A Queueing Network Approach to the Module Allocation Problem in Distributed Systems

by
R. M. Bryant
and
J. R. Agre

Computer Sciences Technical Report #430

May 1981

A Queueing Network Approach to the Module Allocation Problem in Distributed Systems

R. M. Bryant*
and
J. R. Agre**

ABSTRACT

Given a collection of distributed programs and the modules they use, the module allocation problem is to determine an assignment of modules to processors that minimizes the total execution cost of the programs. Standard approaches to this problem are based on solving either a network flow problem or a constrained 0-1 integer programming problem.

In this paper we discuss an alternative approach to the module allocation problem where a closed, multiclass queueing network is solved to determine the cost of a particular module allocation. The advantage of this approach is that the execution cost can be expressed in terms of performance measures of the system such as response time. An interchange heuristic is proposed as a method of searching for a good module allocation using this model and empirical evi-

---

*Ray Bryant is with the Computer Sciences Department, University of Wisconsin-Madison, 1210 West Dayton Street, Madison, Wisconsin 53706, and for this work was supported in part by NSF grant MCS-800-3341.

**Jon Agre is with the Computer Sciences Department, University of Maryland, College Park, Maryland 20742, and for this work was supported in part by AFOSR grant 78-3654.

dence for the success of the heuristic is given. The heuristic normally finds module allocations with costs within 10 percent of the optimal module allocation.

Fast, approximate queueing network solution techniques based on mean-value-analysis allow each heuristic search to be completed in a few seconds of CPU time. The computational complexity of each search is O(M K (K + N) C) where M is the number of modules, K is the number of sites in the network, N is the number of communications processors, and C is the number of distributed program types. It appears that substantial problems of this type could be solved using the methods we describe.

Key words and phrases: task allocation problem, file assignment problem, distributed computer systems, multiclass queueing network model, mean-value analysis.

CR categories: 8.1, 8.3

This paper is to be presented at the 1981 Sigmetrics Conference on Measurement and Modelling of Computer Systems, September 14-16, 1981, Las Vegas, Nevada.

A Queueing Network Approach to the
Module Allocation Problem
in Distributed Systems

1. Introduction

The problem of determining optimal file and program locations in a computer system has received considerable attention for computer networks [5,6,9,16], multicomputer systems [7,14,23] and distributed data base systems [11,15] Standard approaches to this problem are based on solving either a network flow problem [23] or a 0-1 integer programming problem. (See, for example, [7]). Both of these methods suffer from the disadvantage that they do not properly model queueing delay. Thus the optimization criterion cannot be directly related to response time or to congestion measures of the system. Instead the allocation problem is solved based on simple estimates of program execution times [14].

In this paper we show how the same data used to formulate the network flow or integer programming solutions to this problem can be used to generate a closed queueing network model of the system. This model can then be solved to obtain estimates of program execution time that include queueing and communications delay. To simplify our discussion, we consider a variant of the task allocation problem [7] where each distributed program receives service sequentially from a collection of modules at different sites in the system. We refer to this problem as the "module

allocation problem."

The model can be extended in a straightforward way to handle file assignment problems as well. Extensions to the parallel processing case appear to be possible but are not considered in this paper.

Formally stated, the module allocation problem is to determine an assignment of modules to processors that minimizes the execution cost of the distributed programs running on the system. The cost function we use in this paper is the sum of the response ratios (response time divided by requested processor time) across all distributed programs on the system. Inputs to the model include the number of distributed programs of each type, the modules used by each distributed program, module execution times per program, and estimates of the amount of data transferred between modules during the processing of each program. Modules located at the same site are assumed to have immediate access to this data; the data must be transferred over communication lines between modules that are not coresident.

An interchange heuristic is proposed as a method of finding a good module allocation. In an empirical study of 40 randomly generated test cases, the heuristic search usually found module allocations that were less than 10 percent below optimal; these allocations normally produced 40 percent improvement over a randomly chosen module allocation. Finding these approximate solutions required solving 3M (K-1) queueing networks where M is the number of modules and K

is the number of nodes in the network. Optimal solutions were determined through exhaustive search at a cost of K^M queueing network solutions.

For many reasonably sized module allocation problems, the storage requirements and computational costs of exactly solving each queueing network model during the search are prohibitive. Three approximate solution techniques [17,18,20] were investigated to decrease the resource requirements of the exact solution techniques to a manageable level. When these techniques are combined with the interchange search, the result is an algorithm capable of rapidly finding good module allocations. For example, each interchange search on a 10 program, 5 module, 5 node problem required as little as 4 seconds of cpu time on a Univac 1100/80 system. The complexity of the resulting algorithm is $O(M\ K\ (K + N)\ C)$ where M is the number of modules, K the number of sites in the network, N the number of communications servers, and C is the number of distributed program types. It appears likely that the procedures outlined in this paper should be feasible for the solution of substantial allocation problems.

<u>1.1. Relation to Previous Work</u> While most of the integer programming formulations to the file assignment problem use linear cost criterion independent of queueing delay, some researchers have used queueing delay to formulate waiting time constraints [6,15]. The formulas used in these cases

are based on assuming infinite source poisson arrivals. While our approach does not allow waiting time constraints, the finite size of the workload is properly modeled with a closed queueing network.

Closed queueing network models have been applied the file assignment problem [10,25]. The approach used in these cases was to convert the problem from a discrete to a continuous valued optimization problem and then use non-linear optimization techniques such as gradient search to find an optimal solution. The real-valued solution was then truncated to determine a placement of files on devices.

The integer programming approaches normally result in non-linear problems. These problems are then solved either by reducing the problem to a linear problem [6], by using intelligent search techniques such as "hypercube search" [16], or by using heuristic search techniques [5,15].

There is no substantive previous work in which the file assignment problem with an underlying queueing network model with multiple classes and a general topology was solved for locally optimal allocations. In the module allocation problem we assume that modules (like files in the file assignment problem) may not be subdivided so that integer programming techniques are required. A linearization of this problem does not appear feasible. This leaves only exhaustive search or intelligent search techniques to be used to find optimal solutions. For many reasonably sized problems,

these are intractable.

Heuristic search techniques with reduced resource requirements are needed. In general, the heuristic search techniques produce suboptimal solutions where the degree of approximation is unknown. By testing the heuristic scheme on problems where the optimal solution is known, confidence in the heuristic may be established. This is the approach we follow in this paper.

1.2. Organization of this Paper

In the next section of this paper we describe our hardware and software models. We then present a solution method that uses mean-value analysis (MVA) [18] to efficiently calculate the congestion measures of the system for a particular task allocation. Next we discuss the problems of searching for an optimal module allocation and describe the interchange heuristic that we have found useful for determining good allocations. Empirical evidence for the success of the heuristic is presented. Following this, the results of our experiments with the approximate solution techniques are described.

2. Model Description

Because we are interested in modeling the interplay between the hardware and software of a distributed system, we need to model both of these entities. Our model description is thus logically divided into two portions: (i) a hardware model and (ii) a software model.

2.1. Hardware Model We consider a distributed system with K "nodes" where each node consists of a processor and some communications lines. (We will use the term "server" to refer to a station in the queueing network model so that there is no confusion between "nodes" in the distributed computer system and "nodes" in the queueing network model.) Each processor is modeled by a single processor sharing (PS) server. This is reasonable if the true service discipline is round-robin among all waiting tasks and if the round-robin quantum size is small in relation to the mean service time.

For this paper, we assume that the distributed system is completely connected. That is every node in the system has a direct communications path to each other node.* We assume that each communications path is a half-duplex path that can be represented by a single server in the queueing network. The total number of communication servers in the network is therefore $K(K-1)/2$ and total number of servers J in the queueing network will be $K + K(K-1)/2$.

We will use the term communications processor (CP) to refer to the communications servers. We assume that CP's at a particular node operate independently of the CPU at the node. In particular, the CPU service rate at a node is independent of the number of messages being transferred by

* Our method also applies to the case where there is a fixed route between nodes that are not directly connected.

the CP´s at the node.

We will let CPU(i) be the index of the server that represents the CPU at node i in the queueing network and let CP(i,j) be the index of the server that represents the CP between nodes i and j. Figure 1 shows the relationship between the various servers in the hardware model.

We assume that messages are transmitted between nodes by being divided into packets of fixed size. Furthermore, we assume that if packets from several messages are waiting for service from a particular CP, then packets are transmitted in a round-robin order among all messages waiting for the CP (including those messages waiting to be sent the

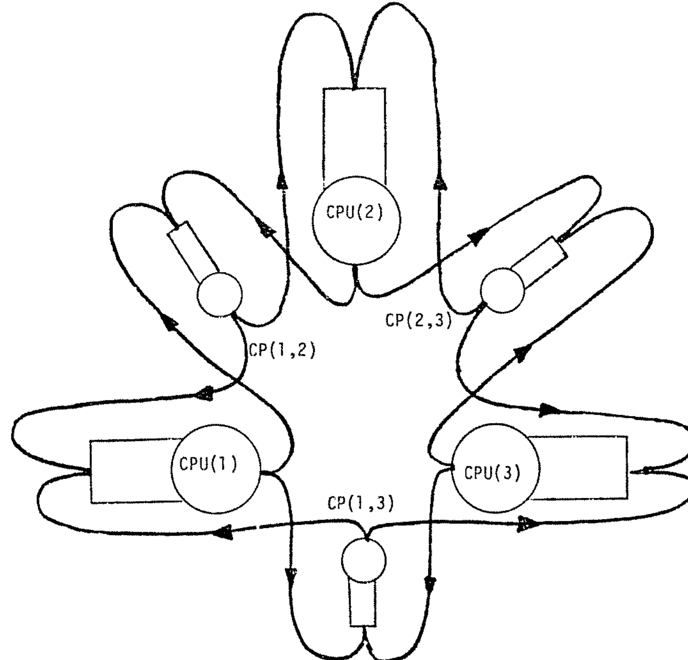

Figure 1
Servers Represented in Hardware Model

other direction). With these assumptions it seems reasonable to assume that the communications lines can be modeled by PS servers as well although this assumption is less tenable than in the CPU case. The primary justification for this assumption is that it allows the service times at the CP's to be class dependent while allowing the queueing network model to be solved efficiently [3].

2.2. Software Model We let M be the total number of software modules in the system. We assume that there is only one copy of each of the M modules.* We assume that a module is an atomic program unit and cannot be split between two nodes in the network.

We suppose that there are D distributed programs to be run on the system. We let p_i be the number of instances of program i. For each distributed program i, we let U_i be the set of modules used by the program. We assume that a program is executing only one module at any given time and that communications transfer time is not overlapped with computation time.

We will use the D by M matrix $E=\{e_{i,j}\}$ and the M by N matrix $R=\{r_{i,j}\}$ to estimate the CPU resource demands from module execution. Each entry $e_{i,j}$ of E with j in U_i gives

*Multiple copies of modules can be handled by counting modules more than once in this total. Even though there is no parallel processing within individual distributed programs, having multiple module copies can improve system throughput by allowing parallel processing across distributed program types.

the total average execution time required to execute module j during the execution of distributed program i on a CPU with instruction rate normalized to 1. Each entry $r_{i,j}$ of R gives the rate factor by which the execution of module i should be adjusted when the module is executed at node j. Thus the average service time required at node j by distributed program k for the execution of module m is given by

$$e_{i,m} / r_{m,j}.$$

We do not consider here the problems of estimating the entries in the matrix E. This matter has been considered in detail by others [7,21,22].

The matrix R represents the speed differences between CPU's on a per module basis. For simplicity, we assume that any module can be executed at any node. A module can effectively be prohibited from executing at a particular node by making the corresponding $r_{m,j}$ value small.

We use the D by M by M matrix $C=\{c_{i,j,k}\}$ and the vectors $S=\{s_i\}$ and $L=\{L_i\}$ to calculate intermodule communication costs. Entry $c_{i,j,k}$ gives the communications time required to transmit data from module j to module k during the execution of program i whenever modules j and k are not coresident and assuming a CP with speed normalized to 1. For consistency we assume $c_{i,j,k}=c_{i,k,j}$ for all j,k. Entry $s_{CP(i,j)}$ gives the speed of the CP connecting nodes i and j. L_i gives the node where module i is located in the network. Thus if j and k are modules used by program i and L_j is not the same as L_k, then the resource demand on $CP(L_i,L_j)$ caused

by execution of program i is given by:

$$c_{i,j,k} / s_{CP(L_i,L_j)}$$

The formulation we have proposed is essentially the same as that used in the integer programming or network flow approaches to the task allocation problem [7]. However, our approach provides the additional capability of estimating congestion delays in the network. In [7], the "cost" function is based on processing time cost and communications volume cost. Performance measures (such as limits on the communications bandwidth) are treated as constraints.

To illustrate how this formulation might be extended to the file allocation problem, note that if one were to identify a CPU server, k, in our model with a disk and to identify a module, m, with a file on that disk, then exactly the same formulation holds. The quantity $e_{i,m}$ can be interpreted as the total average number of disk words required from file m during the execution of program i and $r_{m,k}$ represents the transfer rate of the disk at node k. The value $c_{i,j,m}$ can represent the communication time required to transfer data from file m to module j during the execution of program i. The model can be improved by allowing both CPU and disk servers to be present at one node and by using the techniques of [1,24] to model overlap of I/O, communications and CPU processing. There appear to be no fundamental difficulties in solving this improved model.

3. Queueing Network Model Solution

Given a particular module allocation L we can use the parameters described in the last section to calculate the parameters of a queueing network model of the system. Following the approach of [1,13] who ignore the exact topology of the queueing network they are solving, we ignore the exact order in which the modules are executed. This is possible since the performance characteristics of a queueing network are known to depend only on the total per class service time at each server in the network [8,12]. In our present context, this means that only the following parameters are significant:

(1) The total mean service requirement of each distributed program at each CPU in the network, and

(2) the total mean communications time required at each communications processor in the network during the execution of the distributed program.

Since the order of module execution is immaterial, we will assume that they execute in numerical order and we will define the response time of a distributed program as the sum of the waiting times at each node that the program must visit to receive service.

Let $T=\{t_{i,j}\}$ be the service time matrix for the queueing network. That is, let $t_{i,j}$ be the mean service demand by customer class i at server j. (Because we have assumed the PS discipline at each node in the network, only the mean service times are significant, and the exact form of the

service time distribution does not matter [3]). We represent each distributed program as a distinct customer class in the queueing network. Given the module allocation vector L, $t_{i,j}$ can be calculated according to:

$$t_{i,j} = \sum_{i=1}^{D} \sum_{j \text{ in } U_i} e_{i,j} / r_{j,L_j}$$

when j represents a CPU server and

$$t_{i,j} = \sum_{i=1}^{D} \sum_{\substack{\text{all modules } j,k \text{ with} \\ j \text{ and } k \text{ in } U_i \text{ and} \\ \text{that are not coresident}}} c_{i,j,k} / s_{CP(L_j,L_k)}$$

when j represents a CP server.

Given the matrix T, one can apply any of the standard techniques [4,19] to determine performance characteristics of the queueing network. Because we were primarily interested in response time and mean queue sizes, and because the mean-value analysis (MVA) method [18] is extremely easy to program, we chose MVA as the solution method for our queueing network. MVA also has the advantage that approximations are known that extend the class of networks that can be solved well beyond the class of BCMP-type networks [1]. For details of the MVA algorithm we used, see [1].

The storage requirement of the MVA algorithm increases with the product of the number of customers in each class [4]. In our case storage requirements are proportional to

$$J \prod_{i=1}^{D} (p_i+1)$$

where J = K + K(K-1)/2 is the total number of servers in the queueing network and p_i is the number of instances of program i in the system. In general this storage requirement can be large. For example, if we assume that there is only one instance of each distributed program, then we see that the storage required to solve our queueing network model for K processors and D distributed programs requires $O(K^2 2^D)$ words of storage. Balbo and Bruell [4] mention a device for reducing the storage requirement for this case to $J\ 2^{D-1}$. Even then, this storage requirement can be high, as Table 1 shows. Note that the number of modules M does not change the storage requirement for the queueing network model solution.

Number of Programs (D)	Number of Nodes (N)	Total Number of Servers (J)	Words Required
10	5	15	7680
12	5	15	30720
14	5	15	122880
16	5	15	491520
10	10	55	28160
10	12	78	39936
10	14	105	53760

Table I
Storage Requirements
for MVA example

To solve systems with more than 15 different distributed programs would be impossible. Fortunately, good approximate solution algorithms are known that have a much smaller storage complexity [17,18,20]. These techniques should allow the solution of models with dozens of different programs and dozens of instances of the programs. An investigation of these approximation schemes is given in Section 6.

4. Search Algorithm

Since any module can conceivably run on any processor, there are a total of K^M possible values for the module location vector L. To evaluate the cost function for a particular module allocation requires the solution of a queueing network. In general this solution has computational complexity of the same order as the storage complexity given above. Thus for all but the smallest problems, exhaustive search to find an optimal solution will be out of the question. We therefore propose that an interchange search be used to attempt to find a good module allocation.

Let the cost function G(L) denote the "goodness" of a particular module allocation. In this paper we have used the sum of the response ratios (response time divided by requested processor time) of the distributed programs in the system as the goodness function G. Alternatively, for load balancing purposes, G could be calculated as the root-mean-square of the (total) mean CPU queues over all nodes in the

system. In either case, the search procedure is to attempt to determine a module allocation that, without loss of generality, we may assume minimizes the function G.

Then an interchange search to find the optimum module allocation would proceed in the following way. Pick an initial module allocation L and an order in which the modules are to be moved $O=\{o_i\}$. Solve the queueing network for this initial allocation and calculate the goodness function G. Then starting with module o_1, solve each of the N-1 queueing networks with module o_1 allocated to a node other than its current location and holding all other module allocations fixed. One of these locations has the minimal G value thus far encountered. (If more than one has the minimal G value choose one at random as being best seen so far.) Suppose the best location found for module o_1 is node j. Then set L_{o_1} to j. Now repeat the process for module o_2 and so forth until module o_M has been considered at all N nodes. We will call this process of sequentially shifting each module $o_1, \ldots, o_M$ an <u>iteration</u> of the search. Note that one iteration of the search requires the solution of M(N-1) queueing network models.

At the end of one iteration, one will normally have a much improved solution over the initial guess. However, because the module locations have changed during the search, module o_1 may not now be at its best location with respect to the current location of the other modules. Therefore another iteration of the search should be conducted. This

process is repeated until either a predetermined maximum number of iterations has been performed, or until no modules are moved during an iteration. (Our experience is that at most 3 iterations are ever required.)

Clearly this search procedure need not find the optimal module allocation. However, empirical evidence (See Section 5) suggests that even if the optimum allocation is not found, the result of the search is usually within a few percent of the optimal module allocation.

5. Empirical Validation of the Search Procedure

Since we cannot be sure that the search procedure mentioned above will ever find an optimum module allocation, we have compared the solutions found by the interchange search to optimal module allocations found through exhaustive search. To complete the exhaustive search in a reasonable amount of time, we restricted our attention to a small problem of 5 programs, 5 nodes, and 5 modules with one instance of each distributed program. This gives a total of $5^5 = 3135$ queueing networks to solve to determine an optimal module allocation.

Since we did not have access to measured data for the E and C matrices, we constructed our test cases at random according to the following rules:

(1) For each distributed program i and each module j, j was inserted in the set U_i according to the result of a Bernoulli trial with probability of success b.

(2) For each distributed program i we chose a program execution time factor PE(i) as a uniformly distributed random number between $PE_{min}(i)$ and $PE_{max}(i)$.

(3) For each module we chose a module execution time factor ME(i) as a uniformly distributed random number between $ME_{min}(i)$ and $ME_{max}(i)$.

(4) The execution matrix E was calculated according to

$$e_{i,j} = PE(i) * ME(j)$$

The communications matrix C was calculated as

$$c_{i,j,k} = exponential.f(0.10 * max(e_{i,j} , e_{i,k}));$$

where exponential.f(xbar) is a function that returns an exponential pseudo-random number with mean xbar.

Values of these parameters for the cases we have considered are given in Table 2. Parameters for the hardware portion of the model are given in Table 3. For simplicity, we have assumed that all columns of the matrix R are constant, i. e. the speed of the processor is the only factor in determining module execution times from the E matrix.

We make no claim that these parameters represent those on a real system. Our concern here is to determine the usefulness of the heuristic search procedure we have proposed.

We constructed 10 random problems using the parameters as above. To give the reader a feeling for the parameters of a typical problem we have summarized these in Tables 5 and 6. For each problem we found an optimal solution by exhaustive search and by the heuristic search algorithm

program id	PE_{min}	PE_{max}	module id	ME_{min}	ME_{max}
1	8.0	10.0	1	0.5	2.0
2	0.5	2.0	2	0.2	0.4
3	5.0	6.0	3	1.0	1.5
4	0.5	2.0	4	0.5	2.5
5	8.0	10.0	5	0.25	0.5

module usage probability (b) = 0.5

Table 2
Parameters for Generation
of Software Model

node id	execution rate	to node	from node	CP rate
1	2.0	1	1	1.0
2	1.0	2	1	1.0
3	1.0	3	1	1.0
4	1.0	4	1	1.0
5	0.67	5	1	0.67

All other CP's have rate 0.67.

Table 3
Hardware Parameters
for Sample Solution

described in the last section. To get an estimate of the success rate of the heuristic search, we tried it 10 times on each problem, with a random starting point and random choice for the module movement vector O in each trial. The goodness function was the sum of the response ratios of the distributed programs on the system.

Table 6 summarizes the results of this experiment. The table gives the average goodness value over all module allo-

Modules used by each program

Program id	Module list
1	2 3 4
2	1
3	1 2 3 4 5
4	3
5	2 3 4 5

Time required by program in
Each module it uses (E matrix)

Program	Total Time	(module, time) list
1	18.8	(2, 2.25) (3, 9.04) (4, 7.52)
2	0.93	(1, 0.93)
3	25.3	(1, 3.28) (2, 1.54) (3, 6.16) (4, 5.12) (5, 9.21)
4	3.83	(3, 3.83)
5	30.0	(2, 2.15) (3, 8.63) (4, 7.18) (5, 12.9)

Table 4
Typical Problem Parameters

Communication Costs for Program 1

modules		cost
2	3	1.39
2	4	1.74
3	4	1.15

Communication Costs for Program 5

modules		cost
2	3	1.20
2	4	0.97
2	5	1.27
3	4	2.12
3	5	2.50
4	5	1.58

Communication Costs for Program 3

modules		cost
1	2	0.14
1	3	0.27
1	4	0.92
1	5	0.71
2	3	0.53
2	4	0.67
2	5	1.37
3	4	0.06
3	5	0.62
4	5	0.13

Table 5
Module Communication Cost Matrix

Problem Number	Exhaustive Search Average Goodness	Exhaustive Search Best Goodness	Heuristic Search Found Same	Found Equivalent	Found Worse	Average % Worse	Worst % Worse
1	12.7	6.75	3	5	2	28.8	28.8
2	12.4	7.21	0	0	10	5.4	7.1
3	15.3	7.74	4	4	2	45.6	45.6
4	12.9	7.36	1	3	6	8.3	13.2
5	12.8	7.30	0	4	6	3.2	3.2
6	13.7	8.07	1	2	7	3.8	4.8
7	12.8	6.86	2	1	7	15.5	21.7
8	15.6	8.38	4	6	0		
9	12.1	7.06	0	0	10	2.9	6.5
10	12.3	6.67	2	3	5	14.7	17.7

Table 6
Summary of Exhaustive versus Heuristic Search

Note: Average % Worse only includes the "Found Worse" Cases

cations, the best goodness function value found by the exhaustive search, the number of times (out of 10 trials) when the heuristic search found exactly the same module allocation as the exhaustive search, the number of times that the heuristic search found an equivalent module allocation (i. e. a different module allocation with the optimal goodness function value), and the number of times that the heuristic search failed to find as good of module allocation as the exhaustive search. The last two columns give the percentage difference between the two solutions in the cases that the heuristic search failed.

In all but three of the cases, the heuristic found the same allocation as the exhaustive search case did at least

once in 10 trials. Of the total of 100 heuristic case trials, 45 found either the same solution as the exhaustive search or another solution with an identical goodness function value. Except for Problems 1 and 3 (where only two of the heuristic searches failed to find a value with the same goodness as the global optimum), the maximum percent difference between the optimum allocation and the one that the heuristic search found was less than 22%. The average difference between the optimal allocation and the heuristic search result was 5.1% when averaged over all 100 trials. In general, the heuristic search found a solution 40% better than the average G value. Considering that the exhaustive search took about 6 minutes of CPU time (times are for a Univac 1180 system) while each heuristic search took about 6 seconds of CPU time, the heuristic search appears to have been very successful. In all cases the heuristic search converged in at most 3 iterations, for an overall cost of 60 (=3M(K-1)) queueing network solutions.

We have conducted 2 more experiments like the one described above, with slightly different parameters. Of the 300 heuristic searches in 30 problems that we have performed, 113 found solutions as good as the optimal solution. The average difference between the optimal G value and the value found by the heuristic search was less than 10 percent when averaged over those cases when the solutions were not equivalent. In the 300 trials, the heuristic search seriously failed (i. e. found a solution worse than the optimal

solution by more than 20%) only 4 times.

For problems with more modules, programs, or nodes, the exhaustive search process becomes prohibitively expensive. For the case of D distinct distributed programs with one instance of each program, the computational cost of solving each queueing network is $O(K^2\ 2^D)$. Complexity of exhaustive search is therefore $O(K^M\ K^2\ 2^D)$. For the heuristic search this becomes $O(M\ K^3\ 2^D)$.

We have run our heuristic search on problems as large as 10 programs, 5 nodes and 5 modules. Each heuristic search on this problem took about 6 minutes of CPU time. Our estimates are that one exhaustive search in this case would require about 6 hours of Univac 1100/80 time. To overcome these complexity limits we have used approximate MVA algorithms.

6. Approximate Solution Techniques

Recently Schweitzer [20] (see also Appendix B of[2]), Reiser and Lavenberg [18], and Chandy and Neuse [17] have proposed fast, approximate MVA solution techniques that circumvent the exponential storage and computational growth problems of the direct MVA algorithm. In this section we explore the use of these methods in our module allocation problem.

The speed and accuracy of these methods can be very impressive. For example, when applied to the 8 node, 4 class problem of [18] Schweitzer's algorithm (the fastest

algorithm of the group) runs more than 100 times faster than the direct MVA method. In relation to this method, the other algorithms run 7 (Reiser and Lavenberg), 11 (Chandy and Neuse), and 110 times as long. These proportions depend not only on the number of jobs in the network but also on the characteristics of the network itself. For example, if one of the servers is a bottleneck, Schweitzer's algorithm converges faster and can be as much as 240 times faster than the the direct method! While Schweitzer's algorithm is the fastest, Chandy and Neuse's "MVA plus linearizer" [17] method is by far the most accurate, with relative errors in queue lengths and mean queue times being less than one percent in all cases. Relative errors in the other two methods were usually in the range of 4 to 10 percent.

To compare the accuracy and speed of these methods when applied to the module allocation problem, we randomly generated 100 module allocation problems according to the rules given in Section 5 and used all three methods as well as direct MVA to solve the resulting queueing networks. The results of this experiment are summarized in Table 7. Each of the module allocation problems solved was for a 10 program, 5 node, 5 module case, with one instance of each distributed program. We see that the execution times of the four methods are approximately in the ratio 1:3:30:74. This means that the exhaustive search problems of Section 5 that would have required 6 hours of CPU time using the direct MVA algorithm could be done in about 5 minutes using

Schweitzer's algorithm.

As the number of instances of each distributed program is increased the computational cost of the direct MVA algorithm increases exponentially as discussed in Section 3. Schweitzer's algorithm has computational complexity O(J C) where C is the number of customer classes (distributed program types). Execution time is essentially independent of customer population. Reiser and Lavenberg's algorithm has

Method	Average Execution Time per Queueing Network Solution (seconds)	Maximum Execution Time
direct MVA	6.5	6.5
Schweitzer	.088	.140
Reiser and Lavenberg	.240	.380
Chandy and Neuse	2.4	3.3

Table 7(a)
Execution Time Comparison
(Times are for Univac 1100/80)

Method	Relative Errors (Percent): Response Ratios		Goodness Function	
	Average Error	Maximum Error	Average Error	Maximum Error
Schweitzer	4.7	36.	3.9	9.1
Reiser and Lavenberg	5.4	41.	4.8	9.5
Chandy and Neuse	0.00+	1.7	0.00+	0.33

Table 7(b)
Accuracy Comparison

Table 7

computational complexity O(J P) where J is the number of servers and P is the sum of the number of customers in each class. Storage complexity of both of the methods is O(J C). Since Schweitzer's algorithm has the same accuracy as the Reiser and Lavenberg method and better performance than the latter, we decided not to use the Reiser and Lavenberg method. Chandy and Neuse's method has storage complexity $O(J\ C^2)$. The computational cost of this scheme is more difficult to quantify, but the Chandy and Neuse algorithm calls a version of Schweitzer's algorithm 2(C+1)+1 times when solving a network with C classes. Overall execution cost is actually more like 2.5(C+1) times that of Schweitzer's algorithm due to the "linearizer" calculation which requires $O(J\ C^2)$ operations. Like Schweitzer's algorithm the cost is independent of the network population.

For our application, the accuracy of the Chandy and Neuse algorithm is less important than the speed of the Schweitzer algorithm. We therefore decided to use the latter. Combining the interchange search with the Schweitzer's MVA algorithm results in an approximate solution to the module allocation problem which has complexity $O(M\ K^3\ C)$. Typical execution times on a Univac 1100/80 system were of the order 4 to 6 seconds for a 10 program, 5 node, 5 module problem depending on whether 2 or 3 iterations of the heuristic search were required. Thus it should be possible to solve a problem with 50 programs, 10 nodes and 80 modules in about 30 minutes of CPU time.

This estimate assumes that the network is completely connected. If there are fewer than $O(K^2)$ CP's then the estimate above should be replaced by O(M K (K + N) C), where N is the number of communications processors.

Of course, there is no guarantee that a module allocation found using the approximate algorithm will be the same as one found using the exact MVA algorithm. To estimate how different the allocations might be using the two queueing network solution techniques, we compared optimal module allocations determined using the direct MVA method to optimal module allocations found using the Schweitzer's algorithm. In ten exhaustive search trials on a 5 program, 5 node, 5 module problem, the optimal module allocations using the exact and approximate MVA algorithms were found to be identical in 8 out of 10 cases. In the 2 cases where the optimal allocations were not identical, they differed in the placement of only two modules. While this is not a definitive comparison, it does appear that the error in the approximate method does not grossly change the optimal module allocation.

As a final comparison we repeated the experiment of Section 5 for the 10 program, 5 node, 5 module case. This time we used $PE_{min}(i)=1.0$ and $PE_{max}(i)=100.0$ to generate a wider variety of problems. The results of 10 trials of this experiment are given in Table 8.

	Exhaustive Search		Heuristic Search				
Problem Number	Average Goodness	Best Goodness	Found Same	Found Equivalent	Found Worse	Average % Worse	Worst % Worse
1	83.6	31.7	0	0	10	1.7	3.7
2	97.3	32.4	3	0	7	1.9	13.3
3	83.6	31.9	0	0	10	5.5	6.2
4	78.2	33.3	1	0	9	4.1	9.9
5	88.8	36.3	2	0	8	1.6	6.4
6	85.1	32.2	0	0	10	2.6	3.9
7	83.7	30.6	1	0	9	4.2	5.3
8	79.2	34.3	0	0	10	3.6	17.6
9	83.2	34.0	2	2	6	2.8	5.5
10	88.4	32.0	1	0	9	1.8	5.2

Table 8
Summary of Exhaustive versus Heuristic Search
Using Schweitzer's MVA Solution Algorithm
(10 program, 5 node, 5 module case)

7. Concluding Remarks

We have discussed an alternative approach to the module allocation problem in distributed systems which allows representation of congestion components such as mean queue size or response time in the criterion function. The data required to perform this analysis is essentially of the same form as that used in standard approaches to this problem based on network flow algorithms or 0-1 integer programming. The construction of a queueing model from the data was described and shown to be efficiently solvable. We have suggested a straightforward heuristic search as a method of finding a good module allocation and provided empirical evidence of the success of this heuristic.

One advantage of the heuristic search is that problem constraints such as limited main memory size can easily be handled. The search is merely not allowed to enter a state which the constraints would prohibit. The introduction of such constraints could however increase the number of trials necessary to find a good solution.

Use of exact queueing network solution methods is limited by the exponential growth of storage and computational costs as the number of distributed program types increases. The use of an approximate MVA solution algorithm due to Schweitzer [20] when combined with the interchange search was shown to yield good solutions to the module allocation problem at a computational cost of $O(M\ K\ (K + N)\ C)$ where M is the number of modules, K the number of sites in the network, N the number of communication servers, and C the number of distributed program types. It appears that this technique should be usable on substantial sized module allocation problems.

Our present model assumes that computation on different CPU's is not overlapped nor is communication overlapped with computation. Analytic methods have been proposed to handle limited forms of overlap [1,24]. Whether these methods can be generalized to handle the high degree of overlap possible in a distributed system remains to be seen. Modeling of overlap is especially necessary if this problem is to be extended to handle the file allocation problem as well.

Further research needs to be done on the search technique. By allowing a module to be split between nodes it should be possible to use non-linear optimization techniques to search for a good module allocation. The resulting real valued module location vector L would then have to be truncated to an integer solution, much as is done in [10,25].

8. Acknowledgements

Computer time for this work was provided in part by the computing center at the University of Wisconsin-Madison. The authors would also like to thank Larry Dowdy whose suggestions have resulted in an improved paper.

REFERENCES

[1] Bard, Y., "Some Extensions to Multiclass Queueing Network Analysis," pp. 51-62 in Performance of Computer Systems, ed. M. Arato, A. Butrimenko, and E. Gelenbe, North-Holland (1979).

[2] Bard, Y., "A Model of Shared DASD and Multipathing," Communications of the ACM 23, 10, pp. 564-583 (October 1980).

[3] Baskett, F., K. M. Chandy, R. R. Muntz, and F. G. Palacios, "Open, Closed, and Mixed Networks of Queues with Different Classes of Customers," Journal of the ACM 22, 2, pp. 248-260 (1975).

[4] Bruell, S. C. and G. Balbo, Computational Algorithms for Closed Queueing Networks, Elsevier North-Holland, Inc., New York (1980).

[5] Chandy, K. M. and J. E. Hewes, "File Allocation in Distributed Systems," pp. 10-13 in Computer Performance 1976, ed. P. P. S. Chen and M. A. Franklin, (March 29-31, 1976). Proceedings of the International Symposium on Computer Performance Modeling, Measurement, and Evaluation, Cambridge, Massachusetts.

[6] Chu, W. W., "Optimal File Allocation in a Computer

Network," pp. 82-95 in Computer Communications Networks, ed. N. Abramson and F. F. Kuo, Prentice-Hall (1972).

[7] Chu, W. W., L. J. Holloway, M.-T. Lan, and K. Efe, "Task Allocation in Distributed Data Processing," IEEE Computer, pp. 57-69 (November 1980).

[8] Denning, P. J. and J. P. Buzen, "The Operational Analysis of Queueing Network Models," Computing Surveys 10, 3, pp. 225-262 (September 1978).

[9] Dowdy, L. W. and D. V. Foster, Comparative Models of the File Assignment Problem, Submitted for publication.

[10] Dowdy, L. W., "Optimal Branching Probabilities and Their Relationship to Computer Network File Distribution," Ph.D Dissertation, Department of Computer Science, Duke University (1977).

[11] Fisher, M. L. and D. S. Hochbaum, "Database Location in Computer Networks," Journal of the ACM 27, 4, pp. 718-735 (October 1980).

[12] Giammo, T. P., "Extensions to Exponential Queueing Network Theory for Use in a Planning Environment," Proceedings of the IEEE Fall ´76 Compcon Conference, (1976).

[13] Kobayashi, H. and M. Reiser, "On Generalization of Job Routing Behavior in a Queueing Network Model," Research Report RC-5679, IBM T. J. Watson Research Laboratory, Yorktown Heights, N. Y. (1975).

[14] Lint, B. and T. Agerwala, "Communication Issues in the Design and Analysis of Parallel Algorithms," IEEE Transactions on Software Engineering SE-7, 2, pp. 174-188 (March 1981).

[15] Mahmoud, S. and J. S. Riordon, "Optimal Allocation of Resources in Distributed Information Networks," ACM Trans. on Database Systems 1, 1, pp. 66-78 (March 1976).

[16] Morgan, H. L. and K. D. Levin, "Optimal Program and Data Locations in Computer Networks," Communications of the ACM 20, 5, pp. 315-321 (May 1977).

[17] Neuse, D. and K. M. Chandy, "A Heuristic Algorithm for Queueing Network Models of Computing Systems," Department of Computer Sciences Technical Report #TR-CHAN-81-02, University of Texas, Austin, Texas (March

1981).

[18] Reiser, M. and S. S. Lavenberg, "Mean-Value Analysis of Closed Multichain Queueing Networks," Journal of the ACM 27, 2, pp. 313-322 (April 1980).

[19] Sauer, C. H. and K. M. Chandy, Computer Systems Performance Modeling, Prentice-Hall (1981).

[20] Schweitzer, P., "Approximate Analysis of Multiclass Closed Networks of Queues," International Conference on Stochastic Control and Optimization, (1979).

[21] Smith, C. U. and J. C. Browne, "Modeling Software Systems for Performance Predictions," Proceedings of the Computer Measurement Group X, (December 1979).

[22] Smith, C. U. and J. C. Browne, "Performance Specifications and Analysis of Software Designs," Proceedings of the 1979 Sigmetrics Conference on Simulation, Measurement and Modeling of Computer Systems, pp. 173-182 (August 1979).

[23] Stone, H. S., "Multiprocessor Scheduling with the aid of Network Flow Analysis," IEEE Transactions on Software Engineering SE-3, 5, pp. 315-321 (May 1977).

[24] Towsley, D., K. M. Chandy, and J. C. Browne, "Models for Parallel Processing Within Programs: Application to CPU:I/O and I/O:I/O Overlap," Communications of the ACM 21, 10, pp. 821-831 (October 1978).

[25] Trivedi, K. S., R. A. Wagner, and T. M. Sigmon, "Optimal Selection of CPU Speed, Device Capacities, and File Assignments," Journal of th ACM 27, 3, pp. 457-473 (July 1980).