Chapter 1
Introduction to Scheduling and Load Balancing

Advances in hardware and software technologies have led to increased interest in the use of large-scale parallel and distributed systems for database, real-time, defense, and large-scale commercial applications. The operating system and management of the concurrent processes constitute integral parts of the parallel and distributed environments. One of the biggest issues in such systems is the development of effective techniques for the distribution of the processes of a parallel program on multiple processors. The problem is how to distribute (or schedule) the processes among processing elements to achieve some performance goal(s), such as minimizing execution time, minimizing communication delays, and/or maximizing resource utilization [Casavant 1988, Polychronopoulos 1987]. From a system's point of view, this distribution choice becomes a resource management problem and should be considered an important factor during the design phases of multiprocessor systems.

Process scheduling methods are typically classified into several subcategories as depicted in Figure 1.1 [Casavant 1988]. (It should be noted that throughout this manuscript the terms "job," "process," and "task" are used interchangeably.) Local scheduling performed by the operating system of a processor consists of the assignment of processes to the time-slices of the processor. Global scheduling, on the other hand, is the process of deciding where to execute a process in a multiprocessor system. Global scheduling may be carried out by a single central authority, or it may be distributed among the processing elements. In this tutorial, our focus will be on global scheduling methods, which are classified into two major groups: static scheduling and dynamic scheduling (often referred to as dynamic load balancing).

1.1 Static scheduling

In static scheduling, the assignment of tasks to processors is done before program execution begins. Information regarding task execution times and processing resources is assumed to be known at compile time. A task is always executed on the processor to which it is assigned; that is, static scheduling methods are processor nonpreemptive. Typically, the goal of static scheduling methods is to minimize the overall execution time of a concurrent program while minimizing the communication delays. With this goal in mind, static scheduling methods [Lo 1988, Sarkar 1986, Shirazi 1990, Stone 1977] attempt to:

- predict the program execution behavior at compile time (that is, estimate the process or task, execution times, and communication delays);
- perform a partitioning of smaller tasks into coarser-grain processes in an attempt to reduce the communication costs; and,
- allocate processes to processors.

The major advantage of static scheduling methods is that all the overhead of the scheduling process is incurred at compile time, resulting in a more efficient execution time environment compared to dynamic scheduling methods. However, static scheduling suffers from many disadvantages, as discussed shortly.

Static scheduling methods can be classified into optimal and suboptimal. Perhaps one of the most critical shortcomings of static scheduling is that, in general, generating optimal schedules is an NP-complete problem. It is only possible to generate optimal solutions in restricted cases (for example, when the execution time of all of the tasks is the same and only two processors are used). NP-completeness of optimal static scheduling, with or without communication cost considerations, has been proven in the literature [Chretienne 1989, Papadimitriou 1990, Sarkar 1989]. Here, we give a simple example to demonstrate the difficulties in attaining general optimal schedules.

Assume we have *n* processes, with different execution times, which are to be scheduled on two processing elements, PE_1 and PE_2. Since the goal of a scheduling method is to minimize the completion time of a set of processes, the scheduler must decide which process should be assigned to (or scheduled on) which PE so

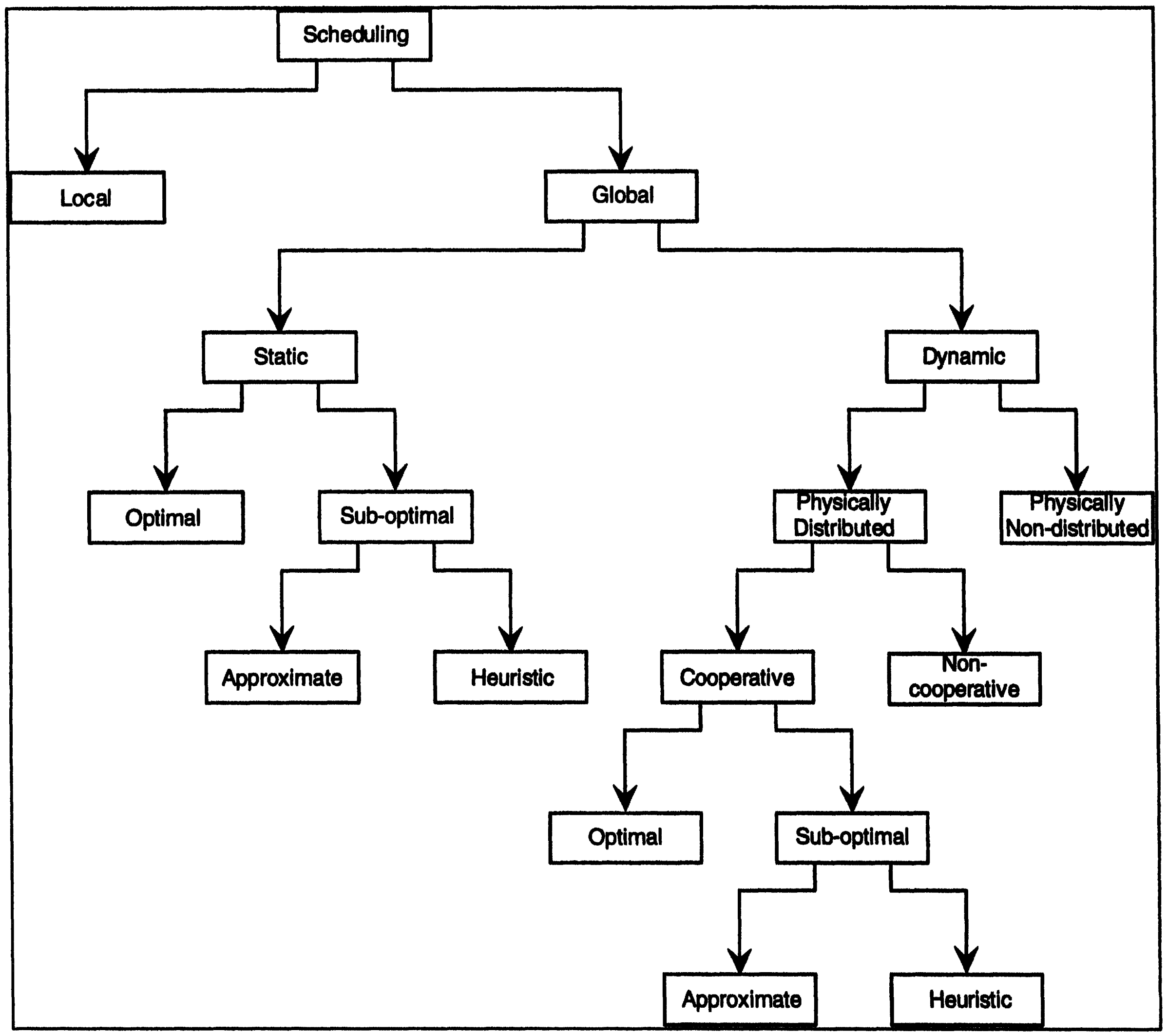


Figure 1.1. Classification of scheduling methods [Casavant 1988].

that the overall completion time is minimized. In this case, the optimum schedule will ensure that the processing loads assigned PE_1 and PE_2 are equal (with no unnecessary delay periods). However, this problem is NP-complete since it can be easily mapped to the well-known, NP-complete "set-partitioning" problem. The set-partitioning problem is as follows:

Let Z be the set of integers. Given a finite set A and a "size" $s(a) \in Z$ for each $a \in A$, find a subset of A (that is , A') such that

$$\sum_{a \in A'} s(a) = \sum_{b \in (A-A')} s(b).$$

Because reaching optimal static schedules is an NP-complete problem, most of the research and development in this area has been focused on suboptimal solutions. These methods are classified into approximate and heuristic approaches. In approximate suboptimal static scheduling methods, the solution space is searched in either a depth-first or a breadth-first fashion. However, instead of searching the entire solution space for an optimal solution, the algorithm stops when a "good" (or acceptable) solution is reached.

Heuristic methods, as the name indicates, rely on rules-of-thumb to guide the scheduling process in the right direction to reach a "near" optimal solution. For example, the length of a critical path for a task is defined as the length of one of several possible longest paths from that task, through several intermediate and dependent tasks, to the end of the program. No concurrent program can complete its execution in a time period less than the length of its critical path. A heuristic scheduling method may take advantage of this fact by giving a higher priority in the scheduling of tasks with longer critical path lengths. The idea is that by scheduling the tasks on the critical path first, we have an opportunity to schedule other tasks around them, thus avoiding the lengthening of the critical path.

It should be noted that there is no universally accepted figure or standard for defining a "good" solution or a degree of "nearness" to an optimal solution. The researchers often use a loose lower-bound on the execution time of a concurrent program (for example, the length of a critical path), and show that their method can always achieve schedules with execution times within a factor of this lower-bound.

In addition to the NP-completeness of optimal general scheduling algorithms, static scheduling suffers from a wide range of problems, most notable of which are the following:

- The insufficiency of efficient and accurate methods for estimating task execution times and communication delays can cause unpredictable performance degradations [Lee 1991, Wang 1991]. The compile-time estimation of the execution time of a program's tasks (or functions) is often difficult to find due to conditional and loop constructs, whose condition values or iteration counts are unknown before execution. Estimating communication delays at compile time is not practical because of the run-time network contention delays.
- Existing task/function scheduling methods often ignore the data distribution issue. This omission causes performance degradations due to run-time communication delays for accessing data at remote sites.
- Finally, static scheduling schemes should be augmented with a tool to provide a performance profile of the predicted execution of the scheduled program on a given architecture. The user can then utilize this tool to improve the schedule or to experiment with a different architecture.

The above shortcomings, as well as the existing and possible solutions to them, will be discussed in detail in the introduction section of the relevant chapters throughout this book.

1.2 Prognosis and future directions in static scheduling

Until now the focus of research in this area has been on the efficiency of scheduling algorithms. By "efficiency" we refer to both the efficiency of the scheduling algorithm itself and the efficiency of the schedule it generates. Based on our experience and evaluations [Shirazi 1990], our contention is that most of the existing heuristic-based static scheduling algorithms perform comparably. Thus, the likelihood is low of finding a scheduling method that can achieve orders-of-magnitude (or even significant) improvements in performance over the existing methods. As a result, we believe that, while this research direction must continue, efforts should be focused on the practical applications of the developed static scheduling methods to the existing parallel systems. This recommendation implies that we may need a whole new set of tools to support the practical application of theoretic scheduling algorithms.

One such tool is a directed acyclic graph (DAG) generator. The input to a DAG generator will be a parallel program, written in the user's choice of language, and its output will be the DAG equivalent of the program (with proper functional dependencies, and execution-time and communication-delay estimations). A more general question in this regard is, if we cannot estimate the execution times and communication delays accurately, can we still get performance via static scheduling? Preliminary results with PYRROS [Yang 1992] show that as long as the task graph is coarse-grain and we use asynchronous communication, we can get good performance. Another supporting tool for static scheduling is a *performance profiler*. The function of a performance profiler tool is to read in the schedule of a parallel program on a given architecture and produce a graphical profile of the expected performance of the input program. The tool should also provide an *ideal parallelism profile* of the application, giving the user an idea of the inherent parallelism in the program.

1.3 Dynamic scheduling

Dynamic scheduling is based on the redistribution of processes among the processors during execution time. This redistribution is performed by transferring tasks from the heavily loaded processors to the lightly loaded processors (called load balancing) with the aim of improving the performance of the application [Eager 1986, Lin 1987, Shivaratri 1992, Wang 1985]. A typical load balancing algorithm is defined by three inherent policies:

- information policy, which specifies the amount of load information made available to job placement decision-makers;
- transfer policy, which determines the conditions under which a job should be transferred, that is, the current load of the host and the size of the job under consideration (the transfer policy may or may not include task migration, that is, suspending an executing task and transferring it to another processor to resume its execution); and
- placement policy, which identifies the processing element to which a job should be transferred.

The load balancing operations may be centralized in a single processor or distributed among all the processing elements that participate in the load balancing process. Many combined policies may also exist. For example, the information policy may be centralized but the transfer and placement policies may be distributed. In that case, all processors send their load information to a central processor and receive system load information from that processor. However, the decisions regarding when and where a job should be transferred are made locally by each processor. If a distributed information policy is employed, each processing element keeps its own local image of the system load. This cooperative policy is often achieved by a gradient distribution of load information among the processing elements [Lin 1987]. Each processor passes its current load information to its neighbors at preset time intervals, resulting in the dispersement of load information among all the processing elements in a short period of time. A distributed-information policy can also be noncooperative. Random scheduling is an example of noncooperative scheduling, in which a heavily loaded processor randomly chooses another processor to which to transfer a job. Random load balancing works rather well when the loads of all the processors are relatively high, that is, when it does not make much difference where a job is executed.

The advantage of dynamic load balancing over static scheduling is that the system need not be aware of the run-time behavior of the applications before execution. The flexibility inherent in dynamic load balancing allows for adaptation to the unforeseen application requirements at run-time. Dynamic load balancing is particularly useful in a system consisting of a network of workstations in which the primary performance goal is maximizing utilization of the processing power instead of minimizing execution time of the applications. The major disadvantage of dynamic load balancing schemes is the run-time overhead due to:

- the load information transfer among processors,
- the decision-making process for the selection of processes and processors for job transfers, and
- the communication delays due to task relocation itself.

1.4 Prognosis and future directions in dynamic load balancing

Up until now research and development in the dynamic load balancing area have been focused on the identification and evaluation of efficient policies for information distribution, job transfers, and placement decision-making. In the future, a greater exchange of information among programmer, compiler, and operating system is needed. In particular, we feel the emphasis will be on the development of efficient policies for load information distribution and placement decision-making, because these two areas cause most of the overhead of dynamic load balancing. Although dynamic load balancing incurs overhead due to task transfer operations, a task transfer will not take place unless the benefits of the relocation outweigh its overhead. Thus, it is our contention that in the future research and development in this area will emphasize:

- hybrid dynamic/static scheduling;
- effective load index measures;
- hierarchical system organizations with local load information distribution and local load balancing policies; and
- incorporation of a set of primitive tools at the distributed operating system level, used to implement different load balancing policies depending on the system architecture and application requirements.

1.5 Chapter organization and overview

The three papers in this chapter provide a general overview of the fields of static scheduling and dynamic load balancing. The first paper, by Casavant and Kuhl, presents a comprehensive taxonomy of scheduling in general-purpose concurrent systems. The proposed taxonomy provides the common terminology and classification mechanism necessary in addressing the scheduling problem in concurrent systems. The second paper, by Stone, is included for its historical significance in that it was one of the first papers to address the issue of multiprocessor scheduling. The paper makes use of the modified Ford-Fulkerson algorithm for finding maximum flows in commodity networks to assign a program's modules in a multiprocessor system. The solutions are discussed to the two-processor problem and its extensions to three- and n-processor problems. The last paper by Shivaratri, Krueger, and Singhal, focuses on the problem of the redistribution of the load of the system among its processors so that overall performance is maximized. The paper provides a good introduction and overview of dynamic scheduling, since it presents the motivations and design trade-offs for several load distribution algorithms; several load balancing algorithms are compared and their performances evaluated.

Bibliography

Bokhari, S. H., *Assignment Problems in Parallel and Distributed Computing,* Kluwer Academic Publishers, Norwell, Mass., 1987.

Bokhari, S. H., "Partitioning Problems in Parallel, Pipelined, and Distributed Computing," *IEEE Trans. Computers*, Vol. C-37, No. 1, Jan. 1988, pp. 48–57.

Casavant, T. L., and J. G. Kuhl, "A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems," *IEEE Trans. Software Eng.*, Vol. 14, No. 2, Feb. 1988, pp. 141–154; reprinted here.

Chretienne, P., "Task Scheduling Over Distributed Memory Machines," *Proc. Int'l Workshop Parallel and Distributed Algorithms*, North Holland Publishers, Amsterdam, 1989.

Eager, D. L., E. D. Lazowska, and J. Zahorjan, "Adaptive Load Sharing in Homogeneous Distributed Systems," *IEEE Trans. Software Eng.*, Vol. SE-12, No. 5, May 1986, pp. 662–675.

Goscinski, A., *Distributed Operating Systems,* Addison-Wesley, Reading, Mass., 1992.

Hummel, S. F., E. Schonberg, and L. E. Flynn, "Factoring: A Method for Scheduling Parallel Loops," *Comm. ACM*, Vol. 35, No. 8, Aug. 1992, pp. 90–101.

Lee, B., A. R. Hurson, and T.-Y. Feng, "A Vertically Layered Allocation Scheme for Data Flow Systems," *J. Parallel and Distributed Computing*, Vol. 11, No. 3, 1991, pp. 175–187.

Lewis, T. G., and H. El-Rewini, *Introduction to Parallel Computing*, Prentice Hall, Englewood Cliffs, N.J., 1992.

Lin, F. C. H., and R. M. Keller, "The Gradient Model Load Balancing Method," *IEEE Trans. Software Eng.*, Vol. SE-13, No. 1, Jan. 1987, pp. 32–38.

Lo, V. M., "Heuristic Algorithms for Task Assignment in Distributed Systems," *IEEE Trans. Computers*, Vol. C-37, No. 11, Nov. 1988, pp. 1384–1397.

Papadimitriou, C., and M. Yannakakis, "Towards an Architecture-Independent Analysis of Parallel Algorithms," *SIAM J. Comput.*, Vol. 19, 1990, pp. 322–328.

Polychronopoulos, C. D., and D. J. Kuck, "Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers," *IEEE Trans. Computers*, Vol. C-36, No. 12, Dec. 1987, pp. 1425–1439.

Polychronopoulos, C. D., *Parallel Programming and Compilers,* Kluwer Academic Publishers, Norwell, Mass., 1988.

Sarkar, V., and J. Hennessy, "Compile-Time Partitioning and Scheduling of Parallel Programs," *Symp. Compiler Construction*, ACM Press, New York, N.Y., 1986, pp. 17–26.

Sarkar, V., *Partitioning and Scheduling Parallel Programs for Multiprocessors,* MIT Press, Cambridge, Mass., 1989.

Shirazi, B., M. Wang, and G. Pathak, "Analysis and Evaluation of Heuristic Methods for Static Task Scheduling," *J. Parallel and Distributed Computing*, Vol. 10, 1990, pp. 222–232.

Shirazi, B., and A. R. Hurson, eds., *Proc. Hawaii Intl Conf. Systems Sciences*, Special Software Track on "Scheduling and Load Balancing," Vol. 2, IEEE CS Press, Los Alamitos, Calif., 1993, pp. 484–486.

Shirazi, B., and A. R. Hurson,, eds., *J. Parallel and Distributed Computing*, Special Issue on "Scheduling and Load Balancing," Vol. 16, No. 4, Dec. 1992.

Shivaratri, N. G., P. Kreuger, and M. Singhal, "Load Distributing for Locally Distributed Systems," *Computer*, Vol. 25, No. 12, Dec. 1992, pp. 33–44; reprinted here.

Stone, H. S.,"Multiprocessor Scheduling with the Aid of Network Flow Algorithms," *IEEE Trans. Software Eng.*, Vol. SE-3, No. 1, Jan. 1977, pp. 85–93; reprinted here.

Wang, M., et al., "Accurate Communication Cost Estimation in Static Task Scheduling," *Proc. 24th Ann. Hawaii Int'l Conf. System Sciences*, Vol. I, IEEE CS Press, Los Alamitos, Calif., 1991, pp. 10–16.

