ L^* + 2.$$

L^* represents the number of bins in the optimal packing for list L [26]. Yao has shown that for any on-line algorithm the worst case performance can be no better than 3/2 L^* [21].

Off-line algorithms, however, perform better, since the list of items presented to the algorithm can be conditioned. Typically they are sorted into decreasing sequence. The best-fit decreasing rule, BFD, consists of sorting the list of items into decreasing order, and then applying the best-fit rule. Johnson [26] has shown that the BFD algorithm has a worst case performance of

$$(11/9)\ L^* \leq BFD(L) \leq (11/4)\ L^* + 4.$$

At this point, we will examine the best-fit rule to see why its performance is improved when the list is presented in decreasing order. In order to do this, it is useful to think of the best-fit rule as partitioning the set of nonempty bins into two classes. The first class consists of

those bins whose gaps are not smaller than the size of the item to be added. This group, G_1, represents the domain of the traditional best-fit rule. From this group, the bin which will have the smallest gap, once the new item is added is chosen.

The second group, G_2, consists of those bins whose gap is too small to receive the new item. These bins are not candidates to receive the new item when using the best-fit rule. Thus, as group G_2 grows large, the number of choices available to the best-fit rule grows small.

Now it is easy to see why the best-fit decreasing algorithm works better. The larger the a_i, the fewer the bins from which the algorithm can choose the best-fit. Since the larger items are at the beginning of the list, at which time the maximum cardinality of the set G_1 is small, the impact that an item will have on the size of G_1 is small. On the other hand, if a large item is placed in the middle of the list, it will mean that many of the partially filled bins will not have enough room to hold it. The best-fit algorithm will, therefore, have fewer places to store the item than would otherwise be the case if a_i was smaller. The BFD algorithm is successful because the presorting allows more bins to be examined as potential candidates to receive a new item, than if the list of items was unsorted. This suggests, that if the performance of a best-fit rule is to be increased, that a way to expand the

number of bins from which a choice is made, must be found.

5.2 The Best-Fit With Replacement Rule

In order to increase the packing performance of the best-fit rule, the search space (number of bins which are considered) will be expanded. The following rule causes both sets G_1 and G_2 to be searched when an item, a_i, is presented for packing.

1. For those bins in G_1, find the bin, B_i, whose gap will be minimum with the addition of a_i. If no $B_i \in G_1$ satisfies this condition, return the null set.

2. For those bins in G_2 with a gap > 0, find a bin, B_j, whose gap will be minimum when one object stored within it is replaced by a_i. If no $B_j \in G_2$ satisfies this condition, then return the null set.

3. Select from the two sets generated above, the non-null set that contains the bin with the smallest projected gap. In case of ties, select the bin with the smallest index. If it is from G_1, then add a_i to the contents of the bin. If it is

from G_2, replace the item in the bin with a_i so that the gap is minimized. When both candidate sets are null, place a_i in the next unused bin.

4. If an item has been replaced by a_i in Step 3, repeat Steps 1 through 3 until no new replacement occurs and then process a_{i+1}.

This new rule is called best-fit with replacement (BFR). The replacement option expands the set of bins which are considered to all bins, except those which are empty, or whose gaps equal 0. To show how this rule works, the following list will be used:

6 occurences of .129 followed by 6 occurences of .352 followed by 6 occurences of .519.

In Figure 5.1, the BFR packing of this list is compared to the optimal packing and that derived from the BF rule. It takes 10 bins using the BF rule, but only 8 bins with the BFR approach. The optimal packing uses 6 bins. The difference between the optimal packing and the BFR packing is 2 bins, or 1/3 of the optimal packing. The BF rule uses 4 extra bins equal to 2/3 the optimal packing, or double the extra bins.

The list which was used in Figure 5.1, presents a worst case situation for the BF rule [26], because the physical ordering precludes the placement of a small, medium, and

BIN 1	BIN 2	BIN 3	BIN 4	BIN 5	BIN 6	BIN 7	BIN 8	BIN 9	BIN 10
.129	.352	.352	.352	.519	.519	.519	.519	.519	.519
.129	.352	.352	.352						
.129									
.129									
.129									
.129									

A. BF Packing

Bin 1	Bin 2	BIN 3	BIN 4	BIN 5	BIN 6	BIN 7	BIN 8
.352	.352	.519	.519	.519	.519	.519	.519
.129	.352	.352	.352	.352			
.129	.129						
.129							
.129							
.129							

B. BFR Packing

Bin 1	Bin 2	Bin 3	Bin 4	Bin 5	Bin 6
.519	.519	.519	.519	.519	.519
.352	.352	.352	.352	.352	.352
.129	.129	.129	.129	.129	.129

C. Optimal and BFD packing.

Figure 5.1 A comparison of the BF, BFR, BFD, and Optimal Packings for the example list.

large item in each bin. Such a placement is necessary to obtain the optimal packing. On the other hand, the BFR rule, through the use of replacement, is able to make enough reorderings to significantly improve the performance. Later, it will be shown that the BFR rule will repeat its good performance when presented with longer lists.

5.3 Implementation of the BFR Rule as a Parallel Algorithm

Now that the BFR rule is established, an algorithm which uses the rule will be developed. Such an algorithm could be either sequential or parallel. Note, that the improved performance is gained by increasing the number of bins, which are considered each time an item is added. This results in a direct increase in the number of computations which will be required. Although a sequential algorithm may be sufficient for small lists, for larger lists, the number of computations would be prohibitively large. Thus a parallel implementation for our algorithm is desirable. The next question to ask then is, what aspects of the problem can be done in parallel?

The bulk of the new computations are involved in evaluating the capacity of each bin to hold a new item. Therefore, this seems to be the appropriate place to consider parallel acitivities. Since each bin may have a different number of items in it, and that the bins are

divided into two groups, G_1 and G_2, suggests that asynchronous parallel processing may be appropriate. Additionally, the item to be packed must be distributed to the processors, and the results compared to select the proper bin. This suggests a master-slave relationship between the processors. These requirements are well supported by a multiprocessor system. Therefore, a parallel algorithm for the BFR rule on a multiprocessor system will now be presented. The algorithm will be refered to as the MBFR (multiprocessor best-fit with replacement) algorithm.

In the parallel version presented here, a process is associated with each active bin. A bin is active if it is neither empty nor has a gap of zero, when an item, a_i, is presented on-line to be packed. It is given to a master process which broadcasts the item's value to each active process. For each active bin, the associated process computes the new gap that the bin will have if the item is added to it. If the bin has room for the item, it returns a value to the master equal to the new gap. If on the other hand, the item is larger than the current gap, the process scans the list of items in the bin. The items are assumed to be in decreasing order. The scan procedes from largest to smallest until an item, x_j, is found, such that a_i is greater than x_j and a_i is less than or equal to the current gap plus x_j; or the size of x_j plus the gap is less than a_i. In the former instance, a value is returned equal to the

projected gap if x_j were to be replaced by the new item, along with the x_j to be replaced. Otherwise, a null value would be sent, representing that no replacement is possible.

The master then collects the projected gaps or bids from each of the active processes. It then selects the bin with the smallest projected non-null gap to receive the new value. If the bin is from the G_1 set, a signal is sent to the associated process to insert the new item into the ordered list, and reduce the gap for the bin by the value of the item. When the bin is from the G_2 set, the associated process is told to remove the item to be replaced and to insert the new item. The gap is also updated. Next the master broadcasts the replaced item to all the active bins, and the cycle continues until no replacement is necessary. When this occurs, the master reads the next item from the input list. The algorithm will always terminate, because a replaced value may be only smaller than the item to be added. Thus if a replaced value also causes a replacement when it is re-added, eventually it will be too small to replace other items.

A version of the algorithm is given in Appendix A. Simulations of the algorithm have been run on various length lists. The results of these simulations will be discussed in the next section.

5.4 The MBFR Algorithm Simulation

A simulation model of the MBFR algorithm was prepared to investigate the algorithm's performance. The model, in sequential Pascal, simulates the parallel algorithm, and assumes an arbitrary number of processors. Three hundred trials were run using the simulation model. Thirty runs were made for each of ten list lengths, ranging from 18 to 180 items. We now discuss the structure of these lists.

Graham [27] analyzed the performance of the first-fit and best-fit bin packing algorithms. He suggests a class of lists for which these algorithms perform poorly. The example presented previously uses a list belonging to this class. The lists are defined as:

Let n be divisible by 18, and let δ satisfy $0 < \delta \leq 1/84$. Define a list $L = (a_1, a_2, \ldots, a_n)$ by

$$\begin{aligned} &1/6 - 2\delta \quad \text{for} \quad 1 \leq i \leq n/3 \\ &1/3 + \delta \quad \text{for} \quad n/3 < i \leq 2n/3 \\ &1/2 + \delta \quad \text{for} \quad 2n/3 < i \leq n \end{aligned}$$

The optimal packing for such a list is n/3.

For each set of trials, a list of appropriate length (18 to 180) was generated, and then 30 random permutations

of each list were used to test the algorithm.

The results are shown in Tables 5.1, 5.2, and Figures 5.2 through 5.5. Table 5.1 gives the distribution of the number of bins obtained for each set of list lengths. Table 5.2 shows the optimal packing, the average packing, the ratio of the average packing to the optimal packing, and the ratio of the worst case packing to the optimal packing. Additionally, it gives the average number of reverberations (exchanges) per list, the probability that a new item causes a reverberation, and the ratio of the worst case excess bins to the optimal number of bins. Figure 5.2 gives the curve for the average packing relative to the optimal, as a function of list length. To facilitate comparisons, it also shows the performance bounds of 17/10 and 11/9 discussed earlier. Figure 5.3 does the same for the worst case packing. Figure 5.4 shows the ratio of the maximum number of excess bins to the optimal, again as a function of the list length. Finally, Figure 5.5 shows the relationships between the average and maximum number of processors which were required, to the average and worst case packings.

Several aspects of the results stand out. From Figure 5.3, it can be seen that as the number of items in the lists increase, the worst case performance tends to 11/9, and even for short lists, the worst case performance is considerably better than 17/10. Figure 5.2 shows that the average performance is relatively stable, and that it is

Table 5.1 Packings for each simulation list length, for each of the 30 random permutations.

List Lengths	18		36		54		72		90		108		126		144		162		180	
	6*	(15)	12*	(19)	18*	(14)	24*	(11)	30*	(2)	36*	(18)	42*	(8)	48*	(15)	54*	(12)	60*	(11)
	7	(12)	14	(3)	19	(1)	25	(2)	31	(1)	41	(2)	46	(1)	50	(2)	56	(1)	64	(2)
	8	(3)	15	(8)	20	(1)	26	(1)	32	(3)	42	(8)	48	(1)	51	(1)	62	(2)	66	(3)
					21	(3)	28	(2)	33	(3)	43	(1)	49	(3)	53	(1)	63	(1)	67	(2)
					22	(5)	29	(7)	34	(2)	45	(1)	50	(6)	54	(1)	64	(3)	68	(1)
					23	(4)	30	(7)	35	(1)			51	(4)	55	(1)	65	(9)	69	(2)
					24	(2)			36	(9)			52	(7)	56	(1)	66	(2)	70	(3)
									37	(7)					57	(2)			71	(3)
									38	(1)					58	(1)			72	(2)
									39	(1)					59	(5)			73	(1)

* denotes the optimal packing for each list length.

In each column, the leftmost number gives the number of bins used while the parenthesized number gives the number of times that this occured.

Table 5.2 A summary of the performance of the MBFR algorithm for each list length (L).

List Length (L)	18	36	54	72	90	108	126	144	162	180
Optimal Packing	6	12	18	24	30	36	42	48	54	60
Average Packing	6.6	13	20.13	26.96	34.9	38.47	48.17	51.87	60	65.53
Avg $\frac{MBFR}{L^*}$	1.1	1.08	1.12	1.12	1.16	1.07	1.15	1.08	1.11	1.09
Worst Case $\frac{MBFR}{L^*}$	1.33	1.25	1.33	1.25	1.30	1.25	1.24	1.23	1.22	1.22
Reverber-ations	2.66	6.03	8.43	15.23	20.2	19.6	24.07	32.6	32.03	41.4
Probability item causes reverber-ation	.15	.17	.16	.21	.22	.18	.19	.23	.20	.23
$\frac{MBFR(WC)-L^*}{L^*}$	.33	.25	.33	.25	.3	.25	.24	.23	.22	.22

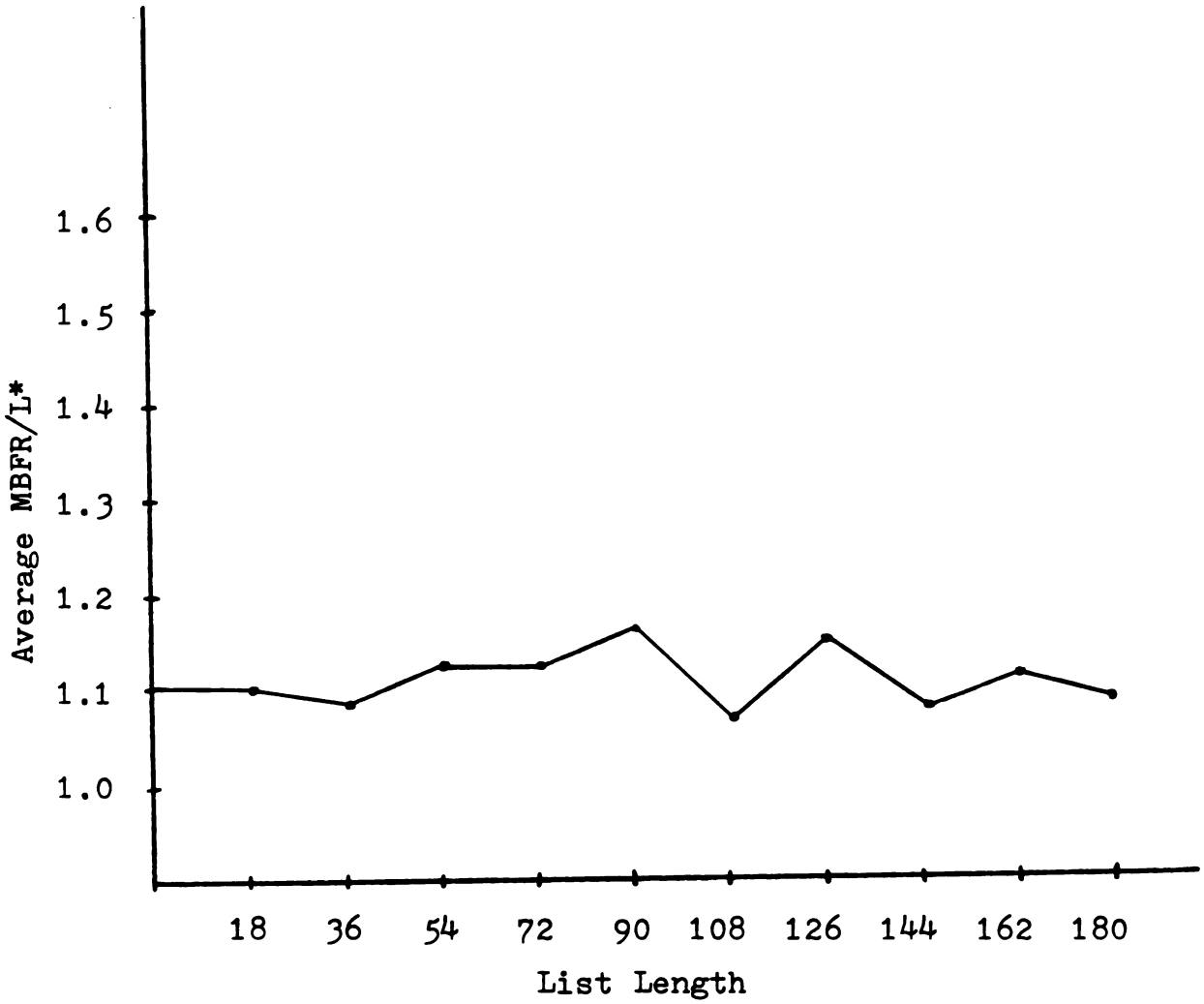


Figure 5.2 Plot of the average packing as a ratio of the optimal packing for the MBFR simulation.

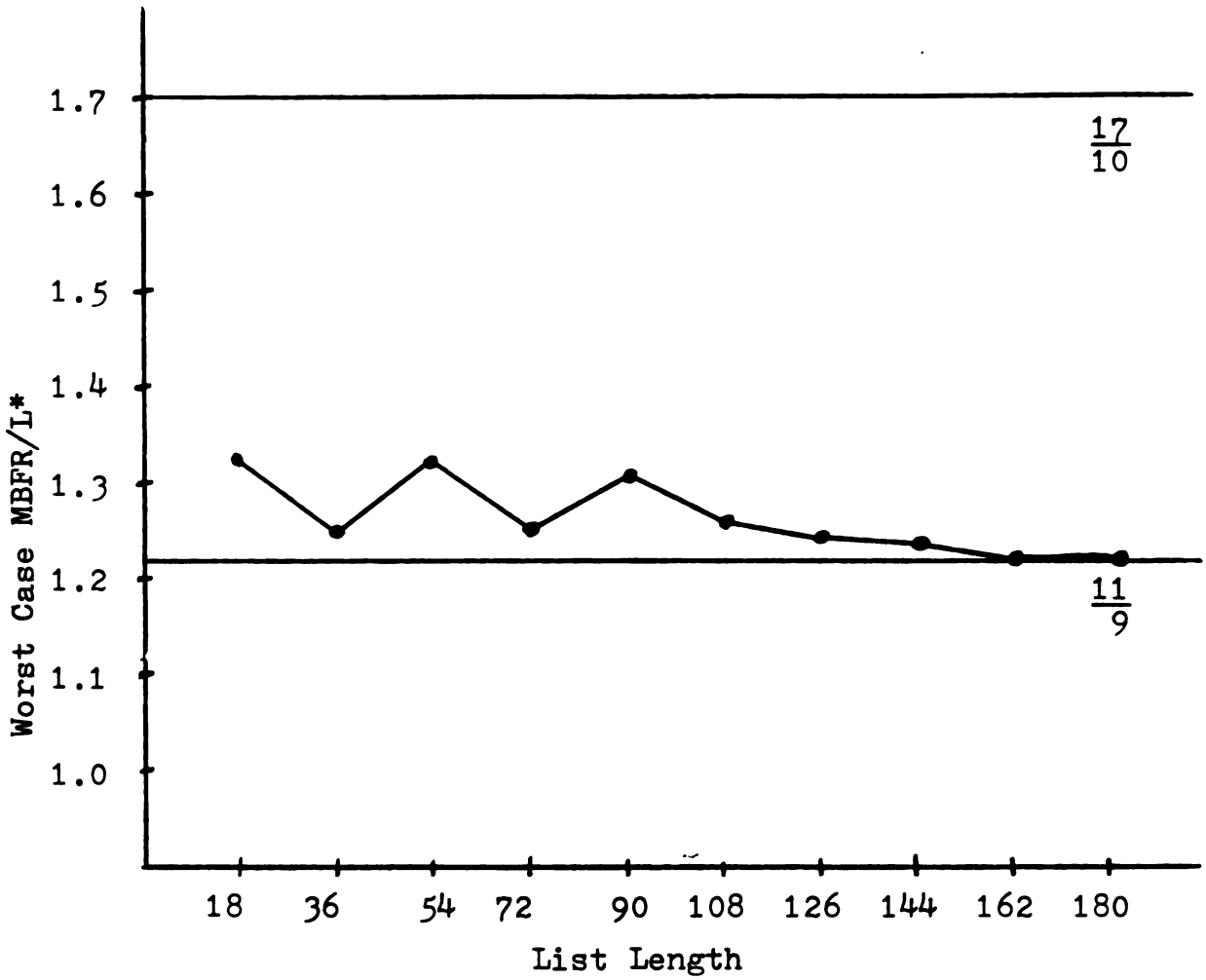


Figure 5.3 Plot of the worst case packing as a ratio of the optimal packing for the MBFR simulation.

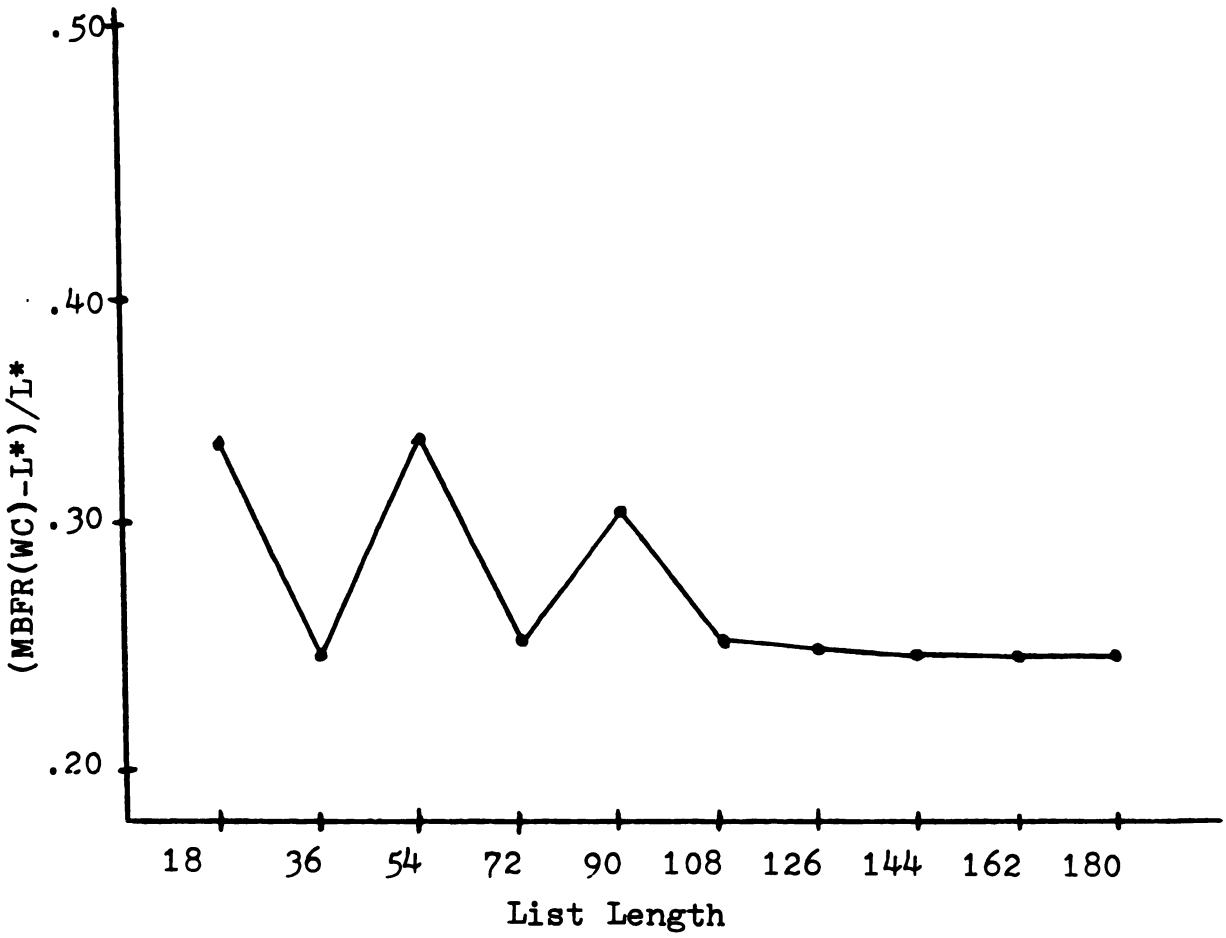


Figure 5.4 Plot of the maximum number of excess bins as a ratio to the optimal packing for the MBFR simulation.

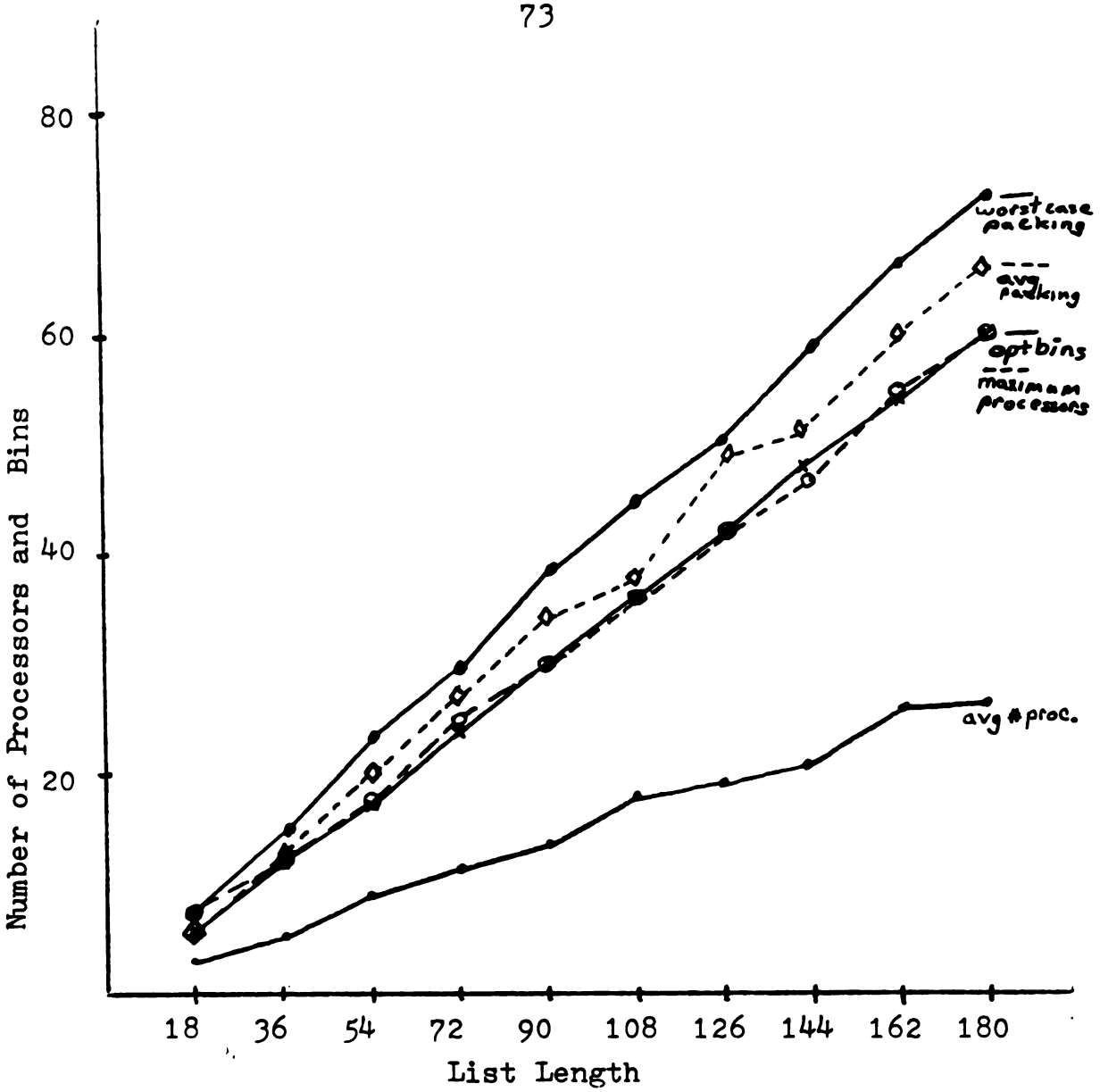


Figure 5.5 Worst case, average, and optimal packings with the maximum and average number of processors required by the algorithm for the MBFR simulation.

considerably better than the worst case; furthermore, by examining Table 5.1, it can be observed that the optimal packing is obtained 42% of the time. Table 5.2 and Figure 5.4 also show the deviation of the worst case from the optimal as a ratio of the optimal. It is interesting to note that the algorithm always performs better than 1/3 and levels off at better than 1/4 for the longer lists.

It should be noted that these results were obtained with the algorithm only having to perform a few reverberations. For example, on the average, only 2.66 items in a list of 18 items caused a reverberation. The maximum number of cascading reverberations for any of the lists was only two. Thus, selecting from G_2 only a limited number of times, still improves the results significantly over that for the best-fit algorithm.

Finally, the results show the number of processors which are required. Remember that the simulation assumed that one processor would be assigned to each active bin. In the worst case, Figure 5.5 shows, that the number of processors seems to be bounded by the optimal packing, not the worst case, as might be expected. The number of processors also appear to be significantly less than the average packing as well. If we look at the average number of processors used, we see that the average is considerably less than the optimal, average, or worst case packings. This shows promise for implementing the algorithm on a

multiprocessor system with a limited number of processors. In such a case, each processor would be assigned several active bins. But as can be seen from Figure 5.5, even for large lists, each processor would still only have to handle a limited number of bins.

5.5 The Computational Complexity of the MBFR Algorithm

This section will discuss the computational complexity of the MBFR algorithm. The multiprocessor algorithm is based on the best-fit with replacement rule. This rule is an extension of the best-fit rule, which was discussed earlier. Therefore, the computational complexity of the best-fit algorithm will be discussed first, in order to give a basis of comparison for the MBFR algorithm. In order to determine the computational complexity of the BF algorithm, we will focus on the computations which test a bin to see if a new item is less than or equal to the current gap. Since this testing process is based on the list length and dominates the computations, the computational complexity will be expressed in terms of the total bins tested for packing a list of n items.

A worst case scenario for the BF algorithm is to attempt to pack a list of items whose sizes are all the same, greater than .5, and less than 1.0. When such a list is packed, only one item can be placed into a bin, and no

bins can ever be completely filled. Thus, when an item a_i, is presented to be packed, i-1 bins must be tested to see if the item will fit. Thus, for a list of n items, $\sum_{i=1}^{n}$ (i-1) bins are tested. This results in a computational complexity of $O(n^2)$.

If the same type list is presented to the MBFR algorithm, the number of bins which are tested will be the same, $\sum_{i=1}^{n}$ (i-1). Although the i-1 bins are tested simultaneously, the master process must still compare i-1 bids each time an item, a_i, is presented to be packed. Note that for the MBFR algorithm, the time which is required to test the individual bins is bounded by a constant, since the bins are tested simultaneously each by a seperate processor. Therefore, the computations that dominate are those of the master process which tests each of the bids generated by the slave processors. Thus, for the MBFR algorithm, the computational complexity is based on the number of bids that the master process must test to pack a list of n items.

The type of list which elicits a worst case performance for the BF algorithm, however does not produce a worst case performance for the MBFR algorithm. The MBFR algorithm will show its worst performance, if two conditions are met. The first condition is that each time an item is added, it must cause an item to be replaced. Also, for each item which is replaced, when it is repacked, it must cause a replacement as well, unless it is the smallest item in the list. The

second condition is that only one item may fit in a bin, but the item may not fill the bin. This condition causes i-1 bins to be tested, if item a_i is presented to be packed. An example of such a list is the following: .6, .7, .8, and .9. That is, all items are greater than .5 and less than 1.0; no items have the same value; and the list is in ascending order. If the first two items are packed, bin 1 will contain .7 and bin 2 will contain .6. When .8 is presented to be packed, 2 bids are generated. .8 will be placed into the first bin and displace item .7. Item .7 is then repacked. Again 2 bids are generated. This time .7 is placed into bin 2 and it displaces .6, which must then be repacked. Thus, if item a_i is presented to be packed, i-1 bids are generated, and the item is packed by displacing the item in the first bin. That item is then repacked, and it will displace the item in the second bin. This process continues until all items have been shifted one bin. This results in the master process having to consider $\sum_{i=1}^{n} (i-1)(i)$ bids. Thus, the MBFR algorithm has a computational complexity of $O(n^3)$.

Therefore, we can see that our simulation suggests that the MBFR algorithm has a better average performance than the BF algorithm. However, in the worst case, it does not perform as well. This is not a major problem for the MBFR algorithm, however, since the worst case for the MBFR algorithm only arises for a list of items with very

restrictive constraints. Also note, that since the list must be ordered for the worst case to occur, the packing of such a list would be best handled by an off-line algorithm.

5.6 Summary

This chapter presented a second sample algorithm to use in our experiments, a multiprocessor bin packing algorithm. Like the ROOTF algorithm introduced earlier, it has a single localized parallel component. However, the ROOTF algorithm has a small parallel component, while the parallel component of the MBFR algorithm is much larger. Thus, we have examples of two algorithms with single, localized parallel components. One has a small parallel component and one has a large parallel component.

A simulation was used to show that the on-line MBFR algorithm has a packing performance comparable to the off-line best-fit decreasing (BFD) algorithm. Although further analysis of the MBFR algorithm would be interesting, for our purposes it is not needed, since we are primarily interested in the structural properties of the algorithm, not its behavior. The ROOTF and MBFR algorithms will be used in Chapter seven, to examine the role of spatial, temporal, and functional information in the design process for these two algorithms. The next chapter will present a model of the algorithm design process. A simulation based

on that model will be used in Chapter seven to conduct the experiments in parallel algorithm design.

CHAPTER 6

The Algorithm Design Model

The previous chapters showed that parallel algorithms may contain additional complexity beyond that found in their sequential counterparts. Some of that complexity resided in the parallel component, and some of that was the result of additions to the sequential component. The size of this added complexity may impact the success of making different types of design decisions. The question which is addressed by this research is; in algorithms with a single parallel component, how does the granularity of the parallel component impact the performance of design methodologies.

In this chapter, we will develope a model of the algorithm design process which will allow us to study this question. The design process will be modeled using the graph language representation of an algorithm, introduced in Chapter three. Section 6.1 will describe the basic design

unit in our model, the design step. A design step, in our model, consists of two phases; a placement process, and an elaboration process. Section 6.2 will discuss the placement process, and Section 6.3 will describe the elaboration process. A procedural description of the model will be presented in Section 6.4. Finally, the chapter will conclude by describing a simulation model based on the abstract model of Sections 6.1 through 6.4. The simulation model will be used in Chapter seven, to conduct experiments in parallel algorithm design.

6.1 A Design Step

In Chapter one, we characterized the design process as producing a sequence of partial designs. In Chapter three, an example of two successive partial designs (Figures 3.3 and 3.4) was used to show that we could use our software metrics to quantify the change in the information content from one partial design to the next. The change in the conceptual volume between the two partial designs was a result of making modifications to the structure of the first partial design. Two basic types of decisions that underlie these modifications can be observed. One process determines the positions in the current partial design where modifications are to take place, i.e., where to make additions, deletions, or alterations. The other process

determines what modifications will be made at these positions. This process includes decisions about the addition, deletion, or alteration of portions of the current partial design. It also determines the size and functionality of the modules which contain those portions of the current partial design that are to be modified.

Each of the above processes have been extensively discussed in the literature [4-10,28,32]. For example, the placement of a set of modifications in a partial design dominates many discussions of design methodologies. These decisions are considered so important that design strategies have often been named after this selection process, i.e., top-down and bottom-up design [5]. Some of these methodologies are based on a hierarchical design approach [34]. That is, the problem is described at several levels of abstraction. At the first level, the algorithm is described in general terms and at succeeding levels, an increasing amount of detail is added. Top-down approaches suggest that the designer create an algorithm in a hierarchical fashion. At one level a main function is created and at succeeding levels, subfunctions which fill in the details of the algorithm, are added [34]. Top-down methodologies dictate the levels or placement of the functions in the algorithm which the designer may deal with during a particular design step. Bottom-up design, likewise describes an algorithm in several levels of abstraction.

However, bottom-up design suggests that the designer start at the lowest levels of detail and group low-level actions together into successively more generalized functions [32]. Although these two methodologies differ, they both aid in the placement of the next set of modifications to the current partial design. In this research, we will call this process the placement process.

The second part of a design step, the process of deciding what modifications are made as part of a design step, has also achieved much attention [5,28,32]. Much of this discussion has centered on dividing an algorithm into manageable size units (modules). Some of these discussions involve; module independence, interfaces between modules (coupling), module cohesion, how modules should cooperate, as well as data hiding. For example, in discussions on cohesion, Wirth [8,32] suggests that certain instruction sequences which perform simple basic actions should be grouped into primitive modules or "action clusters". Others [4-10], offer suggestions such as limiting module descriptions to one physical printer page, or to collect in a module a set of related primitive functions, i.e., basic string handling functions. All of these and similar suggestions serve to decompose the problem, and therefore, the design into components. In this research the process of performing this decomposition and deciding what objects should be added, deleted, or altered as a result of a design

step, will be called the elaboration process.

Recall, that the development of an algorithm can be represented by a sequence of partial designs. The placement and elaboration processes are used to transform one partial design into the next. The set of decisions and resultant actions which lead from one partial design to the next, as a result of the placement and elaboration processes, is called a design step. A design methodology is a set of rules which are used to aid the designer in making the placement and elaboration decisions. Thus, a design methodology can be represented as a set of guidelines which a designer uses in order to make the decisions which lead from one partial design to the next.

For convenience, this research will limit the placement decision for a design step to one location in a partial design. This restriction will not limit the generality of our model. A design step which would have allowed independent modifications to several locations can still be modeled as a sequence of design steps, one per location, for each location where a modification was to be made.

The next section will discuss how our model of the algorithm design process represents the placement process. It will be followed, in Section 6.3, by a similar discussion of the elaboration process.

6.2 The Placement Process

The placement process is the set of decisions which determine where to make the next modification to the current partial design. There are several ways in which this component can be selected. These criteria are based on selecting a component from the set of algorithm components which are not in the current partial design [5-10,28,32].

1. Pick a component which is physically adjacent to a particular component in the current partial design, i.e., in the same module.

2. Pick a component which is concurrent to a component in the current partial design.

3. Pick a component which provides an input to a component in the current partial design.

4. Pick a component which receives an output from a component in the current partial design.

5. Apply criteria 1-4, but only with regard to the most recently developed component, i.e., the relationship

must connect the new component to the component which was dealt with by the last design step.

6. Pick a component which is at the same level of abstraction as the last component, i.e., if the last design step added part of the main procedure, the next component should be part of the main procedure as well.

7. Pick a component which is at an adjacent level of abstraction as the last component, i.e., add a subprocedure to the procedure which was dealt with by the last design step.

8. Pick a component with a similar functionality as the last component, i.e., if the last component did a particular string handling function, the next component should deal with string handling as well.

These criteria are just a few of those used by software designers. However, they serve to illustrate the types of information a designer uses to make the placement decision. The placement decision can be described relative to the current partial design. That is, the decision somehow exploits a relationship between the new component and those components already in the current partial design. These

placement relationships may be based on spatial relationships, criteria 1, 6, and 7, or on informational relationships, criteria 2-4, and 8. In otherwords, these relationships connect components because they are physically close in an algorithm's representation, or they share common functions and/or data items.

The temporal order in which components are added to a design can be used to describe placement relationships. For example, placement relationships may connect the next component to the component which was added or modified by the last design step. However, placement relationships may also connect the next component to a component which was dealt with by the second to last or other previous design step. Also, some of the relationships used by a placement decision may be based on the last design step, while other relationships, which are based on other previous design steps, may be used by the same placement decision. Thus, we can see that the information, which is used by a design step, may have a temporal order to it.

In order to model the placement process, we must identify a set of placement selection criteria to use. Based on the information which is available in our graph language model, the following relationships can be easily identified. They are based on both spatial and informational relationships between the structures of the graph language. These relationships are the following.

1. Components which are concurrent to each other.

2. Components which are in the same module.

3. Components which provide inputs to other components.

4. Components which receive outputs from other components.

The above criteria will be used to select the placement of the next component, based on a relationship with the current partial design. Some of the relationships are based on the physical positions of the objects in the graph language representation of an algorithm (spatial locality). Other of these relationships are based on a common functionality or the direct communication of data between the graph language structures of an algorithm (informational locality). These relationships also have a temporal order to them. If we use the above relationships to connect a new component to the component which was dealt with by the last design step, we say that it is a first order placement relationship. We call this a first order relationship, since the new component is related to a component which was treated by a decision which is one design step away. If we use the relationships to connect the next component to the component which was added

or modified by the second to last design step, we say the placement relationship is second order. A relationship based on a component added or modified by a specific previous design step will be called an nth order placement relationship. It is possible that a design step uses multiple placement relationships. If this is the case, we will assign the temporal order of the highest order placement relationship, which is used by that design step, to the decision.

In order to model placement relationships, we will partition them into two classes. Relationships based on the above four criteria, which are of the first order, will comprise the first class. Relationships which are second or higher order form the second class. Since the relationships in the first class are based on spatial or informational localities, and are temporally local, i.e., are based on the last design step, we will call a placement based on these relationships a localized placement. Placements which are based on the relationships in the second class will be called non-localized placements.

Non-localized placements are those based upon second or higher order temporal relationships. To model this broad spectrum of possibilities would be quite complex. Therefore, non-localized placements will be modeled probabilistically. This will allow us to represent all possible criteria for placements, without having to include

the many combinations of relationships that some of these criteria would require. Thus, if the decision is made to use a non-localized placement, one of the components not in the current partial design will be randomly selected to be treated by the next design step.

Localized placements in our model are dependent on the last component which was dealt with by the previous design step. They will be modeled by one of the four locality relationships described above. Later in Section 6.3, it will be seen that our design model allows for the possibility of errors during the design process. If an error occurs, control structures in the last component may not be complete. To allow these control structures to be completed, a localized placement may also choose the last component, if the previous design step did not complete it. The next design step then has the possibility of completing it.

The process of selecting the next component consists of looking at the last component which was added or modified in the current partial design. Then it is determined whether any of the components which are not in the current partial design are connected to the last component, based on one of these localized relationships. If so, one of those components is selected. There may be several possible candidates for selection, and modeling the decisions to choose between them would be complex. Therefore, the

selection from the list of possible candidates will also be probabilistic. Thus, a localized placement probabilistically chooses the next component from a constrained subset of the remaining components, and a non-localized placement probabilistically chooses from the set of all remaining components.

Recall from the first part of Section 6.2, that several placement criteria were introduced. Those criteria are either modeled directly by the localized placements, or indirectly by non-localized placements. Since we can combine the localized placements in any order, or include any component using a non-localized placement, all possible first or any degree order placement criteria can be represented by our model.

We will now discuss the process by which the decision to use a localized or non-localized placement is made. Many factors influence the progression a designer uses in creating an algorithm [7-9]. These include:

1. The designer's perspective of the problem.

2. Simularities between the problem and those encountered previously by the designer.

3. Available mathematical models of the problem.

4. The language being used to design the algorithm.

5. The design methodology used by the designer, i.e., top-down, bottom-up, stepwise refinement, etc.

These factors are highly variable and differ considerably between designers. Also, we are not interested in how they are determined, but only the relative frequency of using the different criteria. Therefore, the decision to use a localized or non-localized placement will be modeled by a frequency distribution. Likewise, if the choice is made to use a localized placement, the selection of which of the five relationships to use will also be modeled by a frequency distribution. These frequencies will be represented in our model by a set of probabilities. They are the following:

P_R Probability of using a non-localized placement.

P_I Probability of using an input placement relationship.

P_O Probability of using an output placement relationship.

P_C Probability of using a concurrent placement relationship.

P_M Probability of using a module relationship.

P_S Probability of using the same component.

We call these probabilities the design methodology probabilities.

By representing the placement decision as a set of probabilities, we will be able to examine how the relative frequency of use for each of the placement criteria affects the rate at which information is added to an algorithm's design. This will be done by examining the change in conceptual volume between successive partial designs. By varying the design methodology probabilities, different design methodologies can be examined.

The next section will discuss how a component is processed, once its placement has been determined.

6.3 The Elaboration Process

Previously in this chapter, we have been using the term component to refer to those algorithm graph elements which are added or modified in the current partial design as a

result of a design step. We will now consider a more complete description of those graph elements.

In order to design an algorithm, the problem which is to be solved is often decomposed into managable size units. We will call these units design modules. In the literature [5,28,32], several criteria have been suggested for decomposing the design of an algorithm into design modules.

1. Divide the problem into major functions. Let each function be represented by a design module.

2. Design modules should be independent. The meaning of a design module should be independent of the context of the algorithm.

3. Design modules should be self contained, i.e., one should be able to replace a design module in an algorithm by a different one with the same function, and the algorithm should still be correct.

4. The size of a design module should be limited. Its representation may be limited to a number of printer lines.

5. Group primitive actions which perform a basic function together into a design module.

Most of these criteria are based on a functional subdivision of an algorithm, and thus are highly dependent on a designer's perspective of the problem. Two designers given a particular problem, will often arrive at different modular decompositions.

The modular decomposition of an algorithm can also be hierarchical [5,28]. Thus, a designer may decompose an algorithm into a set of high level modules, which are then further decomposed into a set of lower leveled modules. Again, the choice of how to select these modules is variable, and different designers will arrive at different modular decompositions. This variability presents a problem for us in our attempt to model the elaboration process. Therefore, the modular decomposition of the algorithms used by the model will be prespecified.

We will model the design process for an algorithm through the process of recreating a graph language representation of it. We will assume in our model, that algorithms are designed in a modular fashion. Recall, that the graph language is based on a functional representation of an algorithm. The objects in a control structure, or which are in control structures that are nested in the first structure, represent a functional unit. If we assume that the algorithm was created in a modular fashion, it is reasonable to assume that these functional units could be designed as a unit. Thus, these functional units can

represent a possible decomposition of the algorithm into design modules. These design modules are not necessarily unique. One designer may group one combination of functional units together into a design module, and another designer may use a different grouping, even though the underlying functional units are the same. Thus, in order to control the experiments we will prespecify the design modules for an algorithm. These modules will represent the module placement relationship and will remain fixed for all experiments using a given algorithm.

Design modules represent a high level decomposition of a problem. However, the information which is represented by a module may be too much to consider during one design step. Therefore, it is often necessary to further divide a design module into units which can be treated by a single design step.

The portion of an algorithm which can be dealt with at any moment is limited. Halstead [16] and others [29] suggest that humans are limited in the number of mental discriminations per second which they can make. They suggest that the design units programmers deal with are not individual elements, but groups of objects. These groups have been called chunks [30], and represent those discriminations which can be made as a unit. In this research, a chunk is assumed to correspond to such a group of objects, but it also includes the design decisions which

relate those objects together. If an attempt is made to deal with too much information in one chunk, errors often result. Additionally, sometimes a designer makes an error, even when a chunk deals with limited information. Therefore, our model of the algorithm design process will consider the possibility of an error each time an object is considered for inclusion in a chunk.

In our model, a chunk represents the objects which are added or modified in the partial design, as the result of a design step. A chunk is what the previous sections in Chapter six have refered to as a component. Chunks are the result of decomposing a design module into units which can be treated by a single design step. Although, our model will prespecify the modular decomposition of an algorithm, chunks will be determined at the time the algorithm is designed. Chunks are determined as follows. First, a chunk must be contained within a design module. A control structure is used to identify a chunk. Objects which are in that control structure, or which are in control structures nested in the first structure are candidates for a chunk. Note, these objects also represent a functional division of the algorithm. A chunk differs from a design module, in that the size of a chunk is tightly controlled. A probabilistic limit will be placed on the number of objects which may form a chunk. Thus, the functional decomposition, which can be seen in the graph language representation of an

algorithm is used in two ways. First, at a high level we will group these functional units into design modules and feed a description of this grouping into our model. Second, the functional units will allow the model to form basic units which are small enough to be treated by a design step, yet represent related acitivities. These second units are called chunks.

The basic action of our model is a design step. It involves the determination of the starting point for a chunk by the placement process, and then the subsequent elaboration of that chunk. This process is being modeled by recreating a graph language representation of an algorithm. Therefore, we will place one further restriction on the elaboration process. That is, we will restrict the elaboration process to adding graph language elements to a chunk. Since the target algrithm is predetermined, the only way we can insure that the proper algorithm will be designed is to prespecify the elements in the algorithm and eliminate backtracking in the design process. Thus, the elaboration process consists of identifying the objects in a chunk and adding them to the design, subject to the limitations of the designer making an error. As mentioned earlier, each time an object is considered for inclusion in a chunk, the possibility of an error will be considered. This is done for two reasons. First, it is a means to limit the size of a chunk. Second, all designers make errors. This allows

the impact of these errors to be represented in our model.

6.4 The Procedural Model

The previous sections described an informal model of the algorithm design process and its relationship to current software design theory. We will now specify our model of a design step in procedural terms. Figure 6.1 gives a high level outline of our algorithm design model. The model consists of two parts; an environment and a designer. The environment represents the information which is available to the designer. It includes the methodology probabilities, the graph language representation of the algorithm, the localized relationships between the graph language components, and statistics about the algorithm which is being designed. Most of this information is provided to the model and is predetermined. The designer part of the model represents the process of algorithm design. The placement and elaboration activities are represented by this portion of the model. We will now examine the designer process.

The designer process starts in box a of Figure 6.1. Using the methodology probabilities, it decides whether to make the next placement by using a localized or a non-localized decision. If a non-localized decision is selected, the model enters box d, which randomly selects any chunk which is not in the current partial design. If a

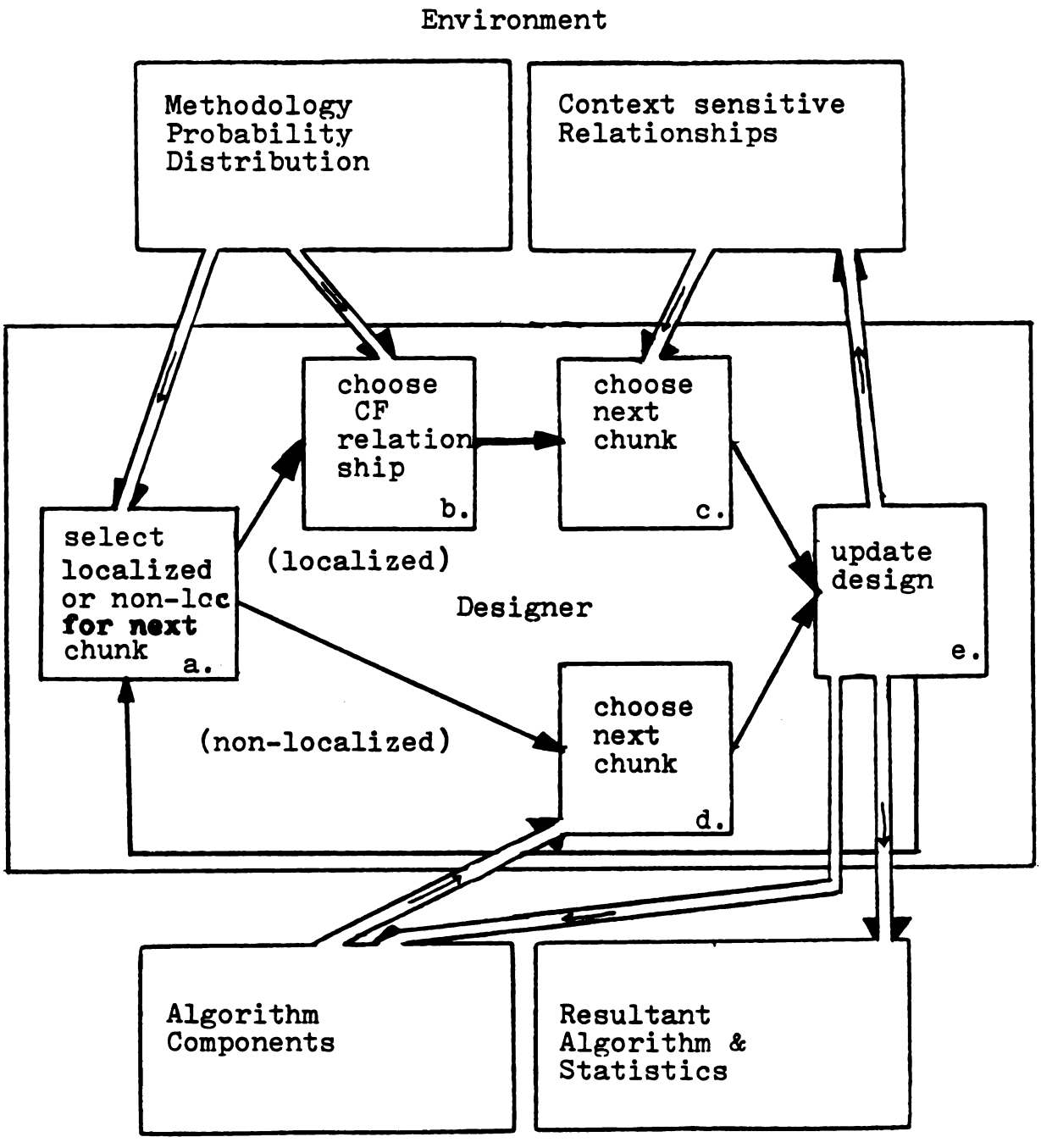


Figure 6.1 Abstract model of the algorithm design process.

localized decision was selected in box a, the model enters box b. Here, a particular localized relationship is chosen, based on the methodology probabilities. Then the model enters box c where a chunk, which is connected to the last elaborated chunk, is randomly chosen, based on the relationship which was selected in box b. Then from either box c or d, the model enters box e.

In box e, the objects which form the chunk are determined. These objects may be input operands, results, operators, conditionals, or other control structures which are nested in the structure that starts the current chunk. The chunk elaboration process consists of selecting these objects from the algorithm's representation in the environment and placing the objects into the current partial design. As the objects are selected, statistics are kept which indicate how many objects have been referenced and how many unique items are in the current partial design. These statistics allow the conceptual volume to be calculated at the end of each design step.

When objects are identified as possible candidates for inclusion in a chunk, the possibility that the designer is making an error is considered. This is modeled by counting the number of objects in the chunk and probabilistically limiting the size of the chunk. This accomadates the notion discussed by Halstead [16] and others [29], that human designers are limited in the number of mental

discriminations that they can make as a unit. If an error is indicated, the object is not added to the cunk, and the elaboration of the chunk is complete. The model then reenters box a and continues in this fashion until all objects have been added to the design.

This completes our discussion of the abstract model of the algorithm design process. We have modeled this process as producing a sequence of partial designs. A design step represents those actions which are used to move from one partial design to its successor. The actions, which make up a design step, were described by two processes; a placement process and an elaboration process. The next section will describe a simulation, which was developed based on this abstract model.

6.5 The Simulation Model of Algorithm Design

The previous sections described the abstract model of the parallel algorithm design process. In this section, a simulation based on the abstract model will be presented. The simulation model was implemented in Pascal on the Cyber 750.

The simulation model consists of two parts. An environment and a designer. The environment represents the information which is used by the model during the design process. It includes a modular description of the target

algorithm, the methodology probabilities, the placement relationships, and the counts which are used to generate the software metrics. The designer represents the process of algorithm design. The designer process embodies the logic that uses the information in the environment to make the placement and elaboration decisions.

6.5.1 The Model's Environment

The environment of the model was implemented as a series of tables and pointers, see Figure 6.2. The graph language algorithm is represented by tables STRUCT, OPERAND, and FUNCT. Table STRUCT describes the structure and state of elaboration for the algorithm. Each graph control structure has an entry in table STRUCT. There are pointers from STRUCT for each graph element to tables OPERAND and FUNCT. A pointer to table FUNCT identifies the control structure's operator, and pointers to table OPERAND identify the control structure's operands, these are the control structure's inputs and results. Additionally, table STRUCT has entries for each graph control structure which indicate the control structure's context, sequential or parallel. There are also flags to indicate if each of the control structure's elements have been added to the design.

Tables OPERAND and FUNCT each have one entry for every operand/operator that are used in the algorithm. Each entry

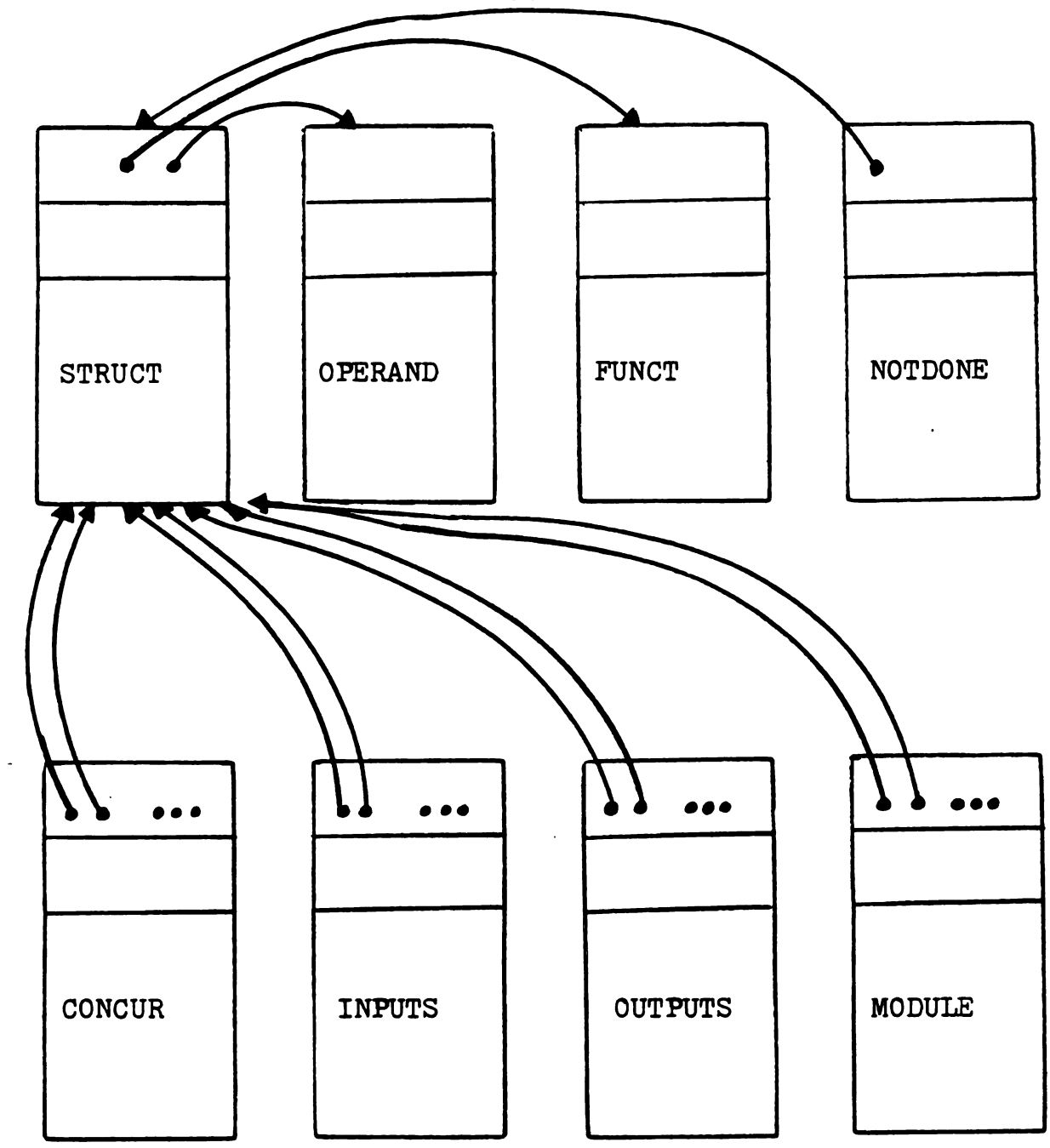


Figure 6.2 Simulation model's representation of the design environment.

consists of two flags. The flags correspond to the context, sequential or parallel, under which references to the operands/operators occur. The flags are set after the first reference to the objects in a particular context are made. This is done in order to achieve a count of unique algorithm elements for the conceptual volume metric.

The first order locality relationships are represented by tables CONCUR, INPUTS, OUTPUTS, and MODULE. CONCUR represents the relationship between graph control structures that are concurrent to each other. INPUTS and OUTPUTS represent the relationships of control structures that provide inputs or receive outputs from other control structures. Finally, MODULE indicates which graph control structures form modules of the algorithm. Each graph control structure which participates in one of the relationships with other graph control structures, has entries in the appropriate tables. An entry consists of a pointer to table STRUCT which identifies a control structure. Additionally, it includes a pointer to table STRUCT for each control structure that is related to the first entry.

Table NOTDONE is used to allow a non-localized placement to choose the next graph control structure. It contains entries for all graph control structures which have not been fully elaborated. An entry consists of a pointer to table STRUCT.

The status of the current design is represented by several variables in the simulation model. POS is a pointer to table STRUCT for the graph control structure of current interest. STEPS indicates how many chunks have been processed. Finally, UNIQUE and TOTELEMENTS hold the element counts which are required for generating the conceptual volume metric.

6.5.2 Design Methodologies In The Simulation Model

A design methodology is represented by a set of frequency distributions. These distributions indicate the relative frequency of use for each of the placement criteria. For convience, these distributions are represented in the model as a cumulative probability distribution, Figure 6.3. The entries are used to select a decision type for choosing the next chunk. The criteria are based on the control structure that identifies the last designed chunk (POS). The basic criteria and their associated codes follow:

S The same control structure should start the next chunk.

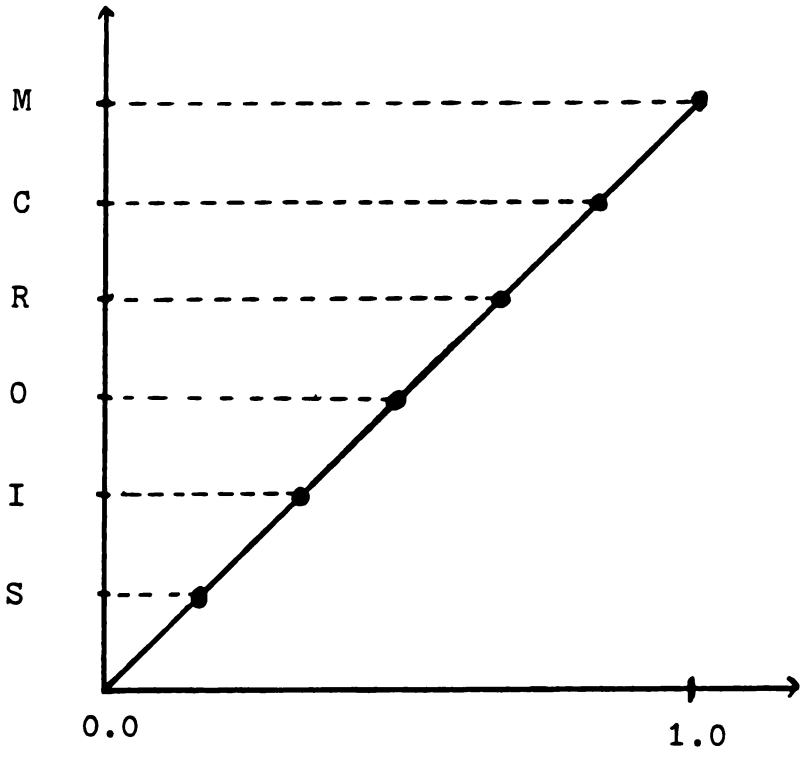


Figure 6.3 Simulation model's representation of a design methodology.

I,O The next chunk should provide an input to, or receive an output from the last chunk.

C The next chunk should be concurrent to the last chunk.

M The next chunk should be in the same module as the last chunk.

R randomly select the next chunk (a non-localized placement).

In the simulation model, table POSITION represents this probability distribution. The table has one entry for each of the accented points in the plot of Figure 6.3. Different design methodologies are modeled by changing the shape of the curve.

6.5.3 The Design Process

This section will discuss the simulation's implementation of the placement and elaboration processes. First, the model is initialized. This is done by reading a graph language based description of the algorithm into table STRUCT. The locality relationship tables CONCUR, INPUTS,

OUTPUTS, and MODULE are provided a description of the first order relationships between the components of the graph language algorithm. From these descriptions, tables NOTDONE, OPERAND, and FUNCT are also initialized. Table POSITION is initialized by reading a cumulative probability distribution. This distribution represents the relative frequency for each of the placement criteria. Finally, since the model is probabilistic, a pseudo-random number generator is required. The seed for the generator is initialized by reading the system clock. This gives a different pseudo-random number sequence for each trial.

The simulation's implementation of the designer process will now be discussed. First, the placement of the next chunk is determined. This is done for all chunks, except the first one. The first chunk always starts with the outer most graph control structure. If we refer to Figure 6.1, we can trace the placement process. The placement process is represented by boxes a through e in Figure 6.1. In box a, a pseudo-random number is generated. This number is used to select the type of decision from the cumulative probability distribution which is in table POSITION. If a non-localized placement is selected, box d is entered. In box d, a chunk is randomly selected from the set of all graph control structures which are not in the current partial design (table NOTDONE). If a localized placement had been selected in box a, the model would have entered box b, instead of box

d. In box b, a first order placement relationship is selected. This is done by generating a pseudo-random number and using it to access table POSITION. Once this is done, box c is entered. In box c, the next chunk is selected by examining the table for the relationship that was selected in box b. If more than one graph control structure is related to the last chunk, one of them is randomly selected. If no structures are related to the last chunk using the appropriate relationship, a different placement decision is tried by returning to box a.

Once the starting control structure for the next chunk has been selected in either box c or d, box e is entered. Box e represents the elaboration process. The objects, which consitute a chunk, are identified from tables STRUCT, OPERAND, and FUNCT. If an object is to be included in a chunk, the entry for the object in table STRUCT is marked. These objects may be results, operators, input operands, conditionals, or nested control structures. Objects are added to a chunk in a predetermined order. First, the need for a chunk is determined by its results. Therefore, the result operands are the first objects which are added to the chunk. After the results are identified, the actions which yield those results are considered. Therefore, the operators are added next. Input operands and conditionals are lower level details, therefore, they are added last. Although all designers may not use this ordering for adding

objects to a chunk, varying this order would not significantly impact the results from our model.

As objects are identified from the tables, the possibility of an error is considered, before an object may be added to the design by marking the tables in the environment. This is done by procedure ERROR. This procedure generates a pseudo-random number and test it against an error distribution, $e^{-x/10}$, where x is the number of objects in the current chunk. This distribution allows us to limit the number of items in a chunk to match the notion discussed by Halstead and others, that the number of discriminations which a designer can make as a unit is limited. The distribution causes the probability of an error to increase exponentially as objects are added to a chunk. It bounds the number of objects at 21. If the pseudo-random number exceeds the value of the distribution, an error is assumed and the current chunk is complete, without including the new object.

As table STRUCT is marked to indicate the object is now in the current partial design, counts of the objects which are referenced and the number of unique objects are accumalated. These counts are used to compute the software metrics. This process of identifying the objects from the graph language description, and marking that description to indicate that the objects are in the current partial design continues until either an error occurs or until no eligible

objects are left to include in the current chunk. Eligible items are those in the control structure which starts the chunk, or those that are in control structures which are nested in the first control structure. Recall, that these objects represent a functional decomposition of the graph language algorithm.

Finally, the elaboration process finishes in box e, by updating the tables in the environment to reflect the new partial design. This updating consists of removing entries from the relationship tables for control structures which have all their objects completed. The software metrics are then computed for the current partial design, and the process continues by reentering box a. Figures 6.4 through 6.6 summarize the designer process.

The next section will illustrate the elaboration process by tracing a sample algorithm through two design steps.

6.6 An Illustration of The Design Step Process

To illustrate the flow of the design process, the algorithm of Figure 6.7 will be used. We will trace through two design steps.

The starting point is the outer most graph control structure, PROG. First its label is marked, then the result OUT is marked. At this point the element counts for the

```
repeat
  if not first chunk
    then choose-next-chunk
    else outer structure is next chunk;
  elaborate next chunk;
  update metrics;
  update environment;
  output metrics;
until (all structures are elaborated);
```

Figure 6.4 An overview of the designer component of the simulation model.

```
procedure choose-next-chunk;
begin
  POS := null;
  repeat
    generate random number;
    find match to random number in table POSITION;
    case match
      S: next chunk is current structure;
      R: select randomly from table NOTDONE;
      I: select randomly from table INPUTS;
      O: select randomly from table OUTPUTS;
      C: select randomly from table CONCUR;
      M: select randomly from table MODULE;
    end case;
  until (POS <> null);
end;
```

Figure 6.5 Procedure for choosing selection criteria and next chunk.

```
procedure elaborate;
begin
  ERROR := false;
  for I := 1 to 5 do
    while STRUCT[POS] is not done and ERROR is false do
      case I
        1: mark-label;
        2: mark-results;
        3: mark-operator;
        4: mark-input-operands;
        5: mark-conditional;
      end case;
end;
```

Figure 6.6 An overview of the elaboration procedure for the simulation model.

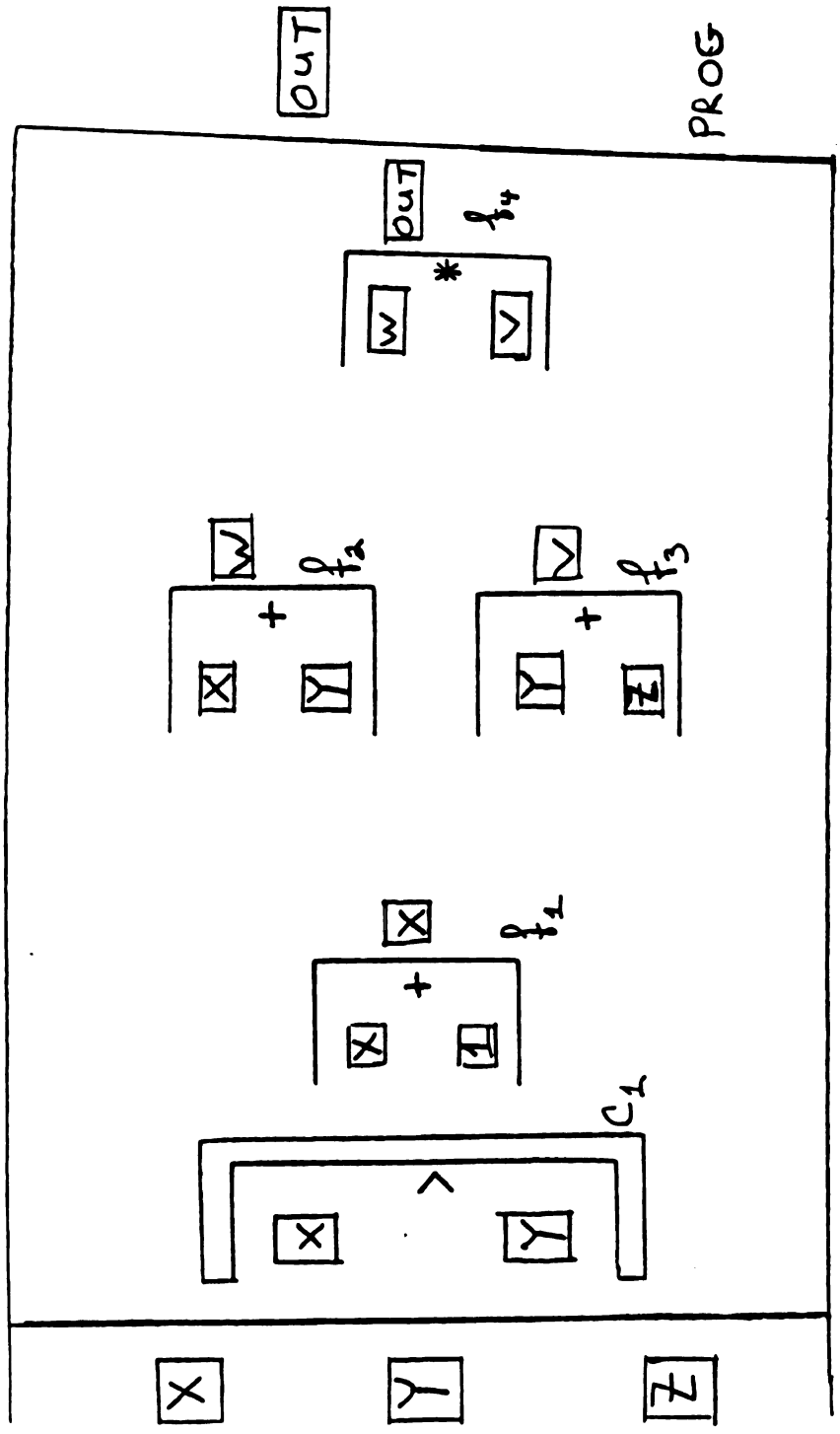


Figure 6.7 A sample algorithm to illustrate the flow of the algorithm design model.

metrics show two references and two unique items. The implied operator DO WHILE is then marked. Then the input operands, which in this case are graph control structures are marked. Assume that f_1, f_2, and f_3 are probabilistically selected and that no errors occur when they are marked, but that an error is probabilistically indicated when we try to mark f_4. Since an error occurs, f_4 is not marked and processing of the first chunk is complete. Thus at the end of design step one, the element counts and conceptual volume will be that which is found in Table 6.1.

The second elaboration is then started. Assume that a non-localized placement was selected based on the methodology probabilities. Also, assume that from the non-completed structures, structure f_2 was randomly chosen. Since f_2's label has already been marked, it is not remarked. The result W is marked, then the operator +, and finally the input operands X and Y are marked. Assume this time that no errors are generated. Since all of the objects in control structure f_2 have been added to the current partial design, this completes step two. Table 6.2 shows the updated metrics.

Now assume a localized placement is selected. Then the determination is made to use the concurrent relationship. Since f_2 started the last chunk, structure f_3 will start the next chunk. Processing continues in this fashion until the

Table 6.1 Element counts and conceptual volume for step one of the abstract model illustration.

Sequential Element	rank (i)	$F^i_{n,s}$	Parallel Element	rank (i)	$F^i_{n,p}$
PROG	1	1	f2	1	1
OUT	2	1	f3	2	1
DO...WHILE	3	1			
f1	4	1			
	4	4		2	2

$\underline{V} = (4+2) \log_2 (4+2) = 15.5$ bits.

Table 6.2 Element counts and conceptual volume for step two of the abstract model illustration.

Sequential Element	rank (i)	$F^i_{n,s}$	Parallel Element	rank (i)	$F^i_{n,p}$
PROG	1	1	f2	1	1
OUT	2	1	f3	2	1
DO...WHILE	3	1	W	3	1
f1	4	1	+	4	1
			X	5	1
			Y	6	1
	4	4		6	6

$\underline{V} = (4+6) \log_2 (4+6) = 33.2$ bits.

entire algorithm has been completed.

6.7 Summary

This chapter has presented a discussion of the algorithm design process. In order to organize and integrate the many suggestions offered in the literature, we have divided the design process into two phases; a placement process and an elaboration process. We have called the process of determining the positions in an algorithm at which to make modifications the placement process. We saw that many design methodologies, i.e., top-down or bottom-up design, focus on this process. We also examined the process of modularizing an algorithm and determining what objects should compose a module. We called this process the elaboration process. This chapter then presented a model of the algorithm design process which was based on the placement and elaboration processes. An implementation of the model as a simulation was presented. This simulation model will allow us to examine the impact of using localized and non-localized placements in the design process. Since we have divided placement decisions into two classes, those which use first order relationships and those which use higher order relationships, we will be limited in our results to comparing only first order decisions to the others as a group. While comparisons between the use of

second and third order relationships may be interesting, such comparisons are beyond the scope of this model.

In the next chapter, the simulation model will be used to examine the impact of making localized and non-localized placements on algorithms with different size parallel components.

CHAPTER 7

The Experiments

This chapter will use the simulation model of the algorithm design process which was developed in Chapter six. Experiments will be conducted to study the impact of different placement decisions on the design of parallel algorithms with different size parallel components. First, some properties of the two sample algorithms, MBFR and ROOTF, will be discussed. Section 7.2 will then describe the different design methodologies which were used in the experiments. The methodologies are described in terms of their use of localized and non-localized placements. Section 7.3 presents a preliminary discussion of the experimental results. These results suggest that the structure of an algorithm impacts a design methodology's performance. In order to examine the role of an algorithm's structure, Section 7.4 describes the differences in the

structures between algorithms ROOTF and MBFR which can be observed using the software metrics. This is followed in Section 7.5 by a discussion on how these structural differences impact the elaboration process. Section 7.6 presents a similar discussion on the placement process. This is followed in Section 7.7 by a discussion of how these results relate to human designers.

7.1 The Sample Algorithms

The ROOTF algorithm from chapter two and the MBFR algorithm from chapter five were used in the experiments. Graph language representations of the algorithms are shown in Figures 7.1 through 7.3. Both algorithms have single local parallel components. The parallel component of the ROOTF algorithm is small, while the parallel component of the MBFR algorithm is much larger. Thus, the algorithms can be used to examine the effects of the amount of parallelism present in an algorithm on the design process for algorithms in which the parallelism is localized.

Algorithm ROOTF has a structure which is predominantly sequential. Recall from Chapter two, that it was created by making a simple transformation to the sequential algorithm BISECT. Although it contains a small parallel segment, the programmer can still visualize the basic control structure as sequential. In fact, the primary parallel construct, the

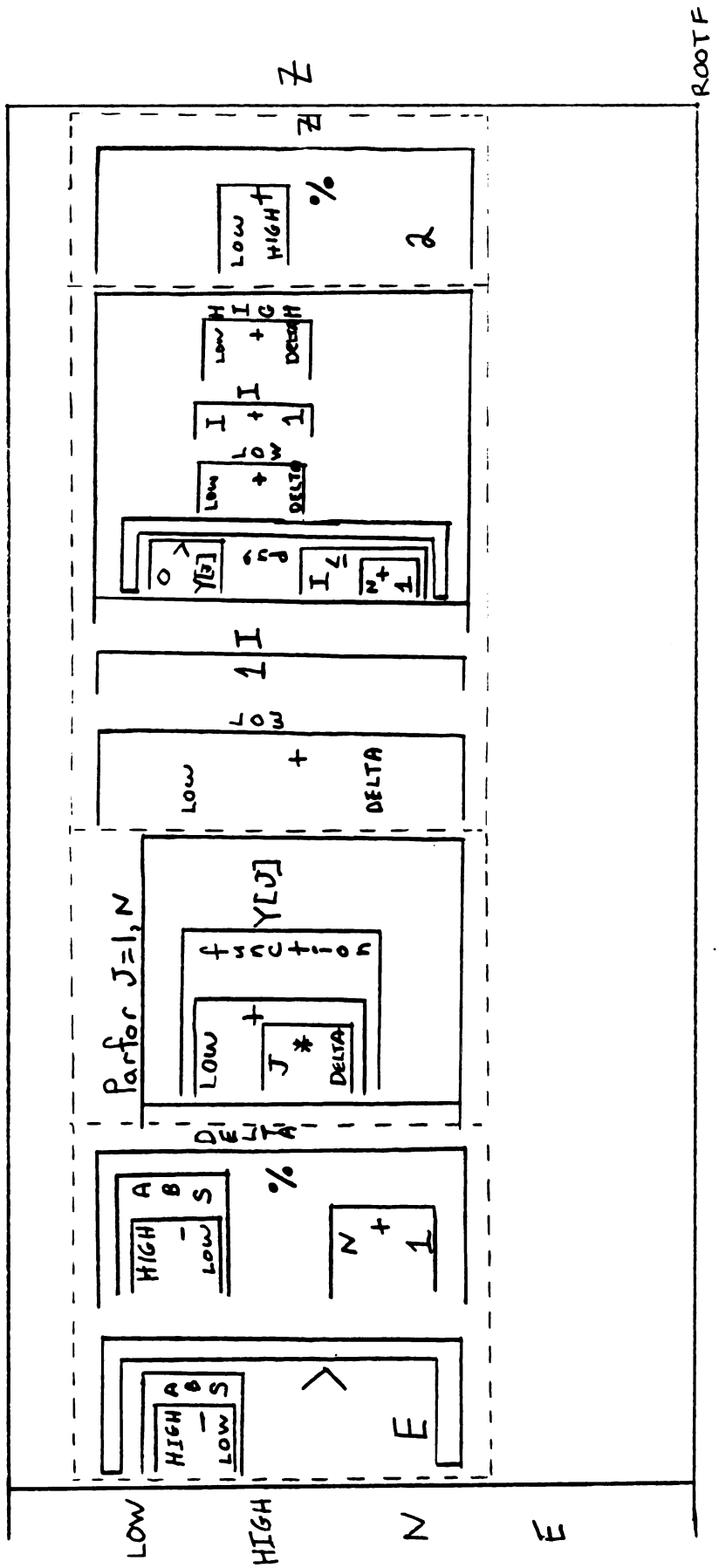


Figure 7.1 Graph language representation of the ROOTF algorithm.

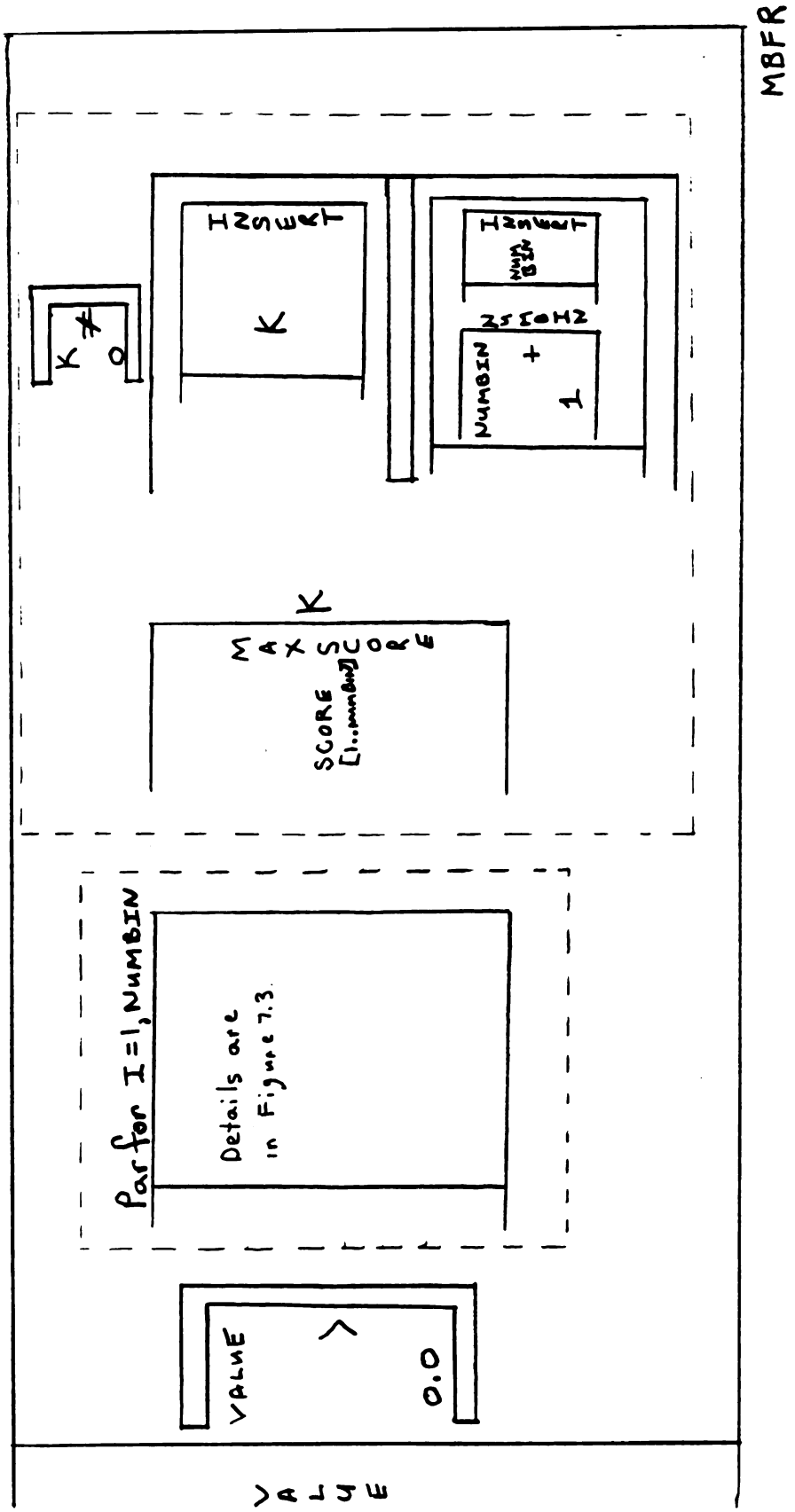


Figure 7.2 Graph language representation of the MBFR algorithm.
(The PARFOR block is expanded in Figure 7.3.)

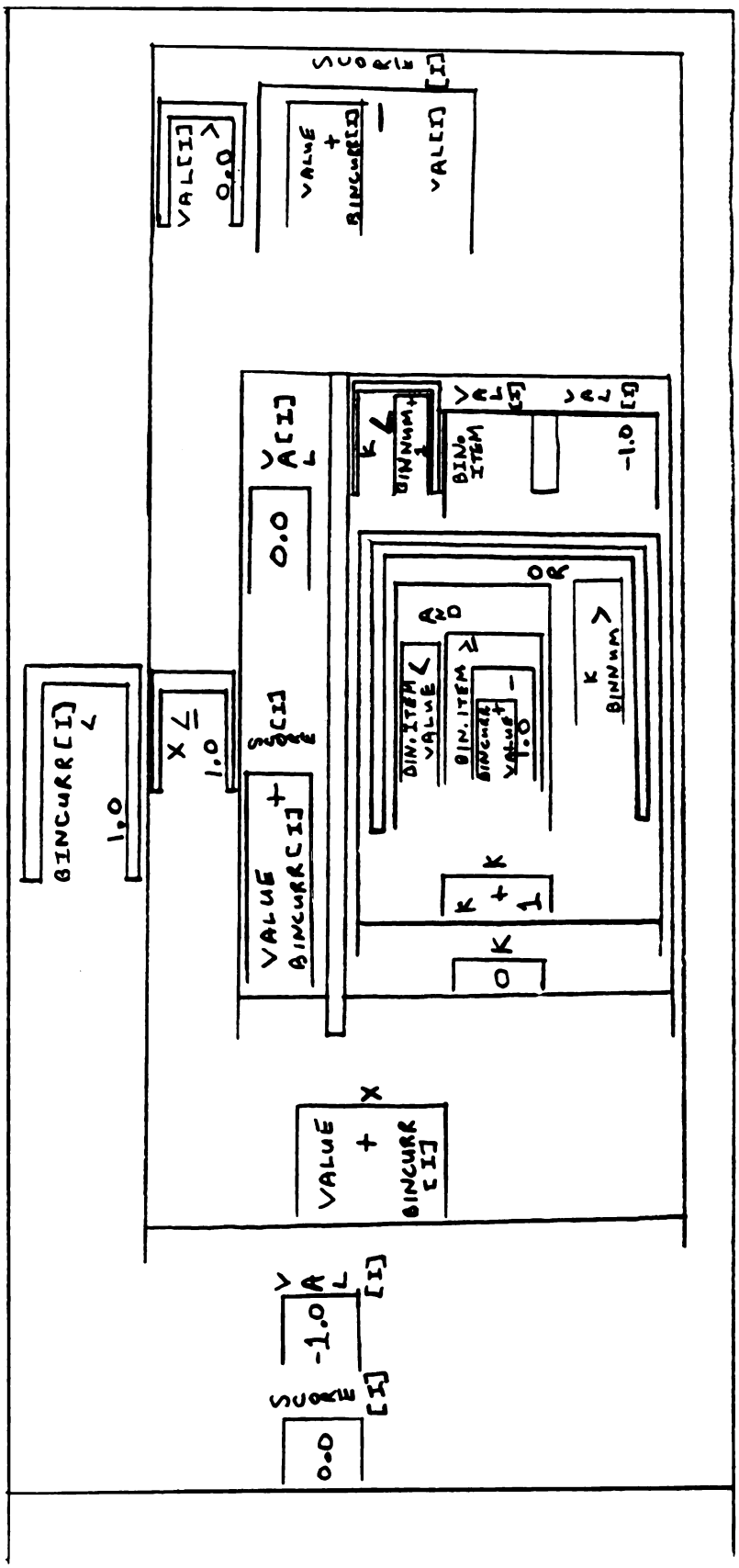


Figure 7.3 The details of the PARFOR block for the MBFR algorithm of Figure 7.2.

PARFOR statement, can be replaced by the sequential FOR statement, and the algorithm will still be correct. Therefore, the performance of the different design methodologies on algorithm ROOTF should relate to their performance on sequential algorithms as well as on algorithms with small localized parallel components.

The MBFR algorithm, on the other hand, has a larger parallel component. The conceptual volume of its parallel component is 652.3 bits versus 53.7 bits for algorithm ROOTF. It contains a more complex interface between the sequential and parallel components. Unlike the root finding algorithm, the number of computations made by the parallel processes will not be the same. It will require more time to determine the bid for a bin with a large number of items, than for a bin which is almost empty. This requires the designer to be more aware of synchronization between the processes. Likewise, the designer needs to deal with more communication between the concurrent processes and the master process. ROOTF only requires passing the value of the function at the subinterval's midpoint. MBFR must pass a bid and the value of an item to replace, or else communicate to the master that those values are null. Therefore, the performance of the design methodologies on algorithm MBFR will be representative of their performances on other, more complex, parallel algorithms.

7.2 Design Methodologies Used In The Experiments

The design of each algorithm was simulated using five classes of design methodologies. Each design methodology was represented by a probability mix. The mixes are shown in Table 7.1. For each mix, the probability of a localized placement was evenly divided among the four first order relationships. The set of mixes was designed to examine the effects of using only localized placements, non-localized placements, or various combinations of both types of placements. Although the use of 100% high order placements can be modeled directly, the nature of the model developed here precluded the use of only first order localized placements. This is because not all possible localized relationships are identifiable in the graph language. All algorithm structures are related to all other structures in an algorithm, but only a subset of these possible relationships are considered by the model. These missing relationships present a difficulty for the model. It is possible that they are the only relationships for the last elaborated control structure. If this is the case, the model would not be able to select another structure using a first order relationship. This would cause the model to deadlock. Thus, the use of predominantly localized

Table 7.1 The design methodology probability mixes used in the parallel algorithm design experiments.

Mix	non-localized percentage	localized percentage
1	100	0
2	75	25
3	50	50
4	25	75
5	5	95

placements was modeled as 95% localized placements and 5% non-localized placements.

In Figures 7.1 and 7.2, the high level modular decompositions which were used for the first order module placement relationships can be seen. The graph control structures within a dashed box form one possible high level modular decomposition. These modular decompositions were used for all of the experiments.

The design mixes in Table 7.1, represent the use of varying combinations of localized and non-localized placements. By modeling design methodologies using these mixes, we will be able to see whether there is a correspondence between the distribution and granularity of the parallel information in an algorithm, and the type of information which should be used to design such algorithms.

7.3 The Experiments

In the experiments, the designs of the root finding and bin packing algorithms were simulated. Since the model is stochastic, 20 trials were run for each algorithm and design mix. Table 7.2 gives the average number of design steps for each set of trials.

In Table 7.2, we can see several things. First, for each design mix, on the average, it took more steps to design algorithm MBFR than algorithm ROOTF. If we look at

Table 7.2 The average number of design steps which were required for each design mix, for algorithms MBFR and ROOTF.

Mix	Average Number of Design Steps	
	ROOTF	MBFR
1	49.0	70.2
2	51.3	67.3
3	50.5	70.0
4	50.8	72.3
5	48.0	72.2

just algorithm ROOTF, we can see that each design mix, on the average, took a different number of design steps. The same observation can be made between the design mixes for algorithm MBFR. These observations suggest two things. First, the design mixes caused different amounts of information to be added to each of the two algorithms. Second, for a given algorithm, different design mixes cause information to be added at different rates. Thus, we can see that not only are there differences in the performances of a design mix between the two algorithms, there are also differences in the performances of the five design mixes for a given algorithm.

Table 7.3 gives the average amount of information added to the ROOTF and MBFR algorithms as the result of a design step, for each design mix. Also shown are the differences in the averages for a design mix between the two algorithms. In Table 7.3, we can see that an average design step always added more information to algorithm MBFR than to algorithm ROOTF. But, by examining the differences in the amount of information added between the two algorithms, we can see that the amount of these differences is not constant. If we look at the differences for mixes 2 through 5, we can see that they decrease from 2.16 bits to 0.41 bits. If we look at just algorithm ROOTF, we can see that as we move from design mix 2 to mix 5, that the averages increase from 11.71 bits to 12.52 bits. But, if we look at the same averages

Table 7.3 The Average information (bits of conceptual volume) added to the algorithm by a design step for each design mix.

Mix	Average Information Added		Difference Between the Averages For Algorithms MBFR and ROOTF
	ROOTF	MBFR	
1	12.26	13.30	1.04
2	11.71	13.87	2.16
3	11.90	13.34	1.44
4	11.83	12.91	1.08
5	12.52	12.93	0.41

for algorithm MBFR, the averages decrease from 13.87 bits to 12.93 bits. Thus, we can see that for our two algorithms, the design mixes behaved in different ways. A given mix caused information to be added to algorithm ROOTF at a different rate than to algorithm MBFR. Also, for a given algorithm, the different mixes caused information to be added at different rates.

We can see further evidence of these differences by examining Figure 7.4. In Figure 7.4, a temporal estimate of five minutes was associated with a design step. Recall, that a design step represents processing a chunk. Therefore, it represents a non-trivial set of decisions. Five minutes was selected as a rough estimate of the average time it would take to make those decisions. Using this estimate, the average design time for each algorithm and design mix was plotted. Design mix one was omitted from the plots, because it represents the use of only non-localized placements. Since some first order relationships would normally be used in the design process, and mixes 2 through 5 represent the increasing use of first order placements, they are of more use to us here. We can see immediately from Figure 7.4, that the MBFR algorithm, on the average, took longer to design than the ROOTF algorithm. This to be is expected, since it is larger. But, we can also see that while mix 2 took the least time to design algorithm MBFR, it took the most time to design algorithm ROOTF. Also, mix 5

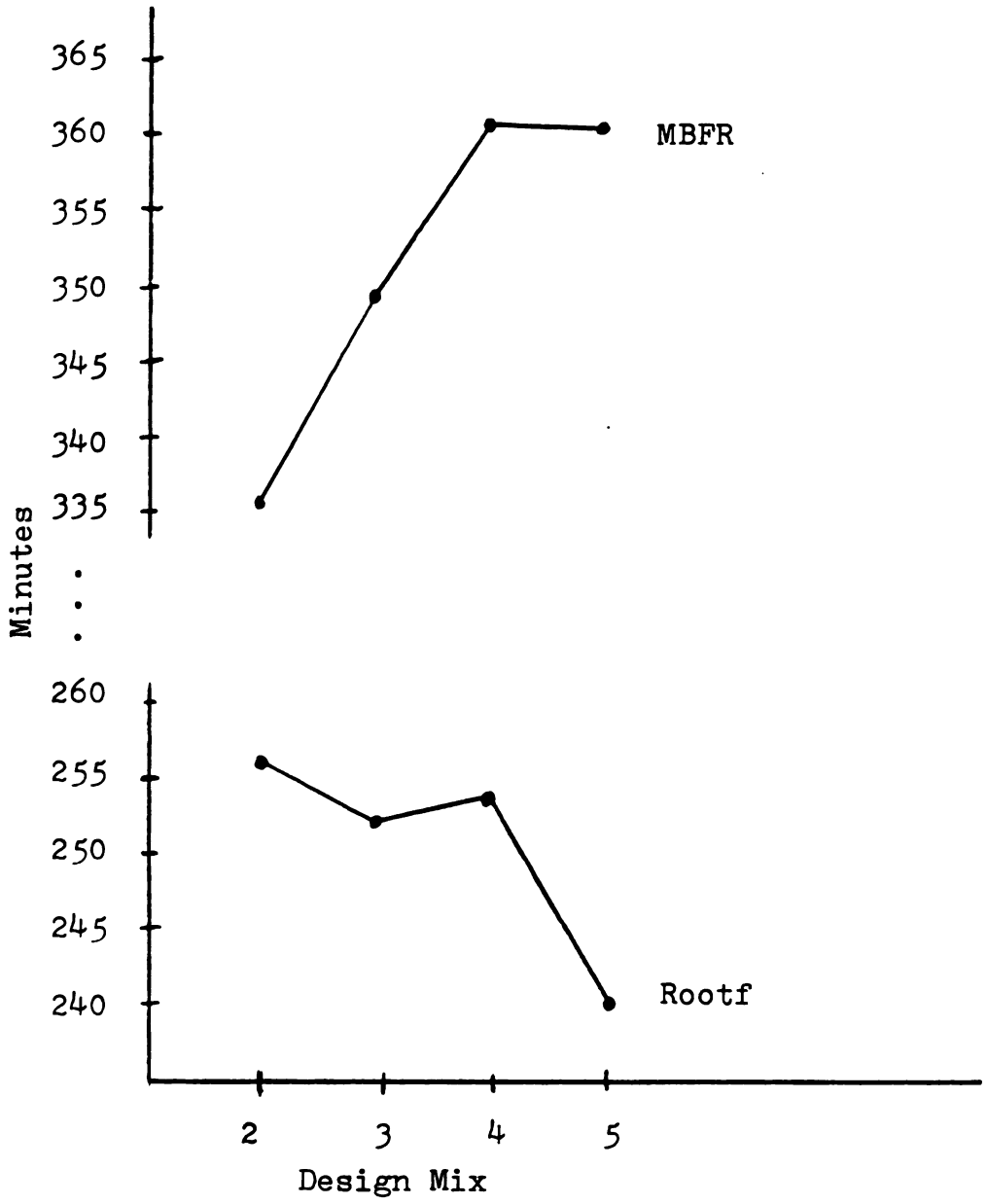


Figure 7.4 Average design time, in minutes, for algorithms ROOTF and MBFR as a function of the design mix.

took the least time for algorithm ROOTF, but the most time for algorithm MBFR. Therefore, we can see that the design methodology which took the least time differed between the two algorithms.

The results from Tables 7.2 and 7.3, and Figure 7.4 suggest two major observations. First, the choice of a design methodology affects the rate at which information is added to an algorithm during its design for the two sample algorithms. Second, since the performance of the design methodologies differed between the two algorithms, the structure of an algorithm, impacts the performance of a design methodology. We will examine how the two phases of a design step are impacted by an algorithm's structure in Sections 7.5 and 7.6. First, however, we will look at the differences in the structures of algorithms ROOTF and MBFR.

7.4 Structural Differences Between Algorithms ROOTF and MBFR

In Table 7.4, several differences in the structures of algorithms ROOTF and MBFR are identified. The first obvious difference between the algorithms is their size. Algorithm MBFR is about 55% larger than algorithm ROOTF. MBFR has a conceptual volume of 934 bits, while the conceptual volume of algorithm ROOTF is 601 bits. Likewise, the sizes of the parallel and sequential components differ. The parallel

Table 7.4 Structural differences between algorithms ROOTF and MBFR.

	ROOTF	MBFR
Conceptual Volume ($\underline{V}$ in bits)	601	934
Parallel Component Conceptual Volume ($\underline{V}_p$)	53	652
Sequential Component Conceptual Volume ($\underline{V}_s$)	480	160
Ratio of Parallel Component Conceptual Volume to Conceptual Volume ($\underline{V}_p$ / $\underline{V}$)	.09	.70
Ratio of Sequential Component Conceptual Volume to Conceptual Volume ($\underline{V}_s$ / $\underline{V}$)	.91	.30

component of algorithm MBFR has a size of 652 bits versus 53 bits for algorithm ROOTF. The sequential component of algorithm MBFR has a size of 160 bits versus 480 bits for algorithm ROOTF. In algorithm ROOTF, the sequential component dominates the conceptual volume, while in algorithm MBFR, the parallel component dominates. For algorithm ROOTF, 9% of the volume is the result of the parallel component and 91% is the result of the sequential component. On the other hand, in algorithm MBFR, the distribution is 70% parallel and 30% sequential.

Thus, our metrics have allowed us to see several differences in the informational structure of algorithms ROOTF and MBFR. In the next sections, we will see how these structural differences impact the design process. First, we will examine how they impact the elaboration process in Section 7.5. Then, we will examine their impact on the placement process in Section 7.6.

7.5 Effects of the Algorithm's Structure on the Elaboration Process

Since algorithm MBFR is larger than algorithm ROOTF, we might expect that there are more unique objects in it. Since conceptual volume is based on the number of unique objects as well as on references to them, this might cause the addition of an object to algorithm MBFR to have a larger

informational impact. Recall that $\underline{V} = N \log_2 n$. Since algorithm MBFR has 73 unique object-context pairs, and algorithm ROOTF has only 51, this explains some of the difference in the average information added by a design step, between the two algorithms. But, it does not explain the opposite trends in the average information added for each design mix, i.e., why does design mix 5 add more information than mix 2, for algorithm ROOTF, whereas mix 2 adds more information than mix 5 to algorithm MBFR. Therefore, we must look for other reasons why the added information added by a design step varies between the two algorithms.

Since algorithm MBFR is larger than algorithm ROOTF, it has more graph control structures. And, as can be seen in Figures 7.1 through 7.3, its graph control structures are nested to more levels than those in algorithm ROOTF. Thus, the size of a chunk may potentially be larger in algorithm MBFR, than in algorithm ROOTF. Recall, that a chunk must be contained in a functional unit. A functional unit is denoted by a graph control structure and those graph control structures which are nested in it. Since more items may be in a chunk in algorithm MBFR, more information may be added to the algorithm, on the average, by a design step.

The number of object-context pairs in the algorithms and the sizes of the chunks, seem to explain why the amount of information added by a design step varies between the two

algorithms. The next section will examine why the five design mixes performed differently for a given algorithm.

7.6 Relationship Between the Placement Process and the Rate Information is Added to an Algorithm's Design

The results from the previous section suggest reasons why the design mixes added information to the two algorithms at different rates. We will now examine why, for a given algorithm, the five design mixes performed differently. Since the impact of the elaboration process is the same for for all of our design mixes, we will examine the impact of an algorithm's structure on the placement process.

The placement process is used to select the next structure to add to an algorithm's design. Depending on the relationship which is used by the placement process, and the structure of the algorithm, the new structure may cause a context shift. Recall, the context of a structure represents whether it is in a sequential or a parallel component of an algorithm. The context under which an object is referenced, affects the volume metric. Recall, $V = N \log_2 n$, where N is the number of objects referenced and n represents the number of unique object-context pairs. Thus, a reference to an object in a new context has a larger impact on the volume, than a reference to the object in the old context. Therefore, a placement decision which causes a

context change may result in information, as measured by the volume metric, being added to an algorithm at a higher rate.

If an algorithm is designed using placement decisions which cause context changes late in the design of an algorithm, the result will be a large jump in the information growth rate when the context change occurs. Since N will be larger late in the design of an algorithm, a small change in n will have a large impact. If context changes occur early, the initial growth rate will be higher, but the effects of the context change will not be as significant. Since N will be small early in the design process, a change in n will not have as large of an impact on the volume as it would later. Therefore, while the total amount of information added to an algorithm is the same, how early or late in an algorithm's design that context changes occur and the number of context changes, will affect the rate at which this information is added.

In the model, the choice of a placement decision may affect when a context change can take place. Whether a placement decision causes a context change or not, may differ between two algorithms. We will now look at the placement decision types, which were used in our model, to see how this might happen. Since the concurrent relationship was not used for our two sample algorithms, it will not be considered here.

The input and output relationships connect structures based on the use of the same data. However, this data might be used in both the sequential and parallel components of an algorithm. In algorithm ROOTF, 60% of the structures which have input and output relationships, are connected to a structure in a different context. In algorithm MBFR, only 35% of the structures which have input and output relationships, are related to structures in a different context. A higher percentage of these structures are related to structures in a different context in algorithm ROOTF. Thus, input and output placement decisions may cause a context change more often in algorithm ROOTF, than in algorithm MBFR. Therefore, a design mix which places an emphasis on the use of the input and output relationships, may have different impacts on the two sample algorithms.

The module relationship is used to make a placement based on a high level modular decomposition of an algorithm. The decompositions which were used in the experiments, only relate structures which are in the same context. Therefore, the use of a module placement decision has the same impact on both algorithms.

The use of a non-localized placement may cause a context change in algorithm ROOTF, with a different probability than in algorithm MBFR. In algorithm ROOTF, 9% of the algorithm represents the parallel component and 91% of the algorithm represents the squential component. In

algorithm MBFR, the distribution is 70% parallel and 30% sequential. Therefore, when a structure is randomly selected using a non-localized placement, there is a higher probability of a context change in algorithm MBFR than in algorithm ROOTF. Thus, a design mix which emphasizes non-localized placements, may cause a context change more often in algorithm MBFR, and thus, add information to it at a higher rate.

It appears that the input, output, and non-localized placements may be responsible for controlling context changes, in our experiments. Since input and output are localized placements, mix 5 represents the highest probability of their use. Mix 5 also represents the smallest use of non-localized placements. As Table 7.3 shows, when algorithm ROOTF was designed using mix 5, as expected, it had the highest average information growth rate. This was expected, since mix 5 allows for the highest use of the input and output placement relationships and the smallest use of non-localized placements and thus, allows the most context changes for algorithm ROOTF. Mix 2 represents the use of localized placements only 25% of the time. When mix 2 was used to design algorithm ROOTF, the information growth rate was the lowest. This was also expected, since a non-localized placement for algorithm ROOTF has a low probability of a context change. Earlier, when we discussed the input and output placements, we saw

that they were less likely to cause a context change for algorithm MBFR than for algorithm ROOTF. But, non-localized placements had a higher probability of causing a context change for algorithm MBFR. Therefore, we might expect different performances for the two algorithms from the design mixes based on the impact of the input, output, and non-localized placements. Indeed this happened. For algorithm MBFR, mix 2 caused the highest information growth rate, and mix 5 caused the smallest.

Thus, we can see that the structure of an algorithm impacts the way in which a design methodology adds information to the algorithm, as the result of a design step. One way this occurs, is by promoting or preventing context changes, when structures are selected for elaboration. The next section will consider how these context changes are related to the activities of human designers.

7.7 Correspondence Between the Model's Results and Human Designers

We will now attempt to explain the model's behavior in terms of real designers. Since humans are limited in the amount of information which they can deal with at a particualr moment [16], it would be useful if a design methodology could help control the amount of information

which must be used to make a design decision. Some parts of a parallel algorithm can be designed independent of the other parts. Therefore, less information is needed to design these independent parts, than if there were dependencies between them. However, process synchronization and intercommunication often requires the designer to consider several processes, when making a design decision about these activities. Some of these activities may be in a parallel portion of the algorithm, while other of these activites may be in a sequential portion of the algorithm. Thus, conceptual differences between the objects in the parallel and sequential components may impact the amount of information which the designer must deal with. These conceptual differences correspond to a context change in the design model. Thus, we can see that if these aspects of a parallel algorithm are designed early, their informational impact on the algorithm will be less than if they are considered at the end of the design process. Since an algorithm only has a small information content at the beginning of the design process, a complex decision will be easier to make then, than if the decision is made later in the design process. This occurs, since the information content of the algorithm is higher later in the design process. Although a context change still causes an increase in the rate at which information is added to an algorithm early in the design, that increase will be less than if the

context change occurs later in the design process.

Others have recognized this in their discussions on parallel algorithm design [32-34]. Although they have not explicitly considered conceptual differences between objects in sequential and parallel components of an algorithm, the increased informational complexity of considering objects in both contexts at the same time, underlies some of their suggestions. For example, Brinch Hansen in [32] discusses development procedures for concurrent algorithms. He divides an algorithm into three components; monitors, classes, and the process. Monitors represent synchronization activities; classes represent shared data items for process intercommunication; and the process represents the remaining sequential-like activities of an algorithm. He suggests designing the monitor and class components first. The remainder of the algorithm can then be designed almost as a sequential algorithm. In other words, he suggests making decisions which involve both parallel and sequential activities early, when the size of the algorithm is small and thus, the amount of information which must be considered from the previous design steps is limited. The process is basically sequential in nature, and only deals with information in a single context. Thus, the effective information burden on the designer is less, than that of the monitor and class components. The lack of context changes when designing the process component of the

algorithm, causes less information to be added to the algorithm, when the process component is designed.

To further demonstrate the efficacy of this approach, an example will be presented. In [33], Coffman and Denning present an example of two cooperating sequential processes, the producer/consumer model. Figure 7.5 shows an adaptation of their model in complete detail. The Producer generates an object; the object is put into a common buffer; and it is then retrieved by the Consumer and disposed of. N represents the number of buffers; array elements B[0..N-1] are the buffers; and S1 and S2 are semaphores. Assume that statements S and W are indivisible. This is necessary for the semaphores to function correctly.

Note from Figure 7.5, that if the designer tries to consider the processes as a whole, not only must the action of the individual processes be considered, but the intricate details of the semaphores and intercommunication must be considered as well.

Figure 7.6 shows how the design of the producer and consumer processes can be simplified, by first addressing process synchronization and intercommunication. Now the manipulation of the semaphores can be designed independent of the processes which use them. Likewise, the details of managing the buffers for process intercommunication can be removed from the design of the producer and consumer processes. By focusing on intercommunication and process

```
begin
 cobegin

   Producer:  produce(w);
          W:  S1 := S1 - 1;
          L:  if S1 < 0 then goto L;
              B[in] := w;
              in := (in + 1) mod N;
          S:  S2 := S2 + 1;
              goto Producer;


   Consumer:
          W:  S2 := S2 - 1;
          L:  if S2 < 0 then goto L;
              w := B[out];
              out := (out + 1) mod N;
          S:  S1 := S1 + 1;
              consume(w);
              goto Consumer;


 coend
end.
```

Figure 7.5 The producer and consumer processes in full detail.

```
   Input(w):  B[in] := w;
              in := (in + 1) mod N.

  Output(w):  w := B[out];
              out := (out + 1) mod N.

    Wait(S):  S := S - 1;
           L: if S < 0 then goto L.

    Send(S):  S := S + 1.


begin
 cobegin

   Producer:  produce(w);
              Wait(S1);
              Input(w);
              Send(S2);
              goto Producer;

   Consumer:  Wait(S2);
              Output(w);
              Send(S1);
              consume(w);
              goto Consumer;

 coend
end.
```

Figure 7.6 The producer and consumer processes with the synchronization and process intercommunication details removed.

```
        Xmit:  Wait(S1);
               Input(w);
               Send(S2).

     Receive:  Wait(S2);
               Output(w);
               Send(S1).


begin
 cobegin

   Producer:   produce(w);
               Xmit(w);
               goto Producer;

   Consumer:   Receive(w);
               consume(w);
               goto Consumer;

 coend
end.
```

Figure 7.7 The producer and consumer processes with all parallel details removed.

synchronization first, we are forced to define a large number of context-operand pairs early in the design process. This is consistent with the approach taken in the model of making n large at the beginning of the design process. Although the information growth rate will be intially higher, the information growth rate will not be as large later in the design process. The designer must still deal with semaphores in the code of Figure 7.6. They are only dealt at a higher level. By combining both synchronization and process intercommunication, and treating their design first, the design of the producer and consumer processes becomes almost trivial, as seen in Figure 7.7. Since the parallel activities have been designed in a seperate module, these complex decisions have been made without the designer having to consider the information which represents the rest of the producer and consumer processes.

7.8 Summary

Our experiments with the ROOTF and MBFR algorithms have shown that the choice of a design methodology may impact the time it takes to design an algorithm. Also, the structure of an algorithm may impact the performance of a design methodology. Our metrics have allowed us to see that a design methodology which has a high probability of causing a context change late in an algorithm's design, will cause

information to be added to an algorithm at a higher rate than if they occur early in an algorithm's design. When designing a parallel algorithm, it is sometimes necessary to deal with objects that occur in both the sequential and parallel contexts, during a single design step. This may occur when designing the process synchronization and intercommunication activities. A decision which involves objects in both contexts is more complex late in an algorithm's design, than if it occurs early. Therefore, it may be easier for a designer to consider these aspects of a parallel algorithm early in the design process. Those parts of the algorithm which are strictly sequential or strictly parallel, can easily be handled later in the design process. Since they only deal with a single context, the informational impact of designing them is the same late in the design process, as it would be if they were designed early.

It was also seen in this chapter, that these results correspond to suggestions that have been made by others for designing parallel algorithms. They suggest a similar order in designing the different components of a parallel algorithm. These suggestions were based on intuition. We now have some quantitative backing for them.

CHAPTER 8

Conclusions

This dissertation began with the premise that parallel algorithms contain structural and semantic information regarding process synchronization and intercommunication, which is not present in sequential algorithms. The impact of this additional information on the design process for parallel algorithms was examined.

The first questions this dissertation addressed are as follows. Can this additional information be quantified? How much more information does a parallel algorithm contain? And, where in an algorithm are the effects of this added information seen? Software metrics were presented which described the information content of parallel algorithms. The metrics are based on those suggested by Halstead, but were extended to consider conceptual differences between those algorithm elements which are found in sequential

versus parallel portions of an algorithm. By examining a sequential root finding algorithm and a parallel version, which was derived from it, we saw that indeed the parallel root finding algorithm does contain additional information. It was also seen that this additional information is not necessarily spread uniformally throughout the algorithm, but it may be concentrated in a small portion of the algorithm. In fact, the bulk of the added information may not necessarily be found in the parallel portion. It may appear in the sequential component. This is a result of the need for the sequential component to support the parallel portion of the algorithm. This support has been given the name subsidization.

Given that parallel algorithms may contain additional structural and semantic information, the next question which was addressed was; should this added information affect the way in which we design parallel algorithms? In order to examine this question, a model of the algorithm design process was created. The model was based on the information content of an algorithm, and the information which is used during the design process.

The design process was modeled as a series of design steps. A design step consists of a placement process and an elaboration process. The placement process determines the positions in an algorithm at which to make a modification. The elaboration process makes those modifications.

Placement decisions were divided into two sets. The first set is those decisions which are based on relationships to those objects which were added to an algorithm by the last design step. The second set represents those decisions which are based on any relationships except those which involve objects added to an algorithm by the last design step. Placements which are made based on the decisions in the first set were called localized placements. Placements which are made based on the decisions in the second set were called non-localized placements. Design methodologies were modeled by using various combinations of localized and non-localized placements.

The designs of a parallel root finding and a parallel bin packing algorithm were simulated. It was seen from the results of these simulations, that the design methodologies performed differently for the two algorithms. Also, for a given algorithm, the performance of different design methodolgies varied. It was suggested that the structures of the algorithms were responsible for these different performances. We saw that the elaboration process was impacted by the number of unique object-context pairs in an algorithm, and it was also impacted by the levels to which the structures in the algorithms were nested.

The placement process was impacted by placement relationships which allow a context change to occur between design steps. A context change early in the design process

has less of an impact on an algorithm's volume, than a context change will have if it occurs later. Therefore, it was suggested that a designer treat those aspects of a parallel algorithm, which may involve dealing with objects that are in both sequential and parallel components of an algorithm, first. Process synchronization and intercommunication often involve objects that are in both sequential and parallel components of an algorithm. Therefore, if they are designed first, the decisions which design them will be less complex, than if they are designed later in the design process when the algorithm will be larger. The rest of the activities of a parallel algorithm, involve objects that are primarily in a single component. Therefore, the complexity of designing these remaining activities will be the same late in the design process, as it would be if they were designed at the beginning. Thus, by designing a parallel algorithm's process synchronization and intercommunication activities first, the overall design process can be simplified. These quantitative predictions seem to be confirmed in the literature. Brinch Hansen and others have made similar suggestions for designing parallel algorithms. Their suggestions have been based on intuition. We now have some quantitative support for them.

The next appropriate step for this research would be to repeat the experiments for other parallel algorithms. Although the ROOTF and MBFR algorithms are representative of

other parallel algorithms with single localized parallel segments, it would be useful to study other such algorithms. Also, it would be interesting to study whether the trends, which were observed here, apply to parallel algorithms in which the parallelism is distributed. Further experiments using our model would offer some insights into this.

In conclusion, this research has demonstrated that there may be informational differences between parallel and sequential algorthms. These differences are primarily concerned with structural and semantic information which must be considered in designing parallel algorithms, but not in designing sequential ones. These differences in information have an impact on the design process for parallel algorithms. It is anticipated, that as we come to a better understanding of these underlying differences between sequential and parallel algorthms, that we will be able to arrive at design techniques which will simplify the design of parallel algorithms. This will also aid in designing languages and operating systems for parallel architectures. The combination of better design techniques and a better design environment, should facilitate the efficient use of parallel architectures.

APPENDIX

APPENDIX A

The Bin Packing Algorithm

A pseudo-code description of the MBFR algorithm is given below. The algorithm repeats for each item in the input list. The algorithm assumes all nonempty bins make a bid for each new item, but in practice, only nonempty bins which are also not full would do so. This was done to simplify the algorithm.

Null bids by active processors for an item are coded as 1 for the bid, and 0 for the replaced item. The master collects the bids and selects the minimum bid which is less than 1. If the new value is greater than 0 for the selected bin, the item is removed from the bin. Next the new item is placed into the selected bin, and the gap is updated. The algorithm then repeats for the removed item.

If there were no bids less than 1, the master inserts the new item into the next empty bin and calculates the appropriate gap. If items are replaced, notice that they get progressively smaller. An item cannot displace one which is larger than itself. Therefore, the replace/insert cycle will always terminate. a parallel alg is being constructed,

Algorithm MBFRBINPACK

```
NUMBINS:=1;                                  *initialize first bin*
GAP[1]:=1;
READ(VALUE)
WHILE NOT END OF LIST DO                     *process list*
BEGIN
   WHILE(VALUE)≠ 0 DO                        *pack initial value as well as exchanges*
      PARFOR I:=1 TO NUMBINS DO              *request bids*
         IF VALUE<=GAP[I]                    *bins from G1 set*
            THEN
               BEGIN
                  BID[I]:=GAP[I]-VALUE;
                  NEWVALUE[I]:=0
               END
            ELSE                             *bins from G2 set*
               BEGIN
                  NEWVALUE[I]:=FINDX(VALUE); *FINDX returns maximum value of bin item
                                              for which GAP+item-value>0 is true. If no
                  IF NEWVALUE[I]=0            item exists it returns to a 0*
                     THEN BID[I]:=1
                     ELSE BID[I]:=NEWVALUE[I]+GAP[I]-VALUE
               END;
      SELECT:=MIN(BID);                      *MIN chooses the bin with the minimum
                                              of all bids, and returns its index*
```

```
      IF BID[SELECT]<1
            THEN IF NEWVALUE[SELECT]>0
                     THEN
                        BEGIN
                           REMOVE(NEWVALUE[SELECT],SELECT    *removes NEWVALUE from the
                                                             selected bins list*
                           INSERT(VALUE,SELECT)              *inserts VALUE into selected
                                                                                  bin*
                           UPDATE G2(VALUE,NEWVALUE[SELECT],GAP[SELECT],SELECT)
                                                             *updates GAP accordingly*
                           VALUE:=NEWVALUE[SELECT]           *new VALUE to pack on next
                                                             pass*
                        END
                     ELSE
                        BEGIN
                           UPDATE G1(VALUE,GAP[SELECT],SELECT)  *update G1 bin by sub-
                                                                tracting VALUE from
                           VALUE:=0                             the GAP for the selected
                        END                                     bin*
            ELSE
               BEGIN
                  NUBINS:=NUBINS+1                 *or start a new bin by setting GAP of
                  OPEN NEW BIN(NUBINS,VALUE);      NEWBIN equal to 1-VALUE*
                  VALUE:=0
               END
         END                                        *of pack exchange loop*
      READ(VALUE)
   END                                             *of list process loop*
```

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] D. J. Kuck, Structure of Computers and Computations, Wiley, New York, 1978.

[2] J. L. Baer, Computer Systems Architecture, Computer Science Press, Inc., 1980.

[3] D. J. Kuck, "Measurements of Parallelism in Ordinary FORTRAN Programs", Computer, January, 1974, pps. 37-46.

[4] E. W. Dijkstra, "GOTO Statement Considered Harmful", CACM, Vol. 11, No. 3, March, 1968, pps. 147-148.

[5] P. Freeman and A. Wasserman (editors), Tutorial on Software Design Techniques, IEEE Computer Society, 1980.

[6] P. J. Brown, "Programming and Documenting Software Projects", Computing Surveys, Vol. 6, No. 4, December, 1974, pps. 213-220.

[7] J. M. Yohe, "An Overview of Programming Practices", Computing Surveys, Vol. 6, No. 4, December, 1974, pps. 221-245.

[8] N. Wirth, "On the Composition of Well-Structured Programs", Computing Surveys, Vol. 6, No. 4, December, 1974, pps. 247-259.

[9] D. E. Knuth, "Structured Programming with GOTO Statements", Computing Surveys, Vol. 6, No. 4, December, 1974, pps. 261-301.

[10] B. W. Kernighan and P. J. Plauger, "Programming Style: Examples and Counter Examples", Computing Surveys, Vol. 6, No. 4, December, 1974, pps. 303-319.

[11] W. B. Ackerman, "Data Flow Languages", Proceedings NCC, Vol. 48, 1979, pps. 1087-1095.

[12] Per Brinch Hansen, "The Programming Language Concurrent Pascal", IEEE Transactions on Software Engineering, Vol. SE-1, No. 2, June, 1975, pps. 199-207.

[13] H. T. Kung, "The Structure of Parallel Algorithms", Advances in Computers, Academic Press, New York, 1980.

[14] S. Owicki and D. Gries, "Proving Parallel Programs Correct", Structured Multiprogramming, Springer Verlag, 1978.

[15] W. Harrison, K. Magel, R. Kluczny, and A. Dekock, "Applying Software Complexity Metrics to Program Maintenace", Computer, Vol. 15, No. 9, September, 1982, pps. 65-79.

[16] M. Halstead, Elements of Software Science, North-Holland, New York, 1977.

[17] R. G. Reynolds and G. Cichanowski, "Metrics for the Comparision of Parallel Algorithms and Their Design Methodologies", Symposium on Empirical Foundations of Information and Software Science, Atlanta, Georgia, November, 1982.

[18] S. Baase, Computer Algorithms, Addison-Wesley, Reading, Massachusetts, 1979.

[19] C. Gear, Applications and Algorithms in Computer Science, Module A, SRA Press, 1978.

[20] R. G. Reynolds and T. L. Chang, "The PAL System - a Parallel Algorithm Design System for VLSI Based Array Architectures", Tenth IMAX Conference, Montreal, August, 1982.

[21] Chi-Chih Yao, A., "New Algorithms for Bin Packing", J. Assoc. Comput. Mach., Vol. 27, No. 2, pps. 207-227.

[22] Kou, L. T., Markowsky, G., "Multidemensional Bin Packing Algorithms", IBM J. Res. and Dev., Vol. 21, No. 5, pps. 443-448.

[23] Coffman, E. G., Garey, M. R., Johnson, D. S., "An Application of Bin Packing to Multiprocessor Scheduling", Siam J. Comput., Vol. 7, No. 1, pps. 1-17.

[24] Shapiro, S. E., "Performance of Heuristic Bin Packing Algorithms with Segments of Random Length", Inf. and Control, Vol. 35, No. 2, pps. 146-158.

[25] Coffman, E. G., Jr., So, K., Hofri, M., Yao, A. C., "A Stochastic Model of Bin Packing", Inf. and Control, Vol. 44, No. 2, 105-115.

[26] Johnson, D. S., "Near Optimal Bin Packing Algorithms", MIT Report, MAC TR-109, June, 1973.

[27] Graham, "Bin Packing", in, Computer and Job-Shop Scheduling Theory, Coffman, E. G., editor, John Wiley and Sons, New York, 1976, pps. 208-225.

[28] Turski, W., Computer Programming Methodology, Heyden, Philadelphia, 1978.

[29] Perlis, A., Sayward, F., Shaw, M., Software Metrics: An Analysis and Evaluation, MIT Press, Cambridge, MA, 1981.

[30] Davis, J. S., "Chunks: A Basis for Complexity Measurement", Symposium on the Empirical Foundations of Information and Software Science, Atlanta, Georgia, November, 1982.

[31] Wirth, Niklaus, "Stepwise Program Development", in Systematic Programming: An Introduction, Prentice Hall, Englewood Cliffs, New Jersey, 1973, pps. 125-154.

[32] Hansen, Per Brinch, The Architecture of Concurrent Programs, Prentice Hall, Englewood Cliffs, New Jersey, 1977.

[33] Coffman, E. G., Denning, P. J., Operating Systems Theory, Prentice Hall, Englewood Cliffs, New Jersey, 1973.

[34] Weinberg, V., "Strategies to Develope Designs", in Structured Analysis, Prentice-Hall, Englewood Cliffs, New Jersey, 1980, pps. 169-203.