Wang, Y.-T., and R. J. T. Morris, "Load Sharing in Distributed Systems," *IEEE Trans. Computers*, Vol. C-34, No. 3, Mar. 1985, pp. 204–217.

Xu, J., and K. Hwang, "Dynamic Load Balancing for Parallel Program Execution on a Message-Passing Multicomputer," *Proc. 2nd IEEE Symp. Parallel and Distributed Processing*, IEEE CS Press, Los Alamitos, Calif., 1990, pp. 402–406.

Yang, T., and A. Gerasoulis, "PYRROS: Static Task Scheduling and Code Generation for Message Passing Multiprocessors," *Proc. 6th ACM Int'l Conf. Supercomputing* (ICS92), ACM Press, New York, N.Y., 1992, pp. 428–437.

A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems

THOMAS L. CASAVANT, MEMBER, IEEE, AND JON G. KUHL, MEMBER, IEEE

Abstract—**One measure of usefulness of a general-purpose distributed computing system is the system's ability to provide a level of performance commensurate to the degree of multiplicity of resources present in the system. Many different approaches and metrics of performance have been proposed in an attempt to achieve this goal in existing systems. In addition, analogous problem formulations exist in other fields such as control theory, operations research, and production management. However, due to the wide variety of approaches to this problem, it is difficult to meaningfully compare different systems since there is no uniform means for qualitatively or quantitatively evaluating them. It is difficult to successfully build upon existing work or identify areas worthy of additional effort without some understanding of the relationships between past efforts. In this paper, a taxonomy of approaches to the resource management problem is presented in an attempt to provide a common terminology and classification mechanism necessary in addressing this problem. The taxonomy, while presented and discussed in terms of *distributed scheduling*, is also applicable to most types of resource management. As an illustration of the usefulness of the taxonomy an annotated bibliography is given which classifies a large number of distributed scheduling approaches according to the taxonomy.**

Index Terms—**Distributed operating systems, distributed resource management, general-purpose distributed computing systems, scheduling, task allocation, taxonomy.**

I. Introduction

THE study of distributed computing has grown to include a large range of applications [16], [17], [31], [32], [37], [54], [55]. However, at the core of all the efforts to exploit the potential power of distributed computation are issues related to the management and allocation of system resources relative to the computational load of the system. This is particularly true of attempts to construct large *general-purpose* multiprocessors [3], [8], [25], [26], [44]-[46], [50], [61], [67].

The notion that a loosely coupled collection of processors could function as a more powerful general-purpose computing facility has existed for quite some time. A large body of work has focused on the problem of managing the resources of a system in such a way as to effectively exploit this power. The result of this effort has been the proposal of a variety of widely differing techniques and methodologies for distributed resource management. Along with these competing proposals has come the inevitable proliferation of inconsistent and even contradictory terminology, as well as a number of slightly differing problem formulations, assumptions, etc. Thus, it is difficult to analyze the relative merits of alternative schemes in a meaningful fashion. It is also difficult to focus common effort on approaches and areas of study which seem most likely to prove fruitful.

This paper attempts to tie the area of distributed scheduling together under a common, uniform set of terminology. In addition, a taxonomy is given which allows the classification of distributed scheduling algorithms according to a reasonably small set of salient features. This allows a convenient means of quickly describing the central aspects of a particular approach, as well as a basis for comparison of commonly classified schemes.

Earlier work has attempted to classify certain aspects of the scheduling problem. In [9], Casey gives the basis of a hierarchical categorization. The taxonomy presented here agrees with the nature of Casey's categorization. However, a large number of additional fundamental distinguishing features are included which differentiate between existing approaches. Hence, the taxonomy given here provides a more detailed and complete look at the basic issues addressed in that work. Such detail is deemed necessary to allow meaningful comparisons of different approaches. In contrast to the taxonomy of Casey, Wang [65] provides a taxonomy of load-sharing schemes. Wang's taxonomy succinctly describes the range of approaches to the load-sharing problem. The categorization presented describes solutions as being either *source initiative* or *server initiative*. In addition, solutions are characterized along a continuous range according to the degree of information dependency involved. The taxonomy presented here takes a much broader view of the distributed scheduling problem in which load-sharing is only one of several possible *basic* strategies available to a system designer. Thus the classifications discussed by Wang describe only a narrow category within the taxonomy.

Among existing taxonomies, one can find examples of flat and hierarchical classification schemes. The taxonomy proposed here is a hybrid of these two—hierarchical as long as possible in order to reduce the total number of classes, and flat when the descriptors of the system may be chosen in an arbitrary order. The levels in the hier-

Manuscript received August 30, 1985.
T. L. Casavant was with the Department of Electrical and Computer Engineering, University of Iowa, Iowa City, IA 52242. He is now with the School of Electrical Engineering, Purdue University, West Lafayette, IN 47907.
J. G. Kuhl is with the Department of Electrical and Computer Engineering, University of Iowa, Iowa City, IA 52242.
IEEE Log Number 8718386.

archy have been chosen in order to keep the description of the taxonomy itself small, and do not necessarily reflect any ordering of importance among characteristics. In other words, the descriptors comprising the taxonomy do not attempt to hierarchically order the characteristics of scheduling systems from more to less general. This point should be stressed especially with respect to the positioning of the flat portion of the taxonomy near the bottom of the hierarchy. For example, load balancing is a characteristic which pervades a large number of distributed scheduling systems, yet for the sake of reducing the size of the description of the taxonomy, it has been placed in the flat portion of the taxonomy and, for the sake of brevity, the flat portion has been placed near the bottom of the hierarchy.

The remainder of the paper is organized as follows. In Section II, the scheduling problem is defined as it applies to distributed resource management. In addition, a taxonomy is presented which serves to allow *qualitative* description and comparison of distributed scheduling systems. Section III will present examples from the literature to demonstrate the use of the taxonomy in qualitatively describing and comparing existing systems. Section IV presents a discussion of issues raised by the taxonomy and also suggests areas in need of additional work.

In addition to the work discussed in the text of the paper, an extensive annotated bibliography is given in an Appendix. This Appendix further demonstrates the effectiveness of the taxonomy in allowing standardized description of existing systems.

II. The Scheduling Problem and Describing Its Solutions

The *general scheduling* problem has been described a number of times and in a number of different ways in the literature [12], [22], [63] and is usually a restatement of the classical notions of job sequencing [13] in the study of production management [7]. For the purposes of distributed process scheduling, we take a broader view of the scheduling function as a *resource management resource*. This management resource is basically a mechanism or policy used to efficiently and effectively manage the access to and use of a resource by its various consumers. Hence, we may view every instance of the scheduling problem as consisting of three main components.

1) Consumer(s).
2) Resource(s).
3) Policy.

Like other management or control problems, understanding the functioning of a scheduler may best be done by observing the effect it has on its environment. In this case, one can observe the behavior of the scheduler in terms of how the *policy* affects the *resources* and *consumers*. Note that although there is only one policy, the scheduler may be viewed in terms of how it affects either or both resources and consumers. This relationship between the scheduler, policies, consumers, and resources is shown in Fig. 1.

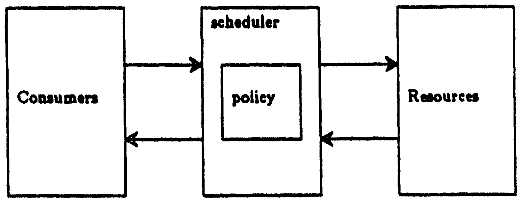


Fig. 1. Scheduling system.

In light of this description of the scheduling problem, there are two properties which must be considered in evaluating any scheduling system 1) the satisfaction of the consumers with how well the scheduler manages the resource in question (performance), and 2) the satisfaction of the consumers in terms of how difficult or costly it is to access the management resource itself (efficiency). In other words, the consumers want to be able to quickly and efficiently access the actual resource in question, but do not desire to be hindered by overhead problems associated with using the management function itself.

One by-product of this statement of the general scheduling problem is the unification of two terms in common use in the literature. There is often an implicit distinction between the terms *scheduling* and *allocation*. However, it can be argued that these are merely alternative formulations of the same problem, with allocation posed in terms of *resource allocation* (from the resources' point of view), and *scheduling* viewed from the consumer's point of view. In this sense, allocation and scheduling are merely two terms describing the same general mechanism, but described from different viewpoints.

A. The Classification Scheme

The usefulness of the four-category taxonomy of computer architecture presented by Flynn [20] has been well demonstrated by the ability to compare systems through their relation to that taxonomy. The goal of the taxonomy given here is to provide a commonly accepted set of terms and to provide a mechanism to allow comparison of past work in the area of distributed scheduling in a qualitative way. In addition, it is hoped that the categories and their relationships to each other have been chosen carefully enough to indicate areas in need of future work as well as to help classify future work.

The taxonomy will be kept as small as possible by proceeding in a hierarchical fashion for as long as possible, but some choices of characteristics may be made independent of previous design choices, and thus will be specified as a set of descriptors from which a subset may be chosen. The taxonomy, while discussed and presented in terms of distributed process scheduling, is applicable to a larger set of resources. In fact, the taxonomy could usefully be employed to classify any set of resource management systems. However, we will focus our attention on the area of process management since it is in this area which we hope to derive relationships useful in determining potential areas for future work.

1) Hierarchical Classification: The structure of the hierarchical portion of the taxonomy is shown in Fig. 2. A discussion of the hierarchical portion then follows.

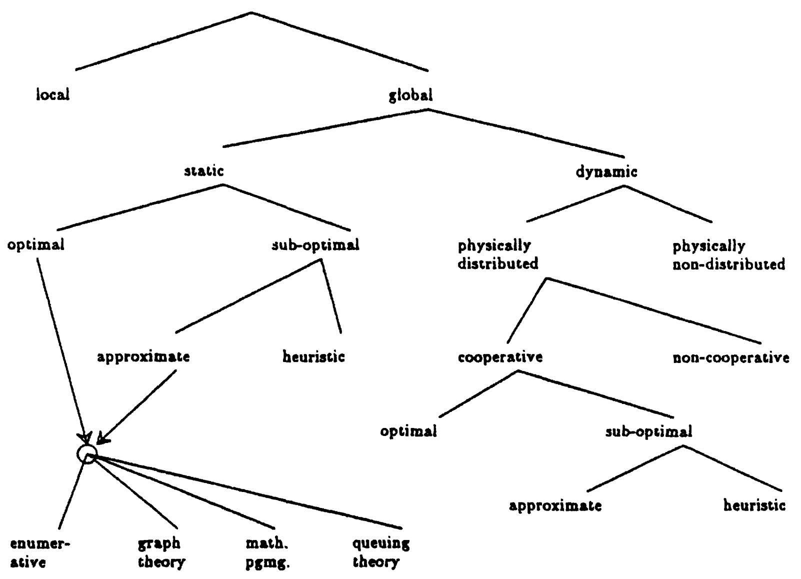


Fig. 2. Task scheduling characteristics.

a) Local Versus Global: At the highest level, we may distinguish between *local* and *global* scheduling. Local scheduling is involved with the assignment of processes to the time-slices of a single processor. Since the area of scheduling on single-processor systems [12], [62] as well as the area of sequencing or job-shop scheduling [13], [18] has been actively studied for a number of years, this taxonomy will focus on global scheduling. Global scheduling is the problem of deciding *where* to execute a process, and the job of local scheduling is left to the operating system of the processor to which the process is ultimately allocated. This allows the processors in a multiprocessor increased autonomy while reducing the responsibility (and consequently overhead) of the global scheduling mechanism. Note that this does not imply that global scheduling must be done by a single central authority, but rather, we view the problems of local and global scheduling as separate issues, and (at least logically) separate mechanisms are at work solving each.

b) Static Versus Dynamic: The next level in the hierarchy (beneath global scheduling) is a choice between *static* and *dynamic* scheduling. This choice indicates the time at which the scheduling or assignment decisions are made.

In the case of static scheduling, information regarding the total mix of processes in the system as well as all the independent subtasks involved in a job or task force [26], [44] is assumed to be available by the time the program object modules are linked into load modules. Hence, each executable image in a system has a static assignment to a particular processor, and each time that process image is submitted for execution, it is assigned to that processor. A more relaxed definition of static scheduling may include algorithms that schedule task forces for a particular hardware configuration. Over a period of time, the topology of the system may change, but characteristics describing the task force remain the same. Hence, the scheduler may generate a new assignment of processes to processors to serve as the schedule until the topology changes again.

Note here that the term *static scheduling* as used in this paper has the same meaning as *deterministic scheduling* in [22] and *task scheduling* in [56]. These alternative terms will not be used, however, in an attempt to develop a consistent set of terms and taxonomy.

c) Optimal Versus Suboptimal: In the case that all information regarding the state of the system as well as the resource needs of a process are known, an *optimal* assignment can be made based on some criterion function [5], [14], [21], [35], [40], [48]. Examples of optimization measures are minimizing total process completion time, maximizing utilization of resources in the system, or maximizing system throughput. In the event that these problems are computationally infeasible, *suboptimal* solutions may be tried [2], [34], [47]. Within the realm of suboptimal solutions to the scheduling problem, we may think of two general categories.

d) Approximate Versus Heuristic: The first is to use the same formal computational model for the algorithm, but instead of searching the entire solution space for an optimal solution, we are satisfied when we find a "good" one. We will categorize these solutions as *suboptimal-approximate*. The assumption that a *good* solution can be recognized may not be so insignificant, but in the cases where a metric is available for evaluating a solution, this technique can be used to decrease the time taken to find an acceptable solution (schedule). The factors which determine whether this approach is worthy of pursuit include:

1) Availability of a function to evaluate a solution.

2) The time required to evaluate a solution.

3) The ability to judge according to some metric the value of an optimal solution.

4) Availability of a mechanism for intelligently pruning the solution space.

The second branch beneath the suboptimal category is labeled *heuristic* [15], [30], [66]. This branch represents the category of static algorithms which make the most realistic assumptions about *a priori* knowledge concerning process and system loading characteristics. It also represents the solutions to the static scheduling problem which require the most reasonable amount of time and other system resources to perform their function. The most distinguishing feature of heuristic schedulers is that they make use of special parameters which affect the system in indirect ways. Often, the parameter being monitored is correlated to system performance in an indirect instead of a direct way, and this alternate parameter is much simpler to monitor or calculate. For example, clustering groups of processes which communicate heavily on the same processor and physically separating processes which would benefit from parallelism [52] directly decreases the overhead involved in passing information between processors, while reducing the interference among processes which may run without synchronization with one another. This result has an impact on the overall service that users receive, but cannot be *directly* related (in a quantitative way) to system performance as the user sees it. Hence, our intuition, if nothing else, leads us to believe that taking the aforementioned actions when possible will improve system performance. However, we may not be able to *prove* that a first-order relationship between the mechanism employed and the desired result exists.

e) Optimal and Suboptimal Approximate Techniques: Regardless of whether a static solution is optimal or suboptimal-approximate, there are four basic categories of task allocation algorithms which can be used to arrive at an assignment of processes to processors.

1) Solution space enumeration and search [48].
2) Graph theoretic [4], [57], [58].
3) Mathematical programming [5], [14], [21], [35], [40].
4) Queueing theoretic [10], [28], [29].

f) Dynamic Solutions: In the dynamic scheduling problem, the more realistic assumption is made that very little *a priori* knowledge is available about the resource needs of a process. It is also unknown in what environment the process will execute during its lifetime. In the static case, a decision is made for a process image before it is ever executed, while in the dynamic case no decision is made until a process begins its life in the dynamic environment of the system. Since it is the responsibility of the running system to decide where a process is to execute, it is only natural to next ask where the decision itself is to be made.

g) Distributed Versus Nondistributed: The next issue (beneath dynamic solutions) involves whether the responsibility for the task of global dynamic scheduling should physically reside in a single processor [44] (*physically nondistributed*) or whether the work involved in making decisions should be *physically distributed* among the processors [17]. Here the concern is with the logical *authority* of the decision-making process.

h) Cooperative Versus Noncooperative: Within the realm of distributed dynamic global scheduling, we may also distinguish between those mechanisms which involve cooperation between the distributed components (*cooperative*) and those in which the individual processors make decisions independent of the actions of the other processors (*noncooperative*). The question here is one of the degree of *autonomy* which each processor has in determining how its own resources should be used. In the noncooperative case individual processors act alone as autonomous entities and arrive at decisions regarding the use of their resources independent of the effect of their decision on the rest of the system. In the cooperative case each processor has the responsibility to carry out its own portion of the scheduling task, but all processors are working toward a common system-wide goal. In other words, each processor's local operating system is concerned with making decisions in concert with the other processors in the system in order to achieve some global goal, instead of making decisions based on the way in which the decision will affect local performance only. As in the static case, the taxonomy tree has reached a point where we may consider optimal, suboptimal-approximate, and suboptimal-heuristic solutions. The same discussion as was presented for the static case applies here as well.

In addition to the hierarchical portion of the taxonomy already discussed, there are a number of other distinguishing characteristics which scheduling systems may have. The following sections will deal with characteristics which do not fit uniquely under any particular branch of the tree-structured taxonomy given thus far, but are still important in the way that they describe the behavior of a scheduler. In other words, the following could be branches beneath several of the leaves shown in Fig. 2 and in the interest of clarity are not repeated under each leaf, but are presented here as a flat extension to the scheme presented thus far. It should be noted that these attributes represent a *set* of characteristics, and any particular scheduling subsystem may possess some subset of this set. Finally, the placement of these characteristics near the bottom of the tree is not intended to be an indication of their relative importance or any other relation to other categories of the hierarchical portion. Their position was determined primarily to reduce the size of the description of the taxonomy.

2) Flat Classification Characteristics:

a) Adaptive Versus Nonadaptive: An adaptive solution to the scheduling problem is one in which the algorithms and parameters used to implement the scheduling policy change dynamically according to the previous and current behavior of the system in response to previous decisions made by the scheduling system. An example of such an adaptive scheduler would be one which takes many parameters into consideration in making its decisions [52]. In response to the behavior of the system, the scheduler may start to ignore one parameter or reduce the importance of that parameter if it believes that parameter is either providing information which is inconsistent with the rest of the inputs or is not providing any information regarding the change in system state in relation to the values of the other parameters being observed. A second example of adaptive scheduling would be one which is based on the stochastic learning automata model [39]. An analogy may be drawn here between the notion of an adaptive scheduler and adaptive control [38], although the usefulness of such an analogy for purposes of performance analysis and implementation are questionable [51]. In contrast to an adaptive scheduler, a nonadaptive scheduler would be one which does not necessarily modify its basic control mechanism on the basis of the history of system activity. An example would be a scheduler which always weighs its inputs in the same way regardless of the history of the system's behavior.

b) Load Balancing: This category of policies, which has received a great deal of attention recently [10], [11], [36], [40]–[42], [46], [53], approaches the problem with the philosophy that being fair to the hardware resources of the system is good for the users of that system. The basic idea is to attempt to balance (in some sense) the load on all processors in such a way as to allow progress by all processes on all nodes to proceed at approximately the same rate. This solution is most effective when the nodes of a system are homogeneous since this allows all nodes to know a great deal about the structure of the other nodes. Normally, information would be passed about the network periodically or on demand [1], [60] in order to allow all nodes to obtain a local estimate concerning the global state of the system. Then the nodes act together in order to remove work from heavily loaded nodes and place it at lightly loaded nodes. This is a class of solutions which relies heavily on the assumption that the information at each node is quite accurate in order to prevent processes from endlessly being circulated about the system without making much progress. Another concern here is deciding on the basic unit used to measure the load on individual nodes.

As was pointed out in Section I, the placement of this characteristic near the bottom of the hierarchy in the flat portion of the taxonomy is not related to its relative importance or generality compared with characteristics at higher levels. In fact, it might be observed that at the point that a choice is made between optimal and suboptimal characteristics, that a specific objective or cost function must have already been made. However, the purpose of the hierarchy is not so much to describe relationships between classes of the taxonomy, but to reduce the size of the overall description of the taxonomy so as to make it more useful in comparing different approaches to solving the scheduling problem.

c) Bidding: In this class of policy mechanisms, a basic protocol framework exists which describes the way in which processes are assigned to processors. The resulting scheduler is one which is usually cooperative in the sense that enough information is exchanged (between nodes with tasks to execute and nodes which may be able to execute tasks) so that an assignment of tasks to processors can be made which is beneficial to all nodes in the system as a whole.

To illustrate the basic mechanism of bidding, the framework and terminology of [49] will be used. Each node in the network is responsible for two roles with respect to the bidding process: *manager* and *contractor*. The manager represents the task in need of a location to execute, and the contractor represents a node which is able to do work for other nodes. Note that a single node takes on both of these roles, and that there are no nodes which are strictly managers or contractors alone. The manager announces the existence of a task in need of execution by a *task announcement*, then receives *bids* from the other nodes (contractors). A wide variety of possibilities exist concerning the type and amount of information exchanged in order to make decisions [53], [59]. The amount and type of information exchanged are the major factors in determining the effectiveness and performance of a scheduler employing the notion of bidding. A very important feature of this class of schedulers is that all nodes generally have full autonomy in the sense that the manager ultimately has the power to decide where to send a task from among those nodes which respond with bids. In addition, the contractors are also autonomous since they are never forced to accept work if they do not choose to do so.

d) Probabilistic: This classification has existed in scheduling systems for some time [13]. The basic idea for this scheme is motivated by the fact that in many assignment problems the number of permutations of the available work and the number of mappings to processors so large, that in order to analytically examine the entire solution space would require a prohibitive amount of time.

Instead, the idea of randómly (according to some known distribution) choosing some process as the next to assign is used. Repeatedly using this method, a number of different schedules may be generated, and then this set is analyzed to choose the best from among those randomly generated. The fact that an important attribute is used to bias the random choosing process would lead one to expect that the schedule would be better than one chosen entirely at random. The argument that this method actually produces a good selection is based on the expectation that enough variation is introduced by the random choosing to allow a *good* solution to get into the randomly chosen set.

An alternative view of probabilistic schedulers are those which employ the principles of decision theory in the form of team theory [24]. These would be classified as probabilistic since suboptimal decisions are influenced by prior probabilities derived from *best-guesses* to the actual states of nature. In addition, these prior probabilities are used to determine (utilizing some random experiment) the next action (or scheduling decision).

e) One-Time Assignment Versus Dynamic Reassignment: In this classification, we consider the entities to be scheduled. If the entities are *jobs* in the traditional batch processing sense of the term [19], [23], then we consider the single point in time in which a decision is made as to where and when the job is to execute. While this technique technically corresponds to a dynamic approach, it is static in the sense that once a decision is made to place and execute a job, no further decisions are made concerning the job. We would characterize this class as one-time assignments. Notice that in this mechanism, the only information usable by the scheduler to make its decision is the information given it by the user or submitter of the job. This information might include estimated execution time or other system resource demands. One critical point here is the fact that once users of a system understand the underlying scheduling mechanism, they may present false information to the system in order to receive better response. This point fringes on the area of psychological behavior, but human interaction is an important design factor to consider in this case since the behavior of the scheduler itself is trying to mimic a general philosophy. Hence, the interaction of this philosophy with the system's users must be considered.

In contrast, solutions in the dynamic reassignment class try to improve on earlier decisions by using information on smaller computation units—the executing subtasks of jobs or task forces. This category represents the set of systems which 1) do not trust their users to provide accurate descriptive information, and 2) use dynamically created information to adapt to changing demands of user processes. This adaptation takes the form of migrating processes (including current process state information). There is clearly a price to be paid in terms of overhead, and this price must be carefully weighed against possible benefits.

An interesting analogy exists between the differentiation made here and the question of preemption versus nonpreemption in uniprocessor scheduling systems. Here, the difference lies in whether to move a process from one place to another once an assignment has been made, while in the uniprocessor case the question is whether to remove the running process from the processor once a decision has been made to let it run.

III. Examples

In this section, examples will be taken from the published literature to demonstrate their relationships to one another with respect to the taxonomy detailed in Section II. The purpose of this section is twofold. The first is to show that many different scheduling algorithms can fit into the taxonomy and the second is to show that the categories of the taxonomy actually correspond, in most cases, to methods which have been examined.

A. Global Static

In [48], we see an example of an optimal, enumerative approach to the task assignment problem. The criterion function is defined in terms of optimizing the amount of time a task will require for all interprocess communication and execution, where the tasks submitted by users are assumed to be broken into suitable modules before execution. The cost function is called a *minimax criterion* since it is intended to minimize the maximum execution and communication time required by any single processor involved in the assignment. Graphs are then used to represent the module to processor assignments and the assignments are then transformed to a type of graph matching known as weak homomorphisms. The optimal search of this solution space can then be done using the A^* algorithm from artificial intelligence [43]. The solution also achieves a certain degree of processor load balancing as well.

Reference [4] gives a good demonstration of the usefulness of the taxonomy in that the paper describes the algorithm given as a solution to the optimal dynamic assignment problem for a two processor system. However, in attempting to make an objective comparison of this paper with other *dynamic* systems, we see that the algorithm proposed is actually a static one. In terms of the taxonomy of Section II, we would categorize this as a static, optimal, graph theoretical approach in which the *a priori* assumptions are expanded to include more information about the set of tasks to be executed. The way in which reassignment of tasks is performed during process execution is decided upon before any of the program modules begin execution. Instead of making reassignment *decisions* during execution, the stronger assumption is simply made that all information about the dynamic needs of a collection of program modules is available *a priori*. This assumption says that if a collection of modules possess a certain communication pattern at the beginning of their execution, and this pattern is completely predictable, that this pattern may change over the course of execution and that these variations are predictable as well. Costs of relocation are also assummed to be available, and this assumption appears to be quite reasonable.

The model presented in [35] represents an example of an optimum mathematical programming formulation employing a branch and bound technique to search the solution space. The goals of the solution are to minimize interprocessor communications, balance the utilization of all processors, and satisfy all other engineering application requirements. The model given defines a cost function which includes interprocessor communication costs and processor execution costs. The assignment is then represented by a set of zero-one variables, and the total execution cost is then represented by a summation of all costs incurred in the assignment. In addition to the above, the problem is subject to constraints which allow the solution to satisfy the load balancing and engineering application requirements. The algorithm then used to search the solution space (consisting of all potential assignments) is derived from the basic branch and bound technique.

Again, in [10], we see an example of the use of the taxonomy in comparing the proposed system to other approaches. The title of the paper—"Load Balancing in Distributed Systems"—indicates that the goal of the solution is to balance the load among the processors in the system in some way. However, the solution actually fits into the static, optimal, queueing theoretical class. The goal of the solution is to minimize the execution time of the entire program to maximize performance and the algorithm is derived from results in Markov decision theory. In contrast to the definition of load balancing given in Section II, where the goal was to even the load and utilization of system resources, the approach in this paper is consumer oriented.

An interesting approximate mathematical programming solution, motivated from the viewpoint of fault-tolerance, is presented in [2]. The algorithm is suggested by the computational complexity of the optimal solution to the same problem. In the basic solution to a mathematical programming problem, the state space is either implicitly or explicitly enumerated and searched. One approximation method mentioned in this paper [64] involves first removing the integer constraint, solving the continuous optimization problem, discretizing the continuous solution, and obtaining a bound on the discretization error. Whereas this bound is with respect to the continuous optimum, the algorithm proposed in this paper directly uses an approximation to solve the discrete problem and bound its performance with respect to the discrete optimum.

The last static example to be given here appears in [66]. This paper gives a heuristic-based approach to the problem by using extractable data and synchronization requirements of the different subtasks. The three primary heuristics used are:

1) loss of parallelism,
2) synchronization,
3) data sources.

The way in which loss of parallelism is used is to assign tasks to nodes one at a time in order to affect the least loss of parallelism based on the number of units required for execution by the task currently under consideration. The synchronization constraints are phrased in terms of *firing conditions* which are used to describe precedence relationships between subtasks. Finally, data source information is used in much the same way a functional program uses precedence relations between parallel portions of a computation which take the roles of varying classes of suppliers of variables to other subtasks. The final heuristic algorithm involves weighting each of the previous heuristics, and combining them. A distinguishing feature of the algorithm is its use of a greedy approach to find a solution, when at the time decisions are made, there can be no guarantee that a decision is optimal. Hence, an optimal solution would more carefully search the solution space using a back track or branch and bound method, as well as using exact optimization criterion instead of the heuristics suggested.

B. Global Dynamic

Among the dynamic solutions presented in the literature, the majority fit into the general category of physically distributed, cooperative, suboptimal, heuristic. There are, however, examples for some of the other classes.

First, in the category of physically nondistributed, one of the best examples is the experimental system developed for the Cm* architecture—Medusa [44]. In this system, the functions of the operating system (e.g., file system, scheduler) are physically partitioned and placed at different places in the system. Hence, the scheduling function is placed at a particular place and is accessed by all users at that location.

Another rare example exists in the physically distributed *noncooperative class*. In this example [27], random level-order scheduling is employed at all nodes independently in a tightly coupled MIMD machine. Hence, the overhead involved in this algorithm is minimized since no information need be exchanged to make random decisions. The mechanism suggested is thought to work best in moderate to heavily loaded systems since in these cases, a random policy is thought to give a reasonably balanced load on all processors. In contrast to a cooperative solution, this algorithm does not detect or try to avoid system overloading by sharing loading information among processors, but makes the assumption that it will be under heavy load most of the time and bases all of its decisions on that assumption. Clearly, here, the processors are not necessarily concerned with the utilization of their own resources, but neither are they concerned with the effect their individual decisions will have on the other processors in the system.

It should be pointed out that although the above two algorithms (and many others) are given in terms relating to general-purpose distributed processing systems, that they do not strictly adhere to the definition of distributed data processing system as given in [17].

In [57], another rare example exists in the form of a physically distributed, cooperative, *optimal* solution in a dynamic environment. The solution is given for the two-processor case in which critical load factors are calculated prior to program execution. The method employed is to use a graph theoretical approach to solving for load factors for each process on each processor. These load factors are then used at run time to determine when a task could run better if placed on the other processor.

The final class (and largest in terms of amount of existing work) is the class of physically distributed, cooperative, suboptimal, heuristic solutions.

In [53] a solution is given which is adaptive, load balancing, and makes one-time assignments of *jobs* to processors. No *a priori* assumptions are made about the characteristics of the jobs to be scheduled. One major restriction of these algorithms is the fact that they only consider assignment of jobs to processors and once a job becomes an active process, no reassignment of processes is considered regardless of the possible benefit. This is very defensible, though, if the overhead involved in moving a process is very high (which may be the case in many circumstances). Whereas this solution cannot exactly be considered as a bidding approach, exchange of information occurs between processes in order for the algorithms to function. The first algorithm (a copy of which resides at each host) compares its own *busyness* with its estimate of the busyness of the least busy host. If the difference exceeds the bias (or threshold) designated at the current time, one job is moved from the job queue of the busier host to the less busy one. The second algorithm allows each host to compare itself with all other hosts and involves two biases. If the difference exceeds bias1 but not bias2, then one job is moved. If the difference exceeds bias2, then two jobs are moved. There is also an upper limit set on the number of jobs which can move at once in the entire system. The third algorithm is the same as algorithm one except that an antithrashing mechanism is added to account for the fact that a delay is present between the time a decision is made to move a job, and the time it arrives at the destination. All three algorithms had an adaptive feature added which would turn off all parts of the respective algorithm except the monitoring of load when system load was below a particular minimum threshold. This had the effect of stopping *processor thrashing* whenever it was practically impossible to balance the system load due to lack of work to balance. In the high load case, the algorithm was turned off to reduce extraneous overhead when the algorithm could not affect any improvment in the system under any redistribution of jobs. This last feature also supports the notion in the noncooperative example given earlier that the load is usually automatically balanced as a side effect of heavy loading. The remainder of the paper focuses on simulation results to reveal the impact of modifying the biasing parameters.

The work reported in [6] is an example of an algorithm which employs the heuristic of load-balancing, and probabilistically estimates the remaining processing times of processes in the system. The remaining processing time for a process was estimated by one of the following methods:

memoryless: $Re(t) = E\{S\}$

pastrepeats: $Re(t) = t$

distribution: $Re(t) = E\{S - t | S > t\}$

optimal: $Re(t) = R(t)$

where $R(t)$ is the remaining time needed given that t seconds have already elapsed, S is the service time random variable, and $Re(t)$ is the scheduler's estimate of $R(t)$. The algorithm then basically uses the first three methods to predict response times in order to obtain an expected delay measure which in turn is used by pairs of processors to balance their load on a pairwise basis. This mechanism is adopted by all pairs on a dynamic basis to balance the system load.

Another adaptive algorithm is discussed in [52] and is based on the bidding concept. The heuristic mentioned here utilizes prior information concerning the known characteristics of processes such as resource requirements, process priority, special resource needs, precedence constraints, and the need for clustering and distributed groups. The basic algorithm periodically evaluates each process at a current node to decide whether to transmit bid requests for a particular process. The bid requests include information needed for contractor nodes to make decisions regarding how well they may be able to execute the process in question. The manager receives bids and compares them to the local evaluation and will transfer the process if the difference between the best bid and the local estimate is above a certain threshold. The key to the algorithm is the formulation of a function to be used in a modified McCulloch-Pitts neuron. The neuron (implemented as a subroutine) evaluates the current performance of individual processes. Several different functions were proposed, simulated, and discussed in this paper. The adaptive nature of this algorithm is in the fact that it dynamically modifies the number of hops that a bid request is allowed to travel depending on current conditions. The most significant result was that the information regarding process clustering and distributed groups seems to have had little impact on the overall performance of the system.

The final example to be discussed here [55] is based on a heuristic derived from the area of Bayesian decision theory [33]. The algorithm uses no *a priori* knowledge regarding task characteristics, and is dynamic in the sense that the probability distributions which allow maximizing decisions to be made based on the most likely current state of nature are updated dynamically. Monitor nodes make observations every p seconds and update probabilities. Every d seconds the scheduler itself is invoked to approximate the current state of nature and make the appropriate maximizing action. It was found that the parameters p and d could be tuned to obtain maximum performance for a minimum cost.

IV. Discussion

In this section, we will attempt to demonstrate the application of the qualitative description tool presented earlier to a role beyond that of classifying existing systems. In particular, we will utilize two behavior characteristics—*performance* and *efficiency*, in conjunction with the classification mechanism presented in the taxonomy, to identify general qualities of scheduling systems which will lend themselves to managing large numbers of processors. In addition, the uniform terminology presented will be employed to show that some earlier-thought-to-be-synonymous notions are actually distinct, and to show that the distinctions are valuable. Also, in at least one case, two earlier-thought-to-be-different notions will be shown to be much the same.

A. Decentralized Versus Distributed Scheduling

When considering the decision-making policy of a scheduling system, there are two fundamental components—responsibility and authority. When responsibility for making and carrying out policy decisions is shared among the entities in a distributed system, we say that the scheduler is *distributed*. When authority is distributed to the entities of a resource management system, we call this *decentralized*. This differentiation exists in many other organizational structures. Any system which possesses decentralized authority must have distributed responsibility, but it is possible to allocate responsibility for gathering information and carrying out policy decisions, without giving the authority to change past or make future decisions.

B. Dynamic Versus Adaptive Scheduling

The terms *dynamic scheduling* and *adaptive scheduling* are quite often attached to various proposed algorithms in the literature, but there appears to be some confusion as to the actual difference between these two concepts. The more common property to find in a scheduler (or resource management subsystem) is the dynamic property. In a dynamic situation, the scheduler takes into account the current state of affairs as it perceives them in the system. This is done during the normal operation of the system under a dynamic and unpredictable load. In an adaptive system, the scheduling policy itself reflects changes in its environment—the running system. Notice that the difference here is one of level in the hierarchical solution to the scheduling problem. Whereas a dynamic solution takes environmental inputs into account when making its decisions, an adaptive solution takes environmental stimuli into account to modify the scheduling policy itself.

C. The Resource/Consumer Dichotomy in Performance Analysis

As is the case in describing the actions or qualitative behavior of a resource management subsystem, the performance of the scheduling mechanisms employed may be viewed from either the resource or consumer point of view. When considering performance from the consumer (or user) point of view, the metric involved is often one of minimizing individual program completion times—*response*. Alternately, the resource point of view also considers the rate of process execution in evaluating performance, but from the view of total system throughput. In contrast to response, *throughput* is concerned with seeing that *all* users are treated fairly and that *all* are making progress. Notice that the resource view of maximizing resource utilization is compatible with the desire for maximum system throughput. Another way of stating this, however, is that all users, when considered as a single collective user, are treated best in this environment of maximizing system throughput or maximizing resource utilization. This is the basic philosophy of load-balancing mechanisms. There is an inherent conflict, though, in trying to optimize both response and throughput.

D. Focusing on Future Directions

In this section, the earlier presented taxonomy, in conjunction with two terms used to quantitatively describe system behavior, will be used to discuss possibilities for distributed scheduling in the environment of a large system of loosely coupled processors.

In previous work related to the scheduling problem, the basic notion of performance has been concerned with evaluating the way in which users' individual needs are being satisfied. The metrics most commonly applied are *response* and *throughput* [23]. While these terms accurately characterize the goals of the system in terms of how well users are served, they are difficult to measure during the normal operation of a system. In addition to this problem, the metrics do not lend themselves well to direct interpretation as to the action to be performed to increase performance when it is not at an acceptable level.

These metrics are also difficult to apply when analysis or simulation of such systems is attempted. The reason for this is that two important aspects of scheduling are necessarily intertwined. These two aspects are *performance* and *efficiency*. Performance is the part of a system's behavior that encompasses how well the resource to be managed is being used to the benefit of all users of the system. Efficiency, though, is concerned with the added cost (or overhead) associated with the resource management facility itself. In terms of these two criteria, we may think of desirable system behavior as that which has the highest level of performance possible, while incurring the least overhead in doing it. Clearly, the exact combination of these two which brings about the most desirable behavior is dependent on many factors and in many ways resembles the space/time tradeoff present in common algorithm design. The point to be made here is that simultaneous evaluation of efficiency and performance is very difficult due to this inherent entanglement. What we suggest is a methodology for designing scheduling systems in which efficiency and performance are separately observable.

Current and future investigations will involve studies to better understand the relationships between performance, efficiency, and their components as they effect quantitative behavior. It is hoped that a much better understanding can be gained regarding the costs and benefits of alternative distributed scheduling strategies.

V. Conclusion

This paper has sought to bring together the ideas and work in the area of resource management generated in the last 10 to 15 years. The intention has been to provide a suitable framework for comparing past work in the area of resource management, while providing a tool for classifying and discussing future work. This has been done through the presentation of common terminology and a taxonomy on the mechanisms employed in computer system resource management. While the taxonomy could be used to discuss many different types of resource management, the attention of the paper and included examples have been on the application of the taxonomy to the processing resource. Finally, recommendations regarding possible fruitful areas for future research in the area of scheduling in large scale general-purpose distributed computer systems have been discussed.

As is the case in any survey, there are many pieces of work to be considered. It is hoped that the examples presented fairly represent the true state of research in this area, while it is acknowledged that not *all* such examples have been discussed. In addition to the references at the end of this paper, the Appendix contains an annotated bibliography for work not explicitly mentioned in the text but which have aided in the construction of this taxonomy through the support of additional examples. The exclusion of any particular results has not been intentional nor should it be construed as a judgment of the merit of that work. Decisions as to which papers to use as examples were made purely on the basis of their applicability to the context of the discussion in which they appear.

Appendix
Annotated Bibliography

Application of Taxonomy to Examples from Literature

This Appendix contains references to additional examples not included in Section III as well as abbreviated descriptions of those examples discussed in detail in the text of the paper. The purpose is to demonstrate the use of the taxonomy of Section II in classifying a large number of examples from the literature.

[1] G. R. Andrews, D. P. Dobkin, and P. J. Downey, "Distributed allocation with pools of servers," in *ACM SIGACT-SIGOPS Symp. Principles of Distributed Computing*, Aug. 1982, pp. 73-83.
Global, dynamic, distributed (however in a limited sense), cooperative, suboptimal, heuristic, bidding, nonadaptive, dynamic reassignment.

[2] J. A. Bannister and K. S. Trivedi, "Task allocation in fault-tolerant distributed systems," *Acta Inform.*, vol. 20, pp. 261-281, 1983.
Global, static, suboptimal, approximate, mathematical programming.

[3] F. Berman and L. Snyder, "On mapping parallel algorithms into parallel architectures," in *1984 Int. Conf. Parallel Proc.*, Aug. 1984, pp. 307-309.
Global, static, optimal, graph theory.

[4] S. H. Bokhari, "Dual processor scheduling with dynamic reassignment," *IEEE Trans. Software Eng.*, vol. SE-5, no. 4, pp. 326-334, July 1979.
Global, static, optimal, graph theoretic.

[5] ——, "A shortest tree algorithm for optimal assignments across space and time in a distributed processor system," *IEEE Trans. Software Eng.*, vol. SE-7, no. 6, pp. 335-341, Nov. 1981.
Global, static, optimal, mathematical programming, intended for tree-structured applications.

[6] R. M. Bryant and R. A. Finkel, "A stable distributed scheduling algorithm," in *Proc. 2nd Int. Conf. Dist. Comp.*, Apr. 1981, pp. 314-323.
Global, dynamic, physically distributed, cooperative, suboptimal, heuristic, probabilistic, load-balancing.

[7] T. L. Casavant and J. G. Kuhl, "Design of a loosely-coupled distributed multiprocessing network," in *1984 Int. Conf. Parallel Proc.*, Aug. 1984, pp. 42-45.
Global, dynamic, physically distributed, cooperative, suboptimal, heuristic, load-balancing, bidding, dynamic reassignment.

[8] L. M. Casey, "Decentralized scheduling," *Australian Comput. J.*, vol. 13, pp. 58-63, May 1981.
Global, dynamic, physically distributed, cooperative, suboptimal, heuristic, load-balancing.

[9] T. C. K. Chou and J. A. Abraham, "Load balancing in distributed systems," *IEEE Trans. Software Eng.*, vol. SE-8, no. 4, pp. 401-412, July 1982.
Global, static, optimal, queueing theoretical.

[10] T. C. K. Chou and J. A. Abraham, "Load redistribution under failure in distributed systems," *IEEE Trans. Comput.*, vol. C-32, no. 9, pp. 799-808, Sept. 1983.
Global, dynamic (but with static parings of supporting and supported processors), distributed, cooperative, suboptimal, provides 3 separate heuristic mechanisms, motivated from fault recovery aspect.

[11] Y. C. Chow and W. H. Kohler, "Models for dynamic load balancing in a heterogeneous multiple processor system," *IEEE Trans. Comput.*, vol. C-28, no. 5, pp. 354-361, May 1979.
Global, dynamic, physically distributed, cooperative, suboptimal, heuristic, load-balancing, (part of the heuristic approach is based on results from queueing theory).

[12] W. W. Chu *et al.*, "Task allocation in distributed data processing," *Computer*, vol. 13, no. 11, pp. 57-69, Nov. 1980.
Global, static, optimal, suboptimal, heuristic, heuristic approached based on graph theory and mathematical programming are discussed.

[13] K. W. Doty, P. L. McEntire, and J. G. O'Reilly, "Task allocation in a distributed computer system," in *IEEE InfoCom*, 1982, pp. 33-38.
Global, static, optimal, mathematical programming (nonlinear spatial dynamic programming).

[14] K. Efe, "Heuristic models of task assignment scheduling in distributed systems," *Computer*, vol. 15, pp. 50-56, June 1982.
Global, static, suboptimal, heuristic, load-balancing.

[15] J. A. B. Fortes and F. Parisi-Presicce, "Optimal linear schedules for the parallel execution of algorithms," in *1984 Int. Conf. Parallel Proc.*, Aug. 1984, pp. 322-329.
Global, static, optimal, uses results from mathematical programming for a large class of data-dependency driven applications.

[16] A. Gabrielian and D. B. Tyler, "Optimal object allocation in distributed computer systems," in *Proc. 4th Int. Conf. Dist. Comp. Systems*, May 1984, pp. 84-95.
Global, static, optimal, mathematical programming, uses a heuristic to obtain a solution close to optimal, employs backtracking to find optimal one from that.

[17] C. Gao, J. W. S. Liu, and M. Railey, "Load bal-

ancing algorithms in homogeneous distributed systems," in *1984 Int. Conf. Parallel Proc.*, Aug. 1984, pp. 302-306.

Global, dynamic, distributed, cooperative, suboptimal, heuristic, probabilistic.

[18] W. Huen *et al.*, "TECHNEC, A network computer for distributed task control," in *Proc. 1st Rocky Mountain Symp. Microcomputers*, Aug. 1977, pp. 233-237.

Global, static, suboptimal, heuristic.

[19] K. Hwang *et al.*, "A Unix-based local computer network with load balancing," *Computer*, vol. 15, no. 4, pp. 55-65, Apr. 1982.

Global, dynamic, physically distributed, cooperative, suboptimal, heuristic, load-balancing.

[20] D. Klappholz and H. C. Park, "Parallelized process scheduling for a tightly-coupled MIMD machine," in *1984 Int. Conf. Parallel Proc.*, Aug. 1984, pp. 315-321.

Global, dynamic, physically distributed, noncooperative.

[21] C. P. Kruskal and A. Weiss, "Allocating independent subtasks on parallel processors extended abstract," in *1984 Int. Conf. Parallel Proc.*, Aug. 1984, pp. 236-240.

Global, static, suboptimal, but optimal for a set of optimistic assumptions, heuristic, problem stated in terms of queuing theory.

[22] V. M. Lo, "Heuristic algorithms for task assignment in distributed systems," in *Proc. 4th Int. Conf. Dist. Comp. Systems*, May 1984, pp. 30-39.

Global, static, suboptimal, approximate, graph theoretic.

[23] ——, "Task assignment to minimize completion time," in *5th Int. Conf. Distributed Computing Systems*, May 1985, pp. 329-336.

Global, static, optimal, mathematical programming for some special cases, but in general is suboptimal, heuristic using the LPT algorithm.

[24] P. Y. R. Ma, E. Y. S. Lee, and J. Tsuchiya, "A task allocation model for distributed computing systems," *IEEE Trans. Comput.*, vol. C-31, no. 1, pp. 41-47, Jan. 1982.

Global, static, optimal, mathematical programming (branch and bound).

[25] S. Majumdar and M. L. Green, "A distributed real time resource manager," in *Proc. IEEE Symp. Distributed Data Acquisition, Computing and Control*, 1980, pp. 185-193.

Global, dynamic, distributed, cooperative, suboptimal, heuristic, load balancing, nonadaptive.

[26] R. Manner, "Hardware task/processor scheduling in a polyprocessor environment," *IEEE Trans. Comput.*, vol. C-33, no. 7, pp. 626-636, July 1984.

Global, dynamic, distributed control and responsibility, but centralized information in hardware on bus lines. Cooperative, optimal, (priority) load balancing.

[27] L. M. Ni and K. Hwang, "Optimal load balancing for a multiple processor system," in *Proc. Int. Conf. Parallel Proc.*, 1981, pp. 352-357.

Global, static, optimal, mathematical programming.

[28] L. M. Ni and K. Abani, "Nonpreemptive load balancing in a class of local area networks," in *Proc. Comp. Networking Symp.*, Dec. 1981, pp. 113-118.

Global, dynamic, distributed, cooperative, optimal and suboptimal solutions given—mathematical programming, and adaptive load balancing, respectively.

[29] J. Ousterhout, D. Scelza, and P. Sindhu, "Medusa: An experiment in distributed operating system structure," *Commun. ACM*, vol. 23, no. 2, pp. 92-105, Feb. 1980.

Global, dynamic, physically nondistributed.

[30] M. L. Powell and B. P. Miller, "Process migration in DEMOS/MP," in *Proc. 9th Symp. Operating Systems Principles* (OS Review), vol. 17, no. 5, pp. 110-119, Oct. 1983.

Global, dynamic, distributed, cooperative, suboptimal, heuristic, load balancing but no specific decision rule given.

[31] C. C. Price and S. Krishnaprasad, "Software allocation models for distributed computing systems," in *Proc. 4th Int. Conf. Dist. Comp. Systems*, May 1984, pp. 40-48.

Global, static, optimal, mathematical programming, but also suggest heuristics.

[32] C. V. Ramamoorthy *et al.*, "Optimal scheduling strategies in a multiprocessor system," *IEEE Trans. Comput.*, vol. C-21, no. 2, pp. 137-146, Feb. 1972.

Global, static, optimal solution presented for comparison with the heuristic one also presented. Graph theory is employed in the sense that it uses task precedence graphs.

[33] K. Ramamritham and J. A. Stankovic, "Dynamic task scheduling in distributed hard real-time systems," in *Proc. 4th Int. Conf. Dist. Comp. Systems*, May 1984, pp. 96-107.

Global, dynamic, distributed, cooperative, suboptimal, heuristic, bidding, one-time assignments (a real time guarantee is applied before migration).

[34] J. Reif and P. Spirakis, "Real-time resource allocation in a distributed system," in *ACM SIGACT-SIGOPS Symp. Principles of Distributed Computing*, Aug. 1982, pp. 84-94.

Global, dynamic, distributed, noncooperative, probabilistic.

[35] S. Sahni, "Scheduling multipipeline and multiprocessor computers," in *1984 Int. Conf. Parallel Processing*, Aug. 1984, pp. 333-337.

Global, static, suboptimal, heuristic.

[36] T. G. Saponis and P. L. Crews, "A model for decentralized control in a fully distributed processing system," in *Fall COMPCON*, 1980, pp. 307-312.

Global, static, suboptimal, heuristic based on load

balancing. Also intended for applications of the nature of coupled recurrence systems.

[37] C. C. Shen and W. H. Tsai, "A graph matching approach to optimal task assignment in distributed computing systems using a minimax criterion," *IEEE Trans. Comput.*, vol. C-34, no. 3, pp. 197–203, Mar. 1985.

Global, static, optimal, enumerative.

[38] J. A. Stankovic, "The analysis of a decentralized control algorithm for job scheduling utilizing Bayesian decision theory," in *Proc. Int. Conf. Parallel Proc.*, 1981, pp. 333–337.

Global, dynamic, distributed, cooperative, suboptimal, heuristic, one-time assignment, probabilistic.

[39] —–, "A heuristic for cooperation among decentralized controllers," in *IEEE INFOCOM 1983*, Apr. 1983, pp. 331–339.

Global, dynamic, distributed, cooperative, suboptimal, heuristic, one-time assignment, probabilistic.

[40] J. A. Stankovic and I. S. Sidhu, "An adaptive bidding algorithm for processes, clusters and distributed groups," in *Proc. 4th Int. Conf. Dist. Comp. Systems*, May 1984, pp. 49–59.

Global, dynamic, physically distributed, cooperative, suboptimal, heuristic, adaptive, bidding, additional heuristics regarding clusters and distributed groups.

[41] J. A. Stankovic, "Simulations of three adaptive, decentralized controlled, job scheduling algorithms," *Comput. Networks*, vol. 8, no. 3, pp. 199–217, June 1984.

Global, dynamic, physically distributed, cooperative, suboptimal, heuristic, adaptive, load-balancing, one-time assignment. Three variants of this basic approach given.

[42] —–, "An application of Bayesian decision theory to decentralized control of job scheduling," *IEEE Trans. Comput.*, vol. C-34, no. 2, pp. 117–130, Feb. 1985.

Global, dynamic, physically distributed, cooperative, suboptimal, heuristic based on results from Bayesian decision theory.

[43] —–, "Stability and distributed scheduling algorithms," in *Proc. ACM Nat. Conf.*, New Orleans, Mar. 1985.

Here there are two separate algorithms specified. The first is a Global, dynamic, physically distributed, cooperative, heuristic, adaptive, dynamic reassignment example based on stochastic learning automata. The second is a Global, dynamic, physically distributed, cooperative, heuristic, bidding, one-time assignment approach.

[44] H. S. Stone, "Critical load factors in two-processor distributed systems," *IEEE Trans. Software Eng.*, vol. SE-4, no. 3, pp. 254–258, May 1978.

Global, dynamic, physically distributed, cooperative, optimal, (graph theory based).

[45] H. S. Stone and S. H. Bokhari, "Control of distributed processes," *Computer*, vol. 11, pp. 97–106, July 1978.

Global, static, optimal, graph theoretical.

[46] H. Sullivan and T. Bashkow, "A large-scale homogeneous, fully distributed machine—I," in *Proc. 4th Symp. Computer Architecture*, Mar. 1977, pp. 105–117.

Global, dynamic, physically distributed, cooperative, suboptimal, heuristic, bidding.

[47] A. M. VanTilborg and L. D. Wittie, "Wave scheduling—Decentralized scheduling of task forces in multicomputers," *IEEE Trans. Comput.*, vol. C-33, no. 9, pp. 835–844, Sept. 1984.

Global, dynamic, distributed, cooperative, suboptimal, heuristic, probabilistic, adaptive. Assumes tree-structured (logically) task-forces.

[48] R. A. Wagner and K. S. Trivedi, "Hardware configuration selection through discretizing a continuous variable solution," in *Proc. 7th IFIP Symp. Comp. Performance Modeling, Measurement and Evaluation*, Toronto, Canada, 1980, pp. 127–142.

Global, static, suboptimal, approximate, mathematical programming.

[49] Y. T. Wang and R. J. T. Morris, "Load sharing in distributed systems," *IEEE Trans. Comput.*, vol. C-34, no. 3, pp. 204–217, Mar. 1985.

Global, dynamic, physically distributed, cooperative, suboptimal, heuristic, one-time assignment, load-balancing.

[50] M. O. Ward and D. J. Romero, "Assigning parallel-executable, intercommunicating subtasks to processors," in *1984 Int. Conf. Parallel Proc.*, Aug. 1984, pp. 392–394.

Global, static, suboptimal, heuristic.

[51] L. D. Wittie and A. M. Van Tilborg, "MICROS, a distributed operating system for MICRONET, a reconfigurable network computer," *IEEE Trans. Comput.*, vol. C-29, no. 12, pp. 1133–1144, Dec. 1980.

Global, dynamic, physically distributed, cooperative, suboptimal, heuristic, load-balancing (also with respect to message traffic).

References

[1] A. K. Agrawala, S. K. Tripathi, and G. Ricart, "Adaptive routing using a virtual waiting time technique," *IEEE Trans. Software Eng.*, vol. SE-8, no. 1, pp. 76–81, Jan. 1982.

[2] J. A. Bannister and K. S. Trivedi, "Task allocation in fault-tolerant distributed systems," *Acta Inform.*, vol. 20, pp. 261–281, 1983.

[3] J. F. Bartlett, "A nonstop kernel," in *Proc. 8th Symp. Operating Systems Principles*, Dec. 1981, pp. 22–29.

[4] S. H. Bokhari, "Dual processor scheduling with dynamic reassignment," *IEEE Trans. Software Eng.*, vol. SE-5, no. 4, pp. 326–334, July 1979.

[5] S. H. Bokhari, "A shortest tree algorithm for optimal assignments across space and time in a distributed processor system," *IEEE Trans. Software Eng.*, vol. SE-7, no. 6, pp. 335–341, Nov. 1981.

[6] R. M. Bryant and R. A. Finkel, "A stable distributed scheduling algorithm," in *Proc. 2nd Int. Conf. Dist. Comp.*, Apr. 1981, pp. 314–323.

[7] E. S. Buffa, *Modern Production Management*, 5th ed. New York: Wiley, 1977.

[8] T. L. Casavant and J. G. Kuhl, "Design of a loosely-coupled distributed multiprocessing network," in *1984 Int. Conf. Parallel Proc.*, Aug. 1984, pp. 42-45.

[9] L. M. Casey, "Decentralized scheduling," *Australian Comput. J.*, vol. 13, pp. 58-63, May 1981.

[10] T. C. K. Chou and J. A. Abraham, "Load balancing in distributed systems," *IEEE Trans. Software Eng.*, vol. SE-8, no. 4, pp. 401-412, July 1982.

[11] Y. C. Chow and W. H. Kohler, "Models for dynamic load balancing in a heterogeneous multiple processor system," *IEEE Trans. Comput.*, vol. C-28, no. 5, pp. 354-361, May 1979.

[12] E. G. Coffman and P. J. Denning, *Operating Systems Theory*. Englewood Cliffs, NJ: Prentice-Hall, 1973.

[13] R. W. Conway, W. L. Maxwell, and L. W. Miller, *Theory of Scheduling*. Reading, MA: Addison-Wesley, 1967.

[14] K. W. Doty, P. L. McEntire, and J. G. O'Reilly, "Task allocation in a distributed computer system," in *IEEE InfoCom*, 1982, pp. 33-38.

[15] K. Efe, "Heuristic models of task assignment scheduling in distributed systems," *Computer*, vol. 15, pp. 50-56, June 1982.

[16] C. S. Ellis, J. A. Feldman, and J. E. Heliotis, "Language constructs and support systems for distributed computing," in *ACM SIGACT-SIGOPS Symp. Principles of Distributed Computing*, Aug. 1982, pp. 1-9.

[17] P. H. Enslow Jr., "What is a "distributed" data processing system," *Computer*, vol. 11, no. 1, pp. 13-21, Jan. 1978.

[18] J. R. Evans *et al.*, *Applied Production and Operations Management*. St. Paul, MN: West, 1984.

[19] I. Flores, *OSMVT*. Boston, MA: Allyn and Bacon, 1973.

[20] M. J. Flynn, "Very high-speed computing systems," *Proc. IEEE*, vol. 54, pp. 1901-1909, Dec. 1966.

[21] A. Gabrielian and D. B. Tyler, "Optimal object allocation in distributed computer systems," in *Proc. 4th Int. Conf. Dist. Comp. Systems*, May 1984, pp. 84-95.

[22] M. J. Gonzalez, "Deterministic processor scheduling," *ACM Comput. Surveys*, vol. 9, no. 3, pp. 173-204, Sept. 1977.

[23] H. Hellerman and T. F. Conroy, *Computer System Performance*. New York: McGraw-Hill, 1975.

[24] Y. Ho, "Team decision theory and information structures," *Proc. IEEE*, vol. 68, no. 6, pp. 644-654, June 1980.

[25] E. D. Jensen, "The Honeywell experimental distributed processor—An overview," *Computer*, vol. 11, pp. 28-38, Jan. 1978.

[26] A. K. Jones *et al.*, "StarOS, a multiprocessor operating system for the support of task forces," in *Proc. 7th Symp. Operating System Prin.*, Dec. 1979, pp. 117-127.

[27] D. Klappholz and H. C. Park, "Parallelized process scheduling for a tightly-coupled MIMD machine," in *1984 Int. Conf. Parallel Proc.*, Aug. 1984, pp. 315-321.

[28] L. Kleinrock, *Queuing Systems, Vol. 2: Computer Applications*. New York: Wiley, 1976.

[29] L. Kleinrock and A. Nilsson, "On optimal scheduling algorithms for time-shared systems," *J. ACM*, vol. 28, no. 3, pp. 477-486, July 1981.

[30] C. P. Kruskal and A. Weiss, "Allocating independent subtasks on parallel processors extended abstract," in *1984 Int. Conf. Parallel Proc.*, Aug. 1984, pp. 236-240.

[31] R. E. Larsen, *Tutorial: Distributed Control*. New York: IEEE Press, 1979.

[32] G. Le Lann, *Motivations, Objectives and Characterizations of Distributed Systems* (Lecture *Notes in Computer Science*, Vol. 105). New York: Springer-Verlag, 1981, pp. 1-9.

[33] B. W. Lindgren, *Elements of Decision Theory*. New York: MacMillan, 1971.

[34] V. M. Lo, "Heuristic algorithms for task assignment in distributed systems," in *Proc. 4th Int. Conf. Dist. Comp. Systems*, May 1984, pp. 30-39.

[35] P. Y. R. Ma, E. Y. S. Lee, and J. Tsuchiya, "A task allocation model for distributed computing systems," *IEEE Trans. Comput.*, vol. C-31, no. 1, pp. 41-47, Jan. 1982.

[36] R. Manner, "Hardware task/processor scheduling in a polyprocessor environment," *IEEE Trans. Comput.*, vol. C-33, no. 7, pp. 626-636, July 1984.

[37] P. L. McEntire, J. G. O'Reilly, and R. E. Larson, *Distributed Computing: Concepts and Implementations*. New York: IEEE Press, 1984.

[38] E. Mishkin and L. Braun Jr., *Adaptive Control Systems*. New York: McGraw-Hill, 1961.

[39] K. Narendra, "Learning automata—A survey," *IEEE Trans. Syst., Man, Cybern.*, vol. SMC-4, no. 4, pp. 323-334, July 1974.

[40] L. M. Ni and K. Hwang, "Optimal load balancing strategies for a multiple processor system," in *Proc. Int. Conf. Parallel Proc.*, 1981, pp. 352-357.

[41] L. M. Ni and K. Abani, "Nonpreemptive load balancing in a class of local area networks," in *Proc. Comp. Networking Symp.*, Dec. 1981, pp. 113-118.

[42] L. M. Ni, K. Hwang, "Optimal load balancing in a multiple processor system with many job classes," *IEEE Trans. Software Eng.*, vol. SE-11, no. 5, pp. 491-496, May 1985.

[43] N. J. Nilsson, *Principles of Artificial Intelligence*. Palo Alto, CA: Tioga, 1980.

[44] J. Ousterhout, D. Scelza, and P. Sindhu, "Medusa: An experiment in distributed operating system structure," *Commun. ACM*, vol. 23, no. 2, pp. 92-105, Feb. 1980.

[45] G. Popek *et al.*, "LOCUS: A network transparent, high reliability distributed system," in *Proc. 8th Symp. O.S. Principles*, Dec. 1981, pp. 169-177.

[46] M. L. Powell and B. P. Miller, "Process migration in DEMOS/MP," in *Proc. 9th Symp. Operating Systems Principles* (OS Review), vol. 17, no. 5, pp. 110-119, Oct. 1983.

[47] C. C. Price and S. Krishnaprasad, "Software allocation models for distributed computing systems," in *Proc. 4th Int. Conf. Dist. Comp. Systems*, May 1984, pp. 40-48.

[48] C. Shen and W. Tsai, "A graph matching approach to optimal task assignment in distributed computing systems using a minimax criterion," *IEEE Trans. Comput.*, vol. C-34, no. 3, pp. 197-203, Mar. 1985.

[49] R. G. Smith, "The contract net protocol: High-level communication and control in a distributed problem solver," *IEEE Trans. Comput.*, vol. C-29, no. 12, pp. 1104-1113, Dec. 1980.

[50] M. H. Solomon and R. A. Finkel, "The ROSCOE distributed operating system," in *Proc. 7th Symp. O.S. Principles*, Dec. 1979, pp. 108-114.

[51] J. A. Stankovic *et al.*, "An evaluation of the applicability of different mathematical approaches to the analysis of decentralized control algorithms," in *Proc. IEEE COMPSAC 82*, Nov. 1982, pp. 62-69.

[52] J. A. Stankovic and I. S. Sidhu, "An adaptive bidding algorithm for processes, clusters and distributed groups," in *Proc. 4th Int. Conf. Dist. Comp. Systems*, May 1984, pp. 49-59.

[53] J. A. Stankovic, "Simulations of three adaptive, decentralized controlled, job scheduling algorithms," *Comput. Networks*, vol. 8, no. 3, pp. 199-217, June 1984.

[54] ——, "A perspective on distributed computer systems," *IEEE Trans. Comput.*, vol. C-33, no. 12, pp. 1102-1115, Dec. 1984.

[55] J. A. Stankovic, "An application of Bayesian decision theory to decentralized control of job scheduling," *IEEE Trans. Comput.*, vol. C-34, no. 2, pp. 117-130, Feb. 1985.

[56] J. A. Stankovic *et al.*, "A review of current research and critical issues in distributed system software," *IEEE Comput. Soc. Distributed Processing Tech. Committee Newslett.*, vol. 7, no. 1, pp. 14-47, Mar. 1985.

[57] H. S. Stone, "Critical load factors in two-processor distributed systems," *IEEE Trans. Software Eng.*, vol. SE-4, no. 3, pp. 254-258, May 1978.

[58] H. S. Stone and S. H. Bokhari, "Control of distributed processes," *Computer*, vol. 11, pp. 97-106, July 1978.

[59] H. Sullivan and T. Bashkow, "A large-scale homogeneous, fully distributed machine—II," in *Proc. 4th Symp. Computer Architecture*, Mar. 1977, pp. 118-124.

[60] A. S. Tanenbaum, *Computer Networks*. Englewood Cliffs, NJ: Prentice-Hall, 1981.

[61] D. P. Tsay and M. T. Liu, "MIKE: A network operating system for the distributed double-loop computer network," *IEEE Trans. Software Eng.*, vol. SE-9, no. 2, pp. 143-154, Mar. 1983.

[62] D. C. Tsichritzis and P. A. Bernstein, *Operating Systems*. New York: Academic, 1974.

[63] K. Vairavan and R. A. DeMillo, "On the computational complexity of a generalized scheduling problem," *IEEE Trans. Comput.*, vol. C-25, no. 11, pp. 1067-1073, Nov. 1976.

[64] R. A. Wagner and K. S. Trivedi, "Hardware configuration selection through discretizing a continuous variable solution," in *Proc. 7th IFIP Symp. Comp. Performance Modeling, Measurement and Evaluation*, Toronto, Canada, 1980, pp. 127-142.

[65] Y. T. Wang and R. J. T. Morris, "Load sharing in distributed systems," *IEEE Trans. Comput.*, vol. C-34, no. 3, pp. 204-217, Mar. 1985.

[66] M. O. Ward and D. J. Romero, "Assigning parallel-executable, intercommunicating subtasks to processors," in *1984 Int. Conf. Parallel Proc.*, Aug. 1984, pp. 392-394.

[67] L. D. Wittie and A. M. Van Tilborg, "MICROS: A distributed operating system for MICRONET, a reconfigurable network computer," *IEEE Trans. Comput.*, vol C-29, no. 12, pp. 1133–1144, Dec. 1980.

Thomas L. Casavant (S'85–'86) received the B.S. degree in computer science and the M.S. and Ph.D. degrees in electrical and computer engineering from the University of Iowa, Iowa City, in 1982, 1983, and 1986, respectively.

He is currently an Assistant Professor of Electrical Engineering at Purdue University, West Lafayette, IN. His research interests include computer architecture, operating systems, distributed systems, and performance analysis.

Dr. Casavant is a member of the IEEE Computer Society and the Association for Computing Machinery.

Jon G. Kuhl (S'76–M'79) received the M.S. degree in electrical and computer engineering and the Ph.D. degree in computer science from the University of Iowa, Iowa City, in 1977 and 1980, respectively.

He is an Associate Professor in the Department of Electrical and Computer Engineering at the University of Iowa. His primary research interests are in distributed systems, parallel processing, and fault-tolerant computing. His other research interests include computer architecture, graph theory, and computer communications.

Multiprocessor Scheduling with the Aid of Network Flow Algorithms

HAROLD S. STONE, MEMBER, IEEE

Abstract—In a distributed computing system a modular program must have its modules assigned among the processors so as to avoid excessive interprocessor communication while taking advantage of specific efficiencies of some processors in executing some program modules. In this paper we show that this program module assignment problem can be solved efficiently by making use of the well-known Ford-Fulkerson algorithm for finding maximum flows in commodity networks as modified by Edmonds and Karp, Dinic, and Karzanov. A solution to the two-processor problem is given, and extensions to three and n-processors are considered with partial results given without a complete efficient solution.

Index Terms—Computer networks, cutsets, distributed computers, Ford-Fulkerson algorithm, load balancing, maximum flows.

I. Introduction

DISTRIBUTED processing has been a subject of recent interest due to the availability of computer networks such as the Advanced Research Projects Agency Network (ARPANET), and to the availability of microprocessors for use in inexpensive distributed computers. Fuller and Siewiorek [8] characterize distributed computer systems in terms of a coupling parameter such that array computers and multiprocessors are tightly coupled, and computers linked through ARPANET are loosely coupled. In loosely coupled systems the cost of interprocessor communication is sufficiently high to discourage the execution of a single program distributed across several computers in the network. Nevertheless, this has been considered for ARPANET by Thomas and Henderson [17], Kahn [12], and Thomas [16].

In this paper we focus on the type of distributed system that Fuller and Siewiorek treat as systems with intermediate coupling. A primary example of this type of system is the multiprocessor interface message processor for ARPANET designed by Bolt, Beranek, and Newman (Heart *et al.* [10]). Two-processor distributed systems are widely used in the form of a powerful central processor connected to a terminal-oriented satellite processor. Van Dam [18] and Foley *et al.* [7] describe two-processor systems in which program modules may float from processor to processor at load time or during the execution of a program. The ability to reassign program modules to different processors in a distributed system is essential to make the best use of system resources as programs change from one computation phase to another or as system load changes.

Manuscript received November 7, 1975; revised July 9, 1976. This work was supported in part by the National Science Foundation under Grant DCR 74-20025.

The author is with the Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, MA 01002.

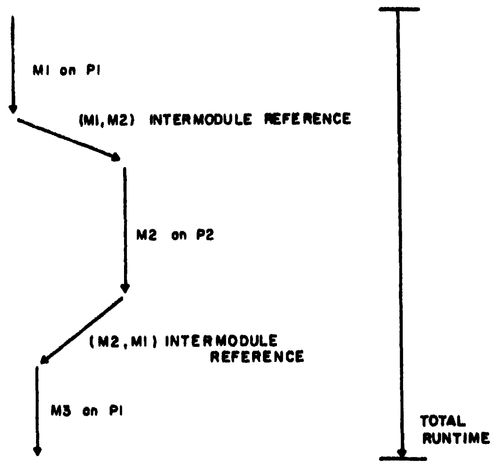
In this paper we treat the problem of assigning program modules to processors in a distributed system. There are two kinds of costs that must be considered in such an assignment. In order to reduce the cost of interprocessor communication, the program modules in a working set should be coresident in a single processor during execution of that working set. To reduce the computational cost of a program, program modules should be assigned to processors on which they run fastest. The two kinds of assignments can be incompatible. The problem is to find an assignment of program modules to processors that minimizes the collective costs due to interprocessor communication and to computation. We show how to make such an assignment efficiently by using methods of Ford and Fulkerson [6], Dinic [3], Edmonds and Karp [4], and Karzanov [13] that have been developed for maximizing flows in commodity networks. The two-processor assignment problem can be stated as a commodity flow problem with suitable modifications. We show how to construct a graph for the n-processor problem for which a minimal cost cut is a minimal cost partition of the graph into n disjoint subgraphs. We give partial results for finding the minimal cost cut but, unfortunately, an efficient solution has not yet been obtained.

In Section II of this paper we discuss the computer model in more detail. Section III reviews the essential aspects of commodity flow problems. The main results of this paper appear in Section IV, in which we show how to solve the two-processor problem, and in Section V, in which we consider the n-processor problem. A brief summary with an enumeration of several questions for future research appears in the final section.

II. The Multiprocessor Model

We use a model of a multiprocessor based on the multiprocessor system at Brown University [18], [15]. In this model, each processor is an independent computer with full memory, control, and arithmetic capability. Two or more processors are connected through a data link or high-speed bus to create a multiprocessor system. The processors need not be identical; in fact, the system cited above has a System/360 computer linked to a microprogrammable minicomputer. This much of the description of the model fits many systems. The distinguishing feature of the multiprocessor under discussion is the manner in which a program executes. In essence, each program in this model is a serial program, for which execution can shift dynamically from processor to processor. The two processors may both be multiprogrammed, and may execute concurrently on different programs, but may not execute concurrently on the same program.

A program is partitioned into functional modules some of



M1 on P1
(M1, M2) INTERMODULE REFERENCE
M2 on P2
(M2, M1) INTERMODULE REFERENCE
M3 on P1
TOTAL RUNTIME

Fig. 1. Module $M1$ calls module $M2$ on a distributed computer system.

which are assigned to particular processors, the remainder of which are permitted to "float" from processor to processor during program execution. Some modules have a fixed assignment because these modules depend on facilitites within their assigned processor. The facilities might be high-speed arithmetic capability, access to a particular data base, the need for a large high-speed memory, access to a specific peripheral device, or the need for any other facility that might be associated with some processor and not with every processor.

When program execution begins, the floating modules are given temporary assignments to processors, assignments that reflect the best guess as to where they should reside for maximum computation speed. As execution progresses, the floating modules are free to be reassigned so as to improve execution speed and to reflect the changing activity of the program. Here we presume that interprocessor communication incurs relatively high overhead, whereas computations that make no calls to other processors incur zero additional overhead. Fig. 1 shows the sequence of events that occur when module $M1$ on processor $P1$ calls module $M2$ on processor $P2$. The program state shifts from processor $P1$ to processor $P2$, and returns to processor $P1$ when module $M2$ returns control to $M1$. Instead of transferring state to $P2$, it may be better to colocate $M1$ and $M2$ on the same computer. The choice depends on the number of calls between $M1$ and $M2$, and on the relationship of $M2$ and $M1$ to other modules.

The advantage of this type of approach over other forms of multiprocessing is that it takes advantage of program locality. When activity within a program is within one locality, in ideal circumstances all the activity can be forced to reside within one processor, thereby eliminating conflicts due to excessive use of shared resources. The locality need not be known beforehand, because program modules tend to float to processors in which their activity is highest. Another type of multiprocessor is exemplified by C.mmp (Wulf and Bell [19]) in which up to 16 minicomputers and 16 memories are linked together through a crossbar. The coupling among processors in this system is very tight because potentially every memory fetch accesses memory through the crossbar. Conflicts are potentially high because two or more processors may attempt

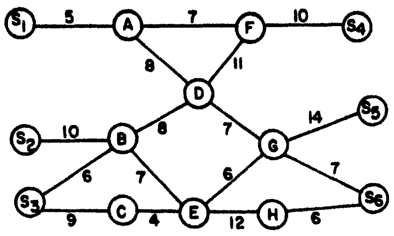


Fig. 2. Commodity flow network.

to access the same memory module simultaneously for an extended period of time.

In the present model, we eliminate the costly crossbar, the degradation on every memory access due to delays through the crossbar, and the potential for conflicts on every memory access to further degrade memory access. If this model is more effective than the crossbar architecture of C.mmp, it is because the costly interprocessor calls incurred in practice occur rarely due to the ability to assign program working sets to a single processor. Whether or not this is the case has not been settled. The question of the relative effectiveness of this architecture and the crossbar architecture has barely been investigated at this writing. The memory assignment algorithm described in the following sections indicates that it is possible to control a distributed computer of the type under study here reasonably efficiently, thereby opening the door to further and deeper studies of this idea.

III. Maximum Network Flows and Minimum Cutsets

In this section we review the commodity flow problem for which Ford and Fulkerson formulated a widely used algorithm [6]. The algorithm is reasonably fast in practice, but the original formulation of the algorithm is subject to rather inefficient behavior in pathological cases. Edmonds and Karp [4] have suggested several different modifications to the Ford-Fulkerson algorithm so as to eliminate many of the inefficiencies, at least for the pathological examples, and to obtain improved worst-case bounds on the complexity of the algorithm. Dinic [3] and Karzanov [13] have derived other ways to increase speed.

In this section we shall describe the problem solved by the Ford-Fulkerson algorithm as modified by Edmonds and Karp, but we shall refer the reader to the literature for descriptions of the various implementations of the algorithm. The important idea is that we can treat an algorithm for the solution of the maximum flow problem as a "black box" to solve the module assignment problem. We need not know exactly what algorithm is contained in the black box, provided that we know the implementation is computationally efficient for our purposes.

The maximum flow problem is a problem involving a commodity network graph. Fig. 2 shows the graph of such a network.

The nodes labeled S_1, S_2, and S_3 are source nodes, and the nodes labeled S_4, S_5, and S_6 are sink nodes. Between these subsets of nodes lie several interior nodes with the entire collection of nodes linked by weighted branches. Source nodes represent production centers, each theoretically capable

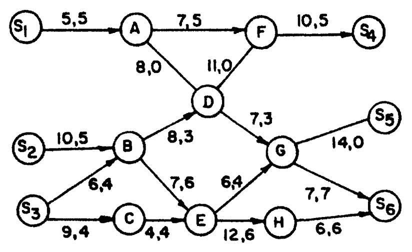


Fig. 3. Flow in a commodity flow network.

of producing an infinite amount of a specific commodity. Sink nodes represent demand centers, each of which can absorb an infinite amount of the commodity. The branches represent commodity transport linkages, with the weight of a branch indicating the capacity of the corresponding link.

A *commodity flow* in this network is represented by weighted directed arrows along the branches of the network, with the weight of the arrow indicating the amount of the flow on that branch, and the direction of the arrow indicating the direction of the commodity flow. Fig. 3 shows a commodity flow for the graph in Fig. 2. Each arrow in Fig. 3 carries a pair of numbers, the first of which is the capacity of that branch and the second of which is the actual flow on that branch. A *feasible* commodity flow in this network is a commodity flow originating from the source nodes and ending at the sink nodes such that: 1) at each intermediate node, the sum of the flows into the node is equal to the sum of the flows out of the node; 2) at each sink node the net flow into the nodes is nonnegative, and at each source node the net flow directed out of the node is nonnegative; and 3) the net flow in any branch in the network does not exceed the capacity of that branch. Note that the flow in Fig. 3 is a feasible flow according to this definition. The three constraints in this definition guarantee that interior nodes are neither sources nor sinks, that source nodes and sink nodes are indeed sources and sinks, respectively, and that each branch is capable of supporting the portion of the flow assigned to it.

The *value* of a commodity flow is the sum of the net flows out of the source nodes of the network. Because the net flow into the network must equal the net flow out of the network, the value of a commodity flow is equal to the sum of net flows into the sink nodes. The value of the flow in Fig. 3 is 18. A *maximum flow* is a feasible flow whose value is maximum among all feasible flows. Fig. 4 shows a maximum flow for the network in Fig. 2.

The maximum flow in Fig. 4 is related to a cutset of the network. A *cutset* of a commodity network is a set of edges which when removed disconnects the source nodes from the sink nodes. No proper subset of a cutset is a cutset. Note that branches (S_1, A), (B, E), (B, D), and (C, E) form a cutset of the graph in Fig. 4. The *weight* of a cutset is equal to the sum of the capacities of the branches in the cutset. The weight of the above cutset is 24, which is equal to the value of the maximum flow. This is not a coincidence, because central to the Ford-Fulkerson algorithm is the following theorem.

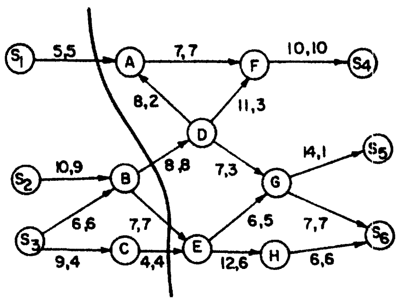


Fig. 4. Maximum flow and minimum cut in a commodity flow network.

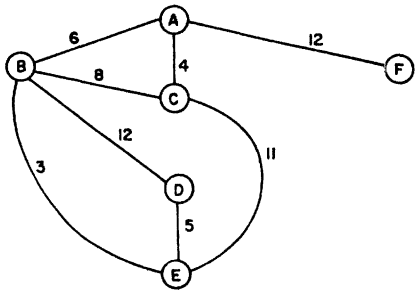


Fig. 5. Intermodule-connection graph.

Max-flow, Min-cut Theorem (Ford and Fulkerson [6]): The value of a maximum flow in a commodity network is equal to the weight of a minimum weight cutset of that network.

The proof of this theorem and a good tutorial description of algorithms for finding a maximum flow and minimum weight cutset appear in Ford and Fulkerson [6]. A very clear treatment of both the Ford-Fulkerson algorithm and the Edmonds-Karp modifications appear in Even [5]. At this point we note that the commodity flow problem as described can be solved in a time bounded from above by the fifth power of the number of nodes in the graph. All of the variations of the maximum flow algorithm of interest compute both a maximum flow in the graph and find a minimum weight cutset. We shall make use of this fact in solving the module assignment problem because each possible assignment corresponds to a cutset of a commodity graph, and the optimum assignment corresponds to a minimum weight cutset.

IV. The Solution to the Two-Processor Assignment Problem

In this section we show how the maximum-flow algorithm finds an optimal partition of a modular program that runs on a two-processor system. In the following section we show how this algorithm generalizes to systems with three or more processors.

To use the maximum flow algorithm we develop a graphical model of a modular program. Fig. 5 shows a typical example in which each node of the graph represents a module and the branches of the graph represent intermodule linkages. The modules should be viewed as program segments which either contain executable instructions plus local data or contain global data accessible to other segments. The weights on the

TABLE I

Module	P_1 Run Time	P_2 Run Time
A	5	10
B	2	∞
C	4	4
D	6	3
E	5	2
F	∞	4

branches indicate the cost of intermodule references when the modules are assigned to different computers. We assume that the cost of an intermodule reference is zero when the reference is made between two modules assigned to the same computer.

The cost function used to compute the branch weights may vary depending on the nature of the system. Initially we choose to minimize the absolute running time of a program, without permitting dynamic reassignments. We modify the constraints later. The weight of a branch under this assumption is the total time charged to the intermodule references represented by the branch. Thus if k references between two modules occur during the running of a program and each reference takes t seconds when the modules are assigned to different computers, then the weight of the branch representing these references is kt.

The graph in Fig. 5 captures the notion of the costs for crossing boundaries between processors. This is only part of the assignment problem. We must also consider the relative running time of a module on different processors. One processor may have a fast floating-point unit, or a faster memory than another processor, and this may bias the assignment of modules. Table I gives the cost of running the modules of the program in Fig. 5 on each of two processors. The symbol ∞ in the table indicates an infinite cost, which is an artifice to indicate that the module cannot be assigned to the processor. Since our objective in this part of the discussion is to minimize the total absolute running time of a program, the costs indicated in Table I give the total running time of each module on each processor.

In this model of a distributed computing system, there is no parallelism or multitasking of module execution within a program. Thus the total running time of a program consists of the total running time of the modules on their assigned processors as given in Table I plus the cost of intermodule references between modules assigned to different processors. Note that an optimum assignment must take into consideration both the intermodule reference costs and the costs of running the modules themselves. For the running example, if we assume that B must be assigned to P_1, and F must be assigned to P_2, then the optimum way of minimizing the intermodule costs alone is to view B and F as source and sink, respectively, of a commodity-flow network. The minimum cut is the minimum intermodule cost cut, and this assigns B, C, D, and E to P_1, and A and F to P_2. On the other hand, an optimum way to minimize only the running time of the individual modules is to assign A, B, and C to P_1 and D, E, and F to P_2. But neither of these assignments minimize the total running time.

To minimize the total running time, we modify the module interconnection graph so that each cutset in the modified graph corresponds in a one-to-one fashion to a module assignment and the weight of the cutset is the total cost for that assignment. With this modification, we can solve a maximum flow problem on the modified graph. The minimum weight cutset obtained from this solution determines the module assignment, and this module assignment is optimal in terms of total cost.

We modify the module interconnection graph as follows.

1) Add nodes labeled S_1 and S_2 that represent processors P_1 and P_2, respectively. S_1 is the unique source node and S_2 is the unique sink node.

2) For each node other than S_1 and S_2, add a branch from that node to each of S_1 and S_2. The weight of the branch to S_1 carries the cost of executing the corresponding module on P_2, and the weight of the branch to S_2 carries the cost of executing the module on P_1. (The reversal of the subscripts is intentional.)

The modified graph is a commodity flow network of the general type exemplified by the graph in Fig. 2. Each cutset of the commodity flow graph partitions the nodes of the graph into two disjoint subsets, with S_1 and S_2 in distinct subsets. We associate a module assignment with each cutset such that if the cutset partitions a node into the subset containing S_1, then the corresponding module is assigned to processor P_1. Thus the cut shown in Fig. 6 corresponds to the assignment of A, B, and C to P_1 and D, E, and F to P_2.

With this association of cutsets to module assignments it is not difficult to see that module assignments and cutsets of the commodity flow graph are in one-to-one correspondence. (The one-to-one correspondence depends on the fact that each interior node is connected directly to a source and sink, for otherwise, a module assignment might correspond to a subset of edges that properly contains a cutset.) The following theorem enables us to use a maximum flow algorithm to find a minimum cost assignment.

Theorem: The weight of a cutset of the modified graph is equal to the cost of the corresponding module assignment.

Proof: A module assignment incurs two types of costs. One cost is from intermodule references from processor to processor. The other cost incurred is the cost of running each module on a specific processor. The cutset corresponding to a module assignment contains two types of branches. One type of branch represents the cost of intermodule references for modules in different processors, and a particular assignment. All costs due to such references contribute to the weight of the corresponding cutset, and no other intermodule references contribute to the weight of the cutset.

The second type of branch in a cutset is a branch from an internal node to a source or sink node. If an assignment places a module in processor P_1, then the branch between that node and S_2 is in the cutset, and this contributes a cost equal to the cost of running that module on P_1, because the

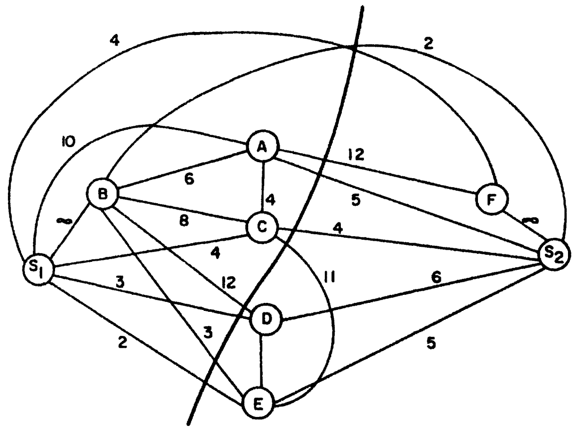


Fig. 6. Modified module-interconnection graph and a cut that determines a module assignment.

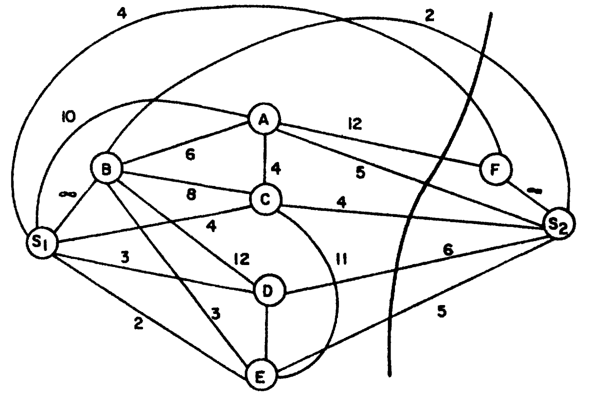


Fig. 7. Minimum cost cut.

branch to S_2 carries the P_1 running time cost. Similarly, the cost of assigning a module to P_2 is reflected by adding the cost of running that module on P_2 to the weight of the cutset.

Thus the weight of a cutset accounts for all costs due to its corresponding assignment, and no other costs contribute to the weight. This proves the theorem.

The theorem indicates that an optimal assignment can be found by running a maximum flow algorithm on the modified graph. A maximum flow algorithm applied to the graph shown in Fig. 6 produces the minimum weight cutset shown in Fig. 7. The corresponding module assignment is different from the assignment that minimizes the cost of intermodule references and the assignment that minimizes the individual running time costs, and its total cost is lower than the total cost of either of the two other assignments.

At this point the basic essentials for treating the module assignment problem as a maximum flow problem are clear. There remain a number of details concerning the practical implementation of the algorithm that merit discussion.

The running example indicates how to select an optimum *static* assignment that minimizes total running time. We mentioned earlier that it makes sense to change assignments dynamically to take advantage of local behavior of programs and the relatively infrequent changes in program locality. To solve the dynamic assignment problem we essentially have to solve a maximum flow problem at each point in time during which the working set of a program changes. Fortunately, the dynamic problem is no harder than the static one, except for the need to solve several maximum flow problems instead of a single one. The only additional difficulty in dynamic assignments is detecting a change in the working set of a program. Since a control program must intervene to perform intermodule transfers across processor boundaries, it should monitor such transfers, and use changes in the rate and nature of such transfers as a signal that the working set has changed. Thus we can reasonably expect to solve the dynamic assignment problem if it is possible to solve a static assignment problem for each working set.

The major difficulty in solving a static assignment problem is obtaining the requisite data for driving the maximum flow algorithm. It is not usually practical to force the user to supply these data, nor is it completely acceptable to use compiler generated estimates. Some of the data can be gathered during program execution. The cost of each invocation of a module on a particular processor can be measured by a control program, and it usually measures this anyway as part of the system accounting. However, we need to know the cost of running a module on each processor to compute an optimal assignment, and we certainly cannot ship a module to every other processor in a system for a time trial to determine its relative running time.

A reasonable approximation is to assume that the running time of a module on processor P_1 is a fixed constant times the running time of that module on processor P_2 where the constant is determined in advance as a measure of the relative power of the processors without taking into consideration the precise nature of the program to be executed. Under these assumptions if we can gather data about intermodule references, we can obtain sufficient data to drive the maximum flow algorithm. If after making one or more assignments a module is executed on different computers, sufficient additional information can be obtained to refine initial estimates of relative processor performance. The refined data should be used in determining new assignments as the data are collected.

How do we collect data concerning intermodule references? Here the collection of data follows a similar philosophy as for running time measurements. Initially an analysis of the static program should reveal where the intermodule references exist, and these in turn can be assumed to be of equal weight to determine an initial assignment. We assume that we automatically measure the intermodule references across processor boundaries because all such references require control program assistance. In measuring these references we obtain new data that refine the original estimates. If as a result of the refinement of these data we reassign modules, then we obtain new data about intermodule links that further refine the data, which in turn permit a more accurate appraisal of a minimum cost assignment.

At this writing the control methodology described here has been implemented by J. Michel and R. Burns on the system described by Stabler [15] and van Dam [18]. That system gathers the requisite statistics in real time in a suitable form

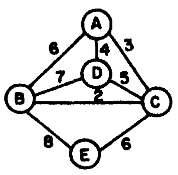


Fig. 8. Module-interconnection graph.

for input to the algorithm. It is too early to say how effective the statistics gathering and automatic reassignment processes are in performing load balancing, but it is safe to say that the ability to gather suitable statistics automatically has been demonstrated.

In closing this section we take up one last subject, that of considering objective functions other than total running time. There are many suitable cost functions one may wish to use. For example, instead of absolute time, one may choose to minimize dollars expended. For this objective function, the intermodule reference costs are measured in dollars per transfer, and the running time costs of each module are measured in dollars for computation time on each processor, taking into account the relative processor speeds and the relative costs per computation on each processor. Many other useful objective functions can be met by choosing the cost function appropriately. We should also point out that generalizations of the maximum flow algorithm are applicable to module assignment problems under more complex cost measures. The most notable maximum flow problem generalization is the selection of a maximum flow with minimum cost. For this problem each flow is associated with both a flow value and a cost. The algorithm selects among several possible maximum flows to return the one of minimum cost. The equivalent problem for the module assignment problem is to find an assignment that achieves fastest computation time and incurs the least dollar cost of all such assignments.

The fact that the two-processor assignment problem is mapped into a commodity flow problem for solution suggests that the flow maximized in the commodity flow problem corresponds to some physical entity flowing between the two-processors in the two-processor assignment problem. There is no such correspondence, however, since the maximal flow value corresponds to time in a two-processor assignment, and not to information flow.

V. Extension to Three or More Processors

In this section we show how the module assignments to three or more processors can be accomplished by using the principles described in the previous section. We first obtain a suitable generalization of the notion of cutset. Then we describe a procedure to construct a modified graph of a program such that its cutsets are in one-to-one correspondence with the multiprocessor cutsets, and the value of each cutset is equal to the cost of the corresponding assignment. Finally we consider how to solve the generalized multiprocessor flow problem, and obtain partial results but no efficient solution.

Let us take as a running example the three-processor program shown in Fig. 8 with the running times for each processor given in Table II. Since we have to assign the nodes to three rather than to two processors, we need to generalize the notion of cutset. Let us designate the source and sink nodes of a commodity network to be distinguished nodes, and we say they are distinguished of type *source* or type *sink*. For the n-processor problem we shall have n types of distinguished nodes. This leads naturally to the following definition.

TABLE II

Module	P_1 Time	P_2 Time	P_3 Time
A	4	∞	∞
B	∞	6	∞
C	∞	∞	7
D	10	7	5
E	4	7	3

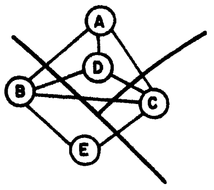


Fig. 9. Cut representing a module assignment to three processors.

Definition: A *cutset* in a commodity flow graph with n types of nodes is a subset of edges that partitions the nodes of the graph into n disjoint subsets, each of which contains all of the distinguished nodes of a single type plus a possibly empty collection of interior nodes. No proper subset of a cutset is also a cutset.

A cutset for the network of Fig. 8 appears in Fig. 9. We shall deal with networks in which there is a single distinguished node of each type, and this node represents the processor to which the interior nodes associated with it are assigned.

Proceeding as before we modify the intermodule reference graph of a program to incorporate the relative running time costs of each module on each processor. Again we add a branch from each interior node to each distinguished node as in the two-processor case. The weight on each such branch is computed by a formula explained below and exemplified in Fig. 10. The weights are selected so that, as in the two-processor case, the value of a cutset is equal to total running time.

For simplicity Fig. 10 does not show the branches from nodes A, B, and C to nodes to which they are not assigned. Suppose that module D runs in time T_i on processor P_i, $i = 1, 2, 3$. Then the branch from node D to distinguished node S_1 carries the weight $(T_2 + T_3 - T_1)/2$, and likewise the branches to nodes S_2 and S_3 carry the weights $(T_1 + T_3 - T_2)/2$, and $(T_1 + T_2 - T_3)/2$, respectively. Under this weight assignment, if D is assigned to processor P_1, the arcs to S_2 and S_3 are cut, and their weights total to T_1, the running time of D on P_1.

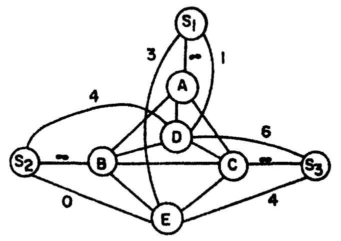


Fig. 10. Modified module-interconnection graph for a three-processor assignment.

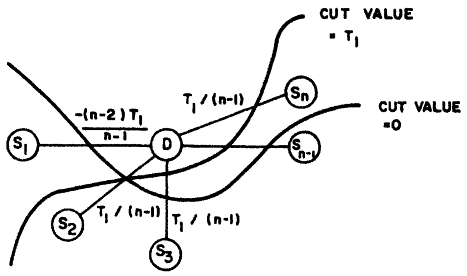


Fig. 11. Two possible assignments of module D in an n-processor system.

This idea generalizes naturally to n-processors. The running time of D on P_1 contributes to the weight of the branches from D to every distinguished node S_i, $i = 1, 2, \cdots, n$. Its contribution to the branch to S_1 is $-(n - 2)T_1/(n - 1)$. Its contribution to the branch to S_i, $i \neq 1$, is $T_1/(n - 1)$. If D is assigned to S_1, then $n - 1$ branches are cut, each of which contributes $T_1/(n - 1)$ to the cutset, giving a net contribution of T_1. If D is assigned to S_i, $i \neq 1$, then n-2 branches contribute $T_1/(n\text{-}1)$ to the cutset weight and the branch to S_1 contributes $-(n\text{-}2)T_1/(n\text{-}1)$ for a net contribution of zero. This is shown graphically in Fig. 11. Consequently, under the graph modification scheme described here, we obtain the desired property that the weight of a cutset is equal to the cost of the corresponding module assignment.

One problem with this scheme is that some individual edges of the graph may have a negative capacity, and this cannot be treated by maximum flow algorithms. This problem can easily be overcome, however. Suppose that among the branches added to node D is an arc with negative weight $-W$, and this is the most negative of the weights of the branches added to node D. Then increase the weights of all branches added to D by the amount $W + 1$, and we claim that no branch has negative weight after this change. Moreover, every cut that isolates D from a distinguished node breaks precisely $(n - 1)$ of the branches added to D, contributing a value to the cutset of $(n - 1)(W + 1)$ plus the run time of D on the processor corresponding to the cut. The term $(n - 1)(W + 1)$ is a constant independent of the assignment so that an assignment that minimizes the cutset weight in the present graph is an assignment that finds a minimum time solution of the original problem.

At this point we come to the question of finding a minimum cutset in an n-processor graph. The solution can be found by exhaustive enumeration but the computational complexity of such an approach is quite unattractive. It seems natural that the n-processor problem can be reduced to several two-processor flow problems, and can be attacked efficiently in this way. In the remainder of this section we show that a two-processor flow can give information about the minimal cut in an n-processor graph, but we are unable to produce a complete efficient algorithm.

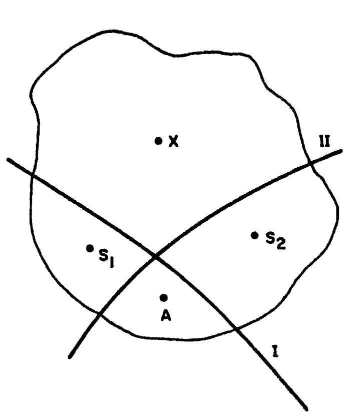


Fig. 12. Two cutsets in a graph.

Specifically, consider what happens when we run a two-processor flow from S_1 to the subset $\{S_i: 2 \leq i \leq n\}$, where the nodes in the subset are all sink nodes for the flow. This two-processor flow produces a cutset that associates some nodes with S_1, and the remainder of the nodes in the graph to the subset of $n - 1$ distinguished nodes. Does this flow give any information about the minimum cut in the n-processor graph? Indeed it does, as is shown by the following theorem. The proof technique is similar to the proof technique used by Gomory and Hu [9] and Hu [11].

Theorem: Let node A be associated with distinguished node S_1 by a two-processor flow algorithm. Then A is associated with S_1 in a minimum cost partition of the n-processor graph.

Proof: Without loss of generality, suppose that A is associated with S_1 by the two-processor flow algorithm, and that it must be associated with S_2 in a minimum cost partition. We prove the theorem by showing that A can be moved from S_2 to S_1 in the minimum cost partition without altering the cost of the partition. Fig. 12 shows the cutset I that associates A with S_1 when the two-processor flow is run, and the cutset II that associates A with S_2 in the minimum cost partition. These cutsets cross each other, thus dividing the graph into four disjoint regions. Let us denote the four regions as S_1, S_2, A, and X according to the nodes appearing in these regions in Fig. 12. (Region X may be empty, but this will not upset the proof.) Let $c(U, V)$ denote the sum of the weights of all branches between two regions U and V.

Since II is a minimal cut the value of II does not exceed the value of a cut that fails to include A with S_2. Thus,

$$c(S_2, X) + c(S_2, S_1) + c(A, X) + c(A, S_1) \leq c(S_2, X) + c(S_2, S_1) + c(S_2, A). \tag{1}$$

From this we find that $c(A, S_1) \leq c(S_2, A) - c(A, X)$, and since all costs are nonnegative we may add $2c(A, X)$ to the right-hand side of the inequality and obtain

$$c(A,S_1) \leq c(S_2,A) + c(A,X). \tag{2}$$

By hypothesis the algorithm associates node A with S_1, with a cost for cut I equal to $c(S_1,X) + c(S_1,S_2) + c(A,X) + c(S_2,A)$. However, if cut I veers slightly to exclude node A, and otherwise remains the same, the cost of the new cut is $c(S_1,X) + c(S_1,S_2) + c(A,S_1)$. Since I is a minimal cost cut, we must have

$$c(A,S_1) \geq c(S_2,A) + c(A,X). \tag{3}$$

From (2) and (3) we find that the inequality in (3) can be changed to equality. Since (3) holds with equality, by substituting (3) into (1) we find $2c(A,X) \leq 0$, which must hold with equality since all costs are nonnegative. Then (1) holds with equality. Thus cut II can be altered to exclude node A from S_2's subset and include A with S_1's subset without changing the cost of partition. This proves the theorem.

The previous theorem suggests that a two-processor algorithm may be used several times to find a minimum n-processor cutset. There are some difficulties in extending the theorem, however, that have left the n-processor problem still unsolved. Among the difficulties are the following.

1) The theorem states that a node associated with a distinguished node by a two-processor flow belongs with that node in the minimum n-processor cutset. Unfortunately, it is easy to construct examples in which a node that belongs with a particular distinguished node in a minimum n-processor cutset fails to be associated with that node by a two-processor flow.

2) Suppose a two-processor flow is run that results in the assignment of one or more nodes to a particular distinguished node. Let these nodes and the corresponding distinguished node be removed from the graph, and run a new two-processor flow from some other distinguished node S_i to all of the other distinguished nodes. In the cut found by this algorithm the nodes associated with S_i need not be associated with S_i in a minimum cost n-processor partition. In other words, the theorem does not apply when graph reduction is performed.

We conjecture that at most n^2 two-processor flows are necessary to find the minimum cost partition for n-processor problems, since there are only $n(n-1)/2$ different flows from one distinguished node to another distinguished node. These flows should somehow contain all the information required to find a minimal cost n-processor partition. It is possible that only n two-processor flows are required to solve the problem since Gomory and Hu [9] have shown that there are only $n-1$ independent two-terminal flows in an n-terminal network. This problem remains open at present.

VI. Summary and Conclusions

The two algorithms presented here provide for the assignment of program modules to two processors to minimize the cost of a computation on a distributed computer system. The algorithm uses a maximum flow algorithm as a subroutine so the complexity of the module assignment is dependent upon the implementation of the maximum flow algorithm used. Fortunately, the maximum flow algorithm is generally efficient and there are various modifications of the algorithm that take advantage of special characteristics of the module to obtain increased efficiency (see Dinic [3], Karzanov [13]). To obtain truly optimal assignments, the costs for intermodule transfers and relative running times have to be known. However, if good estimates are available, these can be used to obtain near-optimal assignments that are satisfactory in a pragmatic sense.

One may choose to use the assignment algorithm in a static sense, that is, to find one assignment that holds for the lifetime of a program, and incurs least cost. We believe it is more reasonable to reassign modules dynamically during a computation at the points where working sets change. Each dynamic assignment then is chosen to be optimal for a given working set. Dynamic identification of working sets and the identification of times at which a working set changes is still a subject of much controversy with proposals favoring particular schemes [2] or disfavoring those schemes [14]. Progress in this area will in turn lead to progress in the ability to make dynamic module assignments in a distributed processor system.

The model presented here is highly simplified and idealized, but it is useful in real systems. Foley *et al.* [7] can use our algorithms in place of their backtracking algorithms to do module reassignment in their distributed computer systems. We suspect that the maximum flow approach is more efficient than backtracking because worst-case performance of backtracking has a much greater complexity than maximum flow algorithm complexity. However, the actual performance of their algorithm may be quite different from worst-case performance, and could have a small average complexity, perhaps lower than the average complexity of maximum flow algorithms. No data on actual running times appears in Foley's report, so there is no information on which to base estimates of relative running times.

There are a number of open problems related to the research reported here. We mention just a few of them here.

1) If the minimum cost module assignment is not unique, then what additional criteria are useful in selecting the most desirable minimum cost assignment? Given such criteria, this form of the problem can be solved by using efficient algorithms to find a maximum flow of minimal cost. Such algorithms are described by Ford and Fulkerson [6] and Edmonds and Karp [4].

2) If a program is divided into tasks that can execute simultaneously under various precedence constraints, how can modules be assigned so as to minimize the cost of computation? This differs from the multiprocessor scheduling problem studied by Coffman and Graham [1] and by others in that there is no cost incurred in that model for interprocessor references.

3) If the various processors in distributed computer systems are each multiprogrammed, and queue lengths become excessive at individual processors, how might modules be reassigned to minimize costs of computation over several programs?

4) Given that a module reassignment incurs a processor-to-processor communication cost, how might the cost of

reassigning a module be factored into the module assignment problem?

Since distributed computer systems are still in early stages of development it is not clear which one of the research questions listed here will emerge to become important questions to solve for distributed computer systems as they come of age. The implementation of the methodology described here on the system at Brown University suggests that automatic load balancing among different processors can be done. We hope that the present and future research will show not only the possibility of load balancing but that it can be done efficiently and that load balancing is an efficient method for tapping the power of a distributed computer system.

Acknowledgment

The author is deeply indebted to Professor Andries van Dam for providing the inspiration and motivation for the research, and to J. Michel and R. Burns for implementing the ideas described here on the ICOPS system at Brown University. Discussions with Professor W. Kohler, S. Bokhari, and P. Jones provided additional stimulation that contributed to the research.

References

[1] E. G. Coffman, Jr., and R. L. Graham, "Optimal scheduling for two-processor systems," *Acta Informatica*, vol. 1, pp. 200-213, 1972.

[2] P. J. Denning, "Properties of the working set mode," *Commun. Ass. Comput. Mach.*, vol. 11, pp. 323-333, May 1968.

[3] E. A. Dinic, "Algorithm for solution of a problem of maximum flow in a network with power estimation," *Soviet Math. Doklady*, vol. 11, no. 5, pp. 1277-1280, 1970.

[4] J. Edmonds and R. M. Karp, "Theoretical improvements in algorithm efficiency for network flow problems," *J. Ass. Comput. Mach.*, vol. 19, pp. 248-264, Apr. 1972.

[5] S. Even, *Algorithmic Combinatorics*. New York: Macmillan, 1973.

[6] L. R. Ford, Jr., and D. R. Fulkerson, *Flows in Networks*. Princeton, NJ: Princeton Univ. Press, 1962.

[7] J. D. Foley *et al.*, "Graphics system modeling," Rome Air Development Center, Final Rep. Contract F30602-73-C-0249, Rep. RADC-TR-211, Aug. 1974.

[8] S. H. Fuller and D. P. Siewiorek, "Some observations on semiconductor technology and the architecture of large digital modules," *Computer*, vol. 6, pp. 14-21, Oct. 1973.

[9] R. E. Gomory and T. C. Hu, "Multiterminal network flows," *J. SIAM*, vol. 9, pp. 551-570, Dec. 1961.

[10] F. E. Heart *et al.*, "A new minicomputer/multiprocessor for the ARPA network," in *Proc. 1973 Nat. Comput. Conf., AFIPS Conf. Proc.*, vol. 42. Montvale, NJ: AFIPS Press, 1973.

[11] T. C. Hu, *Integer Programming and Network Flows*. Reading, MA: Addison-Wesley, 1970.

[12] R. E. Kahn, "Resource-sharing computer communications networks," *Proc. IEEE*, vol. 60, pp. 1397-1407, Nov. 1972.

[13] A. V. Karzanov, "Determining the maximal flow in a network by the method of preflows," *Soviet Math. Doklady*, vol. 15 no. 2, pp. 434-437, 1974.

[14] B. G. Prieve, "Using page residency to select the working set parameter," *Commun. Ass. Comput. Mach.*, vol. 16, pp. 619-620, Oct. 1973.

[15] G. M. Stabler, "A system for interconnected processing," Ph.D. dissertation, Brown Univ., Providence, RI, Oct. 1974.

[16] R. H. Thomas, "A resource sharing executive for the ARPANET," in *Proc. 1973 Nat. Comput. Conf., AFIPS Conf. Proc.*, vol. 42. Montvale, NJ: AFIPS Press, 1973.

[17] R. H. Thomas and D. Henderson, "McRoss–A multi-computer programming system," in *Proc. 1972 Spring Joint Comput. Conf., AFIPS Conf. Proc.*, vol. 40. Montvale, NJ: AFIPS Press, 1972.

[18] A. van Dam, "Computer graphics and its applications," Final Report, NSF Grant GJ-28401X, Brown University, May 1974.

[19] W. A. Wulf and C. G. Bell, "C.mmp–A multi-miniprocessor," in *Proc. 1972 Fall Joint Comput. Conf., AFIPS Conf. Proc.*, vol. 41, Part II. Montvale, NJ: AFIPS Press, 1972, pp. 765-777.

Harold S. Stone (S'61–M'63) received the B.S. degree from Princeton University, Princeton, NJ, in 1960 and the M.S. and Ph.D. degrees from the University of California, Berkeley, in 1961 and 1963, respectively.

While at the University of California he was a National Science Foundation Fellow and Research Assistant in the Digital Computer Laboratory. From 1963 until 1968 he was with Stanford Research Institute, and from 1968 until 1974 he was Associate Professor of Electrical and Computer Science at Stanford University. He is currently Professor of Electrical and Computer Engineering at the University of Massachusetts, Amherst. He has recently been engaged in research in parallel computation, computer architecture, and advanced memory system organization. In the past he has performed research in several areas including combinatorial algorithms, operating systems, and switching theory. He has authored over thirty technical publications, and is an author, coauthor, or editor of five text books in computer science. He has also been a Visiting Lecturer at the University of Chile, the Technical University of Berlin, and the University of Sao Paulo, and has held a NASA Research Fellowship at NASA Ames Research Center.

Dr. Stone is a member of Sigma Xi and Phi Beta Kappa. He has served as Technical Editor of *Computer*, and as a member of the Governing Board of the IEEE Computer Society.

Load Distributing for Locally Distributed Systems

Niranjan G. Shivaratri, Phillip Krueger, and Mukesh Singhal
Ohio State University

Load-distributing algorithms can improve a distributed system's performance by judiciously redistributing the workload among its nodes. This article describes load-distributing algorithms and compares their performance.

The availability of low-cost microprocessors and advances in communication technologies has spurred considerable interest in locally distributed systems. The primary advantages of these systems are high performance, availability, and extensibility at low cost. To realize these benefits, however, system designers must overcome the problem of allocating the considerable processing capacity available in a locally distributed system so that it is used to its fullest advantage.

This article focuses on the problem of judiciously and transparently redistributing the load of the system among its nodes so that overall performance is maximized. We discuss several key issues in load distributing for general-purpose systems, including the motivations and design trade-offs for load-distributing algorithms. In addition, we describe several load-distributing algorithms and compare their performance. We also survey load-distributing policies used in existing systems and draw conclusions about which algorithm might help in realizing the most benefits of load distributing.

Issues in load distributing

We first discuss several load-distributing issues central to understanding its intricacies. In this article, we use the terms computer, processor, machine, workstation, and node interchangeably.

Motivation. A *locally distributed system* consists of a collection of autonomous computers connected by a local area communication network. Users submit tasks at their host computers for processing. As Figure 1 shows, the random arrival of tasks in such an environment can cause some computers to be heavily loaded while other computers are idle or only lightly loaded. Load distributing improves performance by transferring tasks from heavily loaded computers, where service is poor, to lightly loaded computers, where the tasks can take advantage of

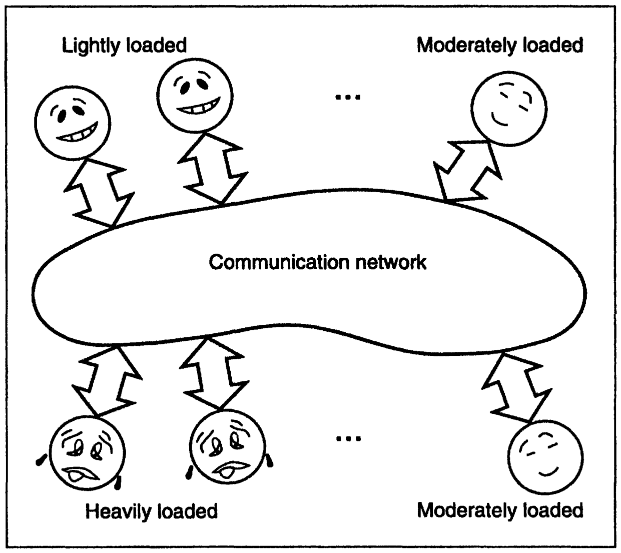


Figure 1. Distributed system without load distributing.

computing capacity that would otherwise go unused.

If workloads at some computers are typically heavier than at others, or if some processors execute tasks more slowly than others, the situation shown in Figure 1 is likely to occur often. The usefulness of load distributing is not so obvious in systems in which all processors are equally powerful and have equally heavy workloads over the long term. However, Livny and Melman[1] have shown that even in such a homogeneous distributed system, at least one computer is likely to be idle while other computers are heavily loaded because of statistical fluctuations in the arrival of tasks to computers and task-service-time requirements. Therefore, even in a homogeneous distributed system, system performance can potentially be improved by appropriate transfers of workload from heavily loaded computers (senders) to idle or lightly loaded computers (receivers).

What do we mean by performance? A widely used performance metric is the *average response time* of tasks. The response time of a task is the time elapsed between its initiation and its completion. Minimizing the average response time is often the goal of load distributing.

Dynamic, static, and adaptive algorithms. Load-distributing algorithms can be broadly characterized as dynamic, static, or adaptive. *Dynamic* load-distributing algorithms use system-state information (the loads at nodes), at least in part, to make load-distributing decisions, while *static* algorithms make no use of such information.

Decisions are hardwired in static load-distributing algorithms using a priori knowledge of the system. For example, under a simple "cyclic splitting" algorithm, each node assigns the ith task it initiates to node $i \bmod N$, where N is the number of nodes in the system. Alternatively, a probabilistic algorithm assigns a task to node i with probability p_i, where the probabilities are determined statically according to factors such as the average task-initiation rate and execution rate for each node. Each of these algorithms can potentially make poor assignment decisions. Because they do not consider node states when making such decisions, they can transfer a task initiated at an otherwise idle node to a node having a serious backlog of tasks.

Dynamic algorithms have the potential to outperform static algorithms by using system-state information to improve the quality of their decisions. For example, a simple dynamic algorithm might be identical to a static algorithm, except that it would not transfer an arriving task if the node where it arrived was idle. A more sophisticated dynamic algorithm might also take the state of the receiving node into account, possibly transferring a task to a node only if the receiving node was idle. A still more sophisticated dynamic algorithm might transfer an executing task if it was sharing a node with another task and some other node became idle. Essentially, dynamic algorithms improve performance by exploiting short-term fluctuations in the system state. Because they must collect, store, and analyze state information, dynamic algorithms incur more overhead than their static counterparts, but this overhead is often well spent. Most recent load-distributing research has concentrated on dynamic algorithms, and they will be our focus for the remainder of this article.

Adaptive load-distributing algorithms are a special class of dynamic algorithms. They adapt their activities by dynamically changing their parameters, or even their policies, to suit the changing system state. For example, if some load-distributing policy performs better than others under certain conditions, while another policy performs better under other conditions, a simple adaptive algorithm might choose between these policies based on observations of the system state. Even when the system is uniformly so heavily loaded that no performance advantage can be gained by transferring tasks, a nonadaptive dynamic algorithm might continue operating (and incurring overhead). To avoid overloading such a system, an adaptive algorithm might instead curtail its load-distributing activity when it observes this condition.

Load. A key issue in the design of dynamic load-distributing algorithms is identifying a suitable *load index*. A load index predicts the performance of a task if it is executed at some particular node. To be effective, load index readings taken when tasks initiate should correlate well with task-response times. Load indexes that have been studied and used include the length of the CPU queue, the average CPU queue length over some period, the amount of available memory, the context-switch rate, the system call rate, and CPU utilization. Researchers have consistently found significant differences in the effectiveness of such load indexes — and that simple load indexes are particularly effective. For example, Kunz[2] found that the choice of a load index has considerable effect on performance, and that the most effective of the indexes we have mentioned is the CPU queue length. Furthermore, Kunz found no performance improvement over this simple measure when combinations of these load indexes were used. It is crucial that the mechanism used to measure load be efficient and impose minimal overhead.

Preemptive versus nonpreemptive transfers. *Preemptive* task transfers involve transferring a partially executed task. This operation is generally expensive, since collecting a task's state (which can be quite large or complex) is often difficult. Typically, a task state consists of a virtual memory image, a process control block, unread I/O buffers and messages, file pointers, timers that have been set, and so on. *Nonpreemptive* task transfers, on the other hand, involve only tasks that have not begun execution and hence do not require transferring the task's state. In both types of transfers, information about the environment in which the task will execute must be transferred to the remote node. This information may include the user's current working directory and the privileges inherited by the task. Nonpreemptive task transfers are also called *task placements*. Artsy and Finkel[3] and Douglis and Ousterhout[4] contain detailed discussions of issues in preemptive task transfer.

Centralization. Dynamic load-distributing algorithms differ in their degree of centralization. Algorithms can be centralized, hierarchical, fully decentralized, or some combination of these. Algorithms with some centralized components are potentially less reliable than fully decentralized algorithms, since the failure of a central component may cause the entire system to fail. A solution to this problem is to maintain redundant components, which can become active when the previously active component fails. A second weakness of centralized algorithms is not so easily remedied: A central component is potentially a bottleneck, limiting load distribution. While hierarchical algorithms can alleviate both problems, the complete solution lies in fully decentralized algorithms.

Components of a load-distributing algorithm. Typically, a dynamic load-distributing algorithm has four components: a *transfer policy*, a *selection policy*, a *location policy*, and an *information policy*.

Transfer policy. A transfer policy determines whether a node is in a suitable state to participate in a task transfer, either as a sender or a receiver. Many proposed transfer policies are *threshold* policies.[1,5-7] Thresholds are expressed in units of load. When a new task originates at a node, the transfer policy decides that the node is a *sender* if the load at that node exceeds a threshold T_1. On the other hand, if the load at a node falls below T_2, the transfer policy decides that the node can be a *receiver* for a remote task. Depending on the algorithm, T_1 and T_2 may or may not have the same value.

Alternatives to threshold transfer policies include *relative* transfer policies. Relative policies consider the load of a node in relation to loads at other system nodes. For example, a relative policy might consider a node to be a suitable receiver if its load is lower than that of some other node by at least some fixed δ. Alternatively, a node might be considered a receiver if its load is among the lowest in the system.

Selection policy. Once the transfer policy decides that a node is a sender, a selection policy selects a task for transfer. Should the selection policy fail to find a suitable task to transfer, the node is no longer considered a sender.

The simplest approach is to select one of the newly originated tasks that caused the node to become a sender. Such a task is relatively cheap to transfer, since the transfer is nonpreemptive.

A selection policy considers several factors in selecting a task:

(1) The overhead incurred by the transfer should be minimal. For example, a small task carries less overhead.

Load sharing versus load balancing

Dynamic load-distributing algorithms can be further classified as being load-sharing or load-balancing algorithms. The goal of a *load-sharing algorithm* is to maximize the rate at which a distributed system performs work when work is available. To do so, load-sharing algorithms strive to avoid *unshared states*[1]: states in which some computer lies idle while tasks contend for service at some other computer.

If task transfers were instantaneous, unshared states could be avoided by transferring tasks only to idle computers. Because of the time required to collect and package a task's state and because of communication delays, transfers are not instantaneous. The lengthy unshared states that would otherwise result from these delays can be partially avoided through *anticipatory transfers*,[1] which are transfers from overloaded to lightly loaded computers, under the assumption that lightly loaded computers are likely to become idle soon. The potential performance gain of these anticipatory transfers must be weighed against the additional overhead they incur.

Load-balancing algorithms also strive to avoid unshared states, but go a step beyond load sharing by attempting to equalize the loads at all computers. Krueger and Livny[2] have shown that load balancing can potentially reduce the mean and standard deviation of task response times, relative to load-sharing algorithms. Because load balancing requires a higher transfer rate than load sharing, however, the higher overhead incurred may outweigh this potential performance improvement.

References

1. M. Livny and M. Melman, "Load Balancing in Homogeneous Broadcast Distributed Systems," *Proc. ACM Computer Network Performance Symp.*, 1982, pp. 47-55. Proceedings printed as a special issue of *ACM Performance Evaluation Rev.*, Vol. 11, No. 1, 1982, pp. 47-55.

2. P. Krueger and M. Livny, "The Diverse Objectives of Distributed Scheduling Policies," *Proc. Seventh Int'l Conf. Distributed Computing Systems*, IEEE CS Press, Los Alamitos, Calif., Order No. 801 (microfiche only), 1987, pp. 242-249.

(2) The selected task should be long lived so that it is worthwhile to incur the transfer overhead.

(3) The number of *location-dependent* system calls made by the selected task should be minimal. Location-dependent calls are system calls that must be executed on the node where the task originated, because they use resources such as windows, the clock, or the mouse that are only at that node.[4,8]

Location policy. The location policy's responsibility is to find a suitable "transfer partner" (sender or receiver) for a node, once the transfer policy has decided that the node is a sender or receiver.

A widely used decentralized policy finds a suitable node through *polling:* A node polls another node to find out whether it is suitable for load sharing. Nodes can be polled either serially or in parallel (for example, multicast). A node can be selected for polling on a random basis,[5,6] on the basis of the information collected during the previous polls,[1,7] or on a nearest neighbor basis. An alternative to polling is to broadcast a query seeking any node available for load sharing.

In a centralized policy, a node contacts one specified node called a *coordinator* to locate a suitable node for load sharing. The coordinator collects information about the system (which is the responsibility of the information policy), and the transfer policy uses this information at the coordinator to select receivers.

Information policy. The information policy decides when information about the states of other nodes in the system is to be collected, from where it is to be collected, and what information is collected. There are three types of information policies:

(1) *Demand-driven policies.* Under these decentralized policies, a node collects the state of other nodes only when it becomes either a sender or a receiver, making it a suitable candidate to initiate load sharing. A demand-driven information policy is inherently a dynamic policy, as its actions depend on the system state. Demand-driven policies may be sender, receiver, or symmetrically initiated. In *sender-initiated* policies, senders look for receivers to which they can transfer their load. In *receiver-initiated* policies, receivers solicit loads from senders. A *symmetrically initiated* policy is a combination of both: Load-sharing actions are triggered by the demand for extra processing power or extra work.

Using detailed state information does not always significantly aid system performance.

(2) *Periodic policies.* These policies, which may be either centralized or decentralized, collect information periodically. Depending on the information collected, the transfer policy may decide to transfer tasks. Periodic information policies generally do not adapt their rate of activity to the system state. For example, the benefits resulting from load distributing are minimal at high system loads because most nodes in the system are busy. Nevertheless, overheads due to periodic information collection continue to increase the system load and thus worsen the situation.

(3) *State-change-driven policies.* Under state-change-driven policies, nodes disseminate information about their states whenever their states change by a certain degree. A state-change-driven policy differs from a demand-driven policy in that it disseminates information about the state of a node, rather than collecting information about other nodes. Under centralized state-change-driven policies, nodes send state information to a centralized collection point. Under decentralized state-change-driven policies, nodes send information to peers.

Stability: We first informally describe two views of stability: the queuing theoretic perspective and the algorithmic perspective. According to the *queuing theoretic perspective*, when the long-term arrival rate of work to a system is greater than the rate at which the system can perform work, the CPU queues grow without bound. Such a system is termed unstable. For example, consider a load-distributing algorithm performing excessive message exchanges to collect state information. The sum of the load due to the external work arriving and the load due to the overhead imposed by the algorithm can become higher than the service capacity of the system, causing system instability.

On the other hand, an algorithm can be stable but still cause a system to perform worse than the same system without the algorithm. Hence, we need a more restrictive criterion for evaluating algorithms — the *effectiveness* of an algorithm. A load-distributing algorithm is effective under a given set of conditions if it improves performance relative to a system not using load distributing. An effective algorithm cannot be unstable, but a stable algorithm can be ineffective.

According to the *algorithmic perspective*, if an algorithm can perform fruitless actions indefinitely with nonzero probability, the algorithm is unstable. For example, consider *processor thrashing:* The transfer of a task to a receiver may increase the receiver's queue length to the point of overloading it, necessitating the transfer of that task to yet another node. This process may repeat indefinitely. In this case, a task moves from one node to another in search of a lightly loaded node without ever receiving any service. Casavant and Kuhl[9] discuss algorithmic instability in detail.

Example algorithms

During the past decade, many load-distributing algorithms have been proposed. In the following sections, we describe several representative algorithms that have appeared in the literature. They illustrate how the components of load-distributing algorithms fit together and show how the choice of components affects system stability. We discuss the performance of these algorithms in the "Performance comparison" section.

Sender-initiated algorithms

Under sender-initiated algorithms, load-distributing activity is initiated by an overloaded node (sender) trying to send a task to an underloaded node (receiver). Eager, Lazowska, and Zahorjan[6] studied three simple, yet effective, fully distributed sender-initiated algorithms.

Transfer policy. Each of the algorithms uses the same transfer policy, a threshold policy based on the CPU queue length. A node is identified as a sender if a new task originating at the node makes the queue length exceed a threshold T. A node identifies itself as a suitable receiver for a task transfer if accepting the task will not cause the node's queue length to exceed T.

Selection policy. All three algorithms have the same selection policy, considering only newly arrived tasks for transfer.

Location policy. The algorithms differ only in their location policies, which we review in the following subsections.

Random. One algorithm has a simple dynamic location policy called random, which uses no remote state information. A task is simply transferred to a node selected at random, with no information exchange between the nodes to aid in making the decision. Useless task transfers can occur when a task is transferred to a node that is already heavily loaded (its queue length exceeds T).

An issue is how a node should treat a transferred task. If a transferred task is treated as a new arrival, then it can again be transferred to another node, providing the local queue length exceeds T. If such is the case, then irrespective of the average load of the system, the system will eventually enter a state in which the nodes are spending all their time transferring tasks, with no time spent executing them. A simple solution is to limit the number of times a task can be transferred. Despite its simplicity, this random location policy provides substantial performance improvements over systems not using load distributing.[6] The "Performance comparison" section contains examples of this ability to improve performance.

Threshold. A location policy can avoid useless task transfers by polling a node (selected at random) to determine whether transferring a task would make its queue length exceed T (see Figure 2). If not, the task is transferred to the selected node, which must execute the task regardless of its state when the task actually arrives. Otherwise, another node is selected at random and is polled. To keep the overhead low, the number of polls is limited by a parameter called the *poll limit.* If no suitable receiver node is found within the poll limit polls, then the node at which the task originated must execute the task. By avoiding useless task transfers, the threshold policy provides a substantial performance improvement over the random location policy.[6] Again, we will examine this improvement in the "Performance comparison" section.

Shortest. The two previous approaches make no effort to choose the best destination node for a task. Under the shortest location policy, a number of nodes (poll limit) are selected at random and polled to determine their queue length. The node with the shortest queue is selected as the destination for task transfer, unless its queue length is greater than or equal to T. The destination node will execute the task regardless of its queue length when the transferred task arrives. The performance improvement obtained by using the shortest location policy over the threshold policy was found to be marginal, indicating that using more detailed state information does not necessarily improve system performance significantly.[6]

Information policy. When either the shortest or the threshold location policy is used, polling starts when the transfer policy identifies a node as the sender of a task. Hence, the information policy is demand driven.

Stability. Sender-initiated algorithms using any of the three location policies cause system instability at high system loads. At such loads, no node is likely to be lightly loaded, so a sender is unlikely to find a suitable destination node. However, the polling activity in sender-initiated algorithms increases as the task arrival rate increases, eventually reaching a point where the cost of load sharing is greater than its benefit. At a more extreme point, the workload that cannot be offloaded from a node, together with the overhead incurred by polling, exceeds the node's CPU capacity and instability results. Thus, the actions of sender-initiated algorithms are not effective at high system loads and cause system instability, because the algorithms fail to adapt to the system state.

Receiver-initiated algorithms

In receiver-initiated algorithms, load-distributing activity is initiated from an underloaded node (receiver), which tries to get a task from an overloaded node (sender). In this section, we describe an algorithm studied by Livny and Melman,[1] and Eager, Lazowska, and Zahorjan[5] (see Figure 3).

Transfer policy. The algorithm's threshold transfer policy bases its decision on the CPU queue length. The policy is triggered when a task departs. If the local queue length falls below the threshold T, then the node is identified as a receiver for obtaining a task from a

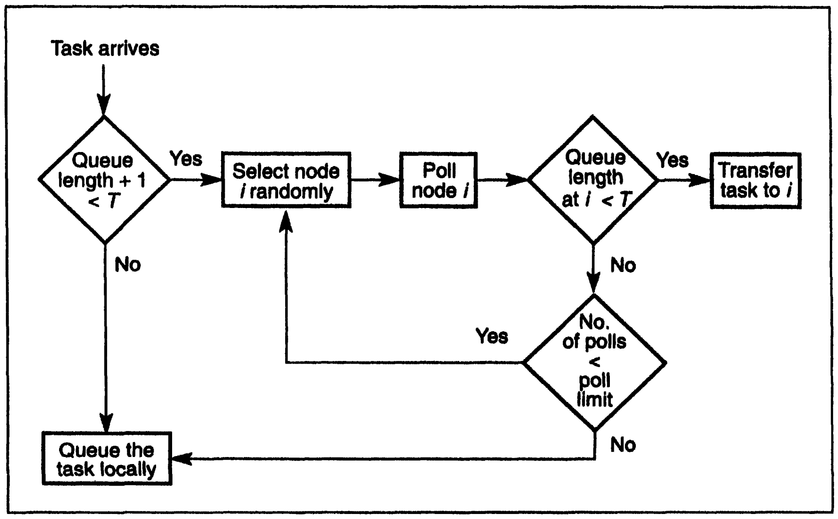


Figure 2. Sender-initiated load sharing using a threshold location policy.

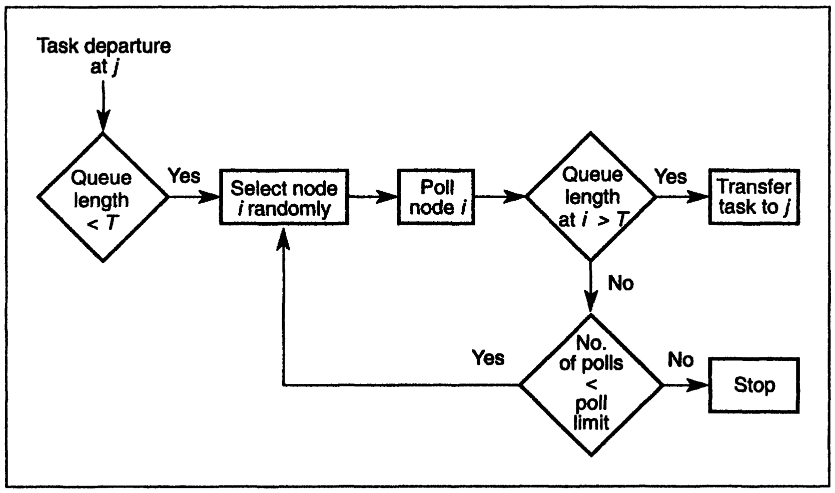


Figure 3. Receiver-initiated load sharing.

node (sender) to be determined by the location policy. A node is identified to be a sender if its queue length exceeds the threshold *T*.

Selection policy. The algorithm considers all tasks for load distributing, and can use any of the approaches discussed under "Selection policy" in the "Issues in load distributing" section.

Location policy. The location policy selects a node at random and polls it to determine whether transferring a task would place its queue length below the threshold level. If not, then the polled node transfers a task. Otherwise, another node is selected at random, and the procedure is repeated until either a node that can transfer a task (a sender) is found or a static poll limit number of tries has failed to find a sender.

A problem with the location policy is that if all polls fail to find a sender, then the processing power available at a receiver is completely lost by the system until another task originates locally at the receiver (which may not happen for a long time). The problem severely affects performance in systems where only a few nodes generate most of the system workload and random polling by receivers can easily miss them. The remedy is simple: If all the polls fail to find a sender, then the node waits until another task departs or for a predetermined period before reinitiating the load-distributing activity, provided the node is still a receiver.[7]

Information policy. The information policy is demand driven, since polling starts only after a node becomes a receiver.

Stability. Receiver-initiated algorithms do not cause system instability because, at high system loads, a receiver is likely to find a suitable sender within a few polls. Consequently, polls are increasingly effective with increasing system load, and little waste of CPU capacity results.

A drawback. Under the most widely used CPU scheduling disciplines (such as round-robin and its variants), a newly arrived task is quickly provided a quantum of service. In receiver-initiated algorithms, the polling starts when a node becomes a receiver. However, these polls seldom arrive at senders just after new tasks have arrived at the senders but before these tasks have begun executing. Consequently, most transfers are preemptive and therefore expensive. Sender-initiated algorithms, on the other hand, make greater use of nonpreemptive transfers, since they can initiate load-distributing activity as soon as a new task arrives.

An alternative to this receiver-initiated algorithm is the reservation algorithm proposed by Eager, Lazowska, and Zahorjan.[5] Rather than negotiate an immediate transfer, a receiver requests that the next task to arrive be nonpreemptively transferred. Upon arrival, the "reserved" task is transferred to the receiver if the receiver is still a receiver at that time. While this algorithm does not require preemptive task transfers, it was found to perform significantly worse than the sender-initiated algorithms.

Symmetrically initiated algorithms

Under symmetrically initiated algorithms,[10] both senders and receivers initiate load-distributing activities for task transfers. These algorithms have the advantages of both sender- and receiver-initiated algorithms. At low system loads, the sender-initiated component is more successful at finding underloaded nodes. At high system loads, the receiver-initiated component is more successful at finding overloaded nodes. However, these algorithms may also have the disadvantages of both sender- and receiver-initiated algorithms. As with sender-initiated algorithms, polling at high system loads may result in system instability. As with receiver-initiated algorithms, a preemptive task transfer facility is necessary.

A simple symmetrically initiated algorithm can be constructed by combining the transfer and location policies described for sender-initiated and receiver-initiated algorithms.

Adaptive algorithms

A stable symmetrically initiated adaptive algorithm. The *main cause* of system instability due to load sharing in the previously reviewed algorithms is indiscriminate polling by the sender's negotiation component. The stable symmetrically initiated algorithm[7] uses the information gathered during polling (instead of discarding it, as the previous algorithms do) to classify the nodes in the system as *sender/overloaded, receiver/underloaded,* or *OK* (nodes having manageable load). The knowledge about the state of nodes is maintained at each node by a data structure composed of a senders list, a receivers list, and an OK list. These lists are maintained using an efficient scheme: List-manipulative actions, such as moving a node from one list to another or determining to which list a node belongs, impose a small and constant overhead, irrespective of the number of nodes in the system. Consequently, this algorithm scales well to large distributed systems.

Initially, each node assumes that every other node is a receiver. This state is represented at each node by a receivers list containing all nodes (except the node itself), and an empty senders list and OK list.

Transfer policy. The threshold transfer policy makes decisions based on the CPU queue length. The transfer policy is triggered when a new task originates or when a task departs. The policy uses two threshold values — a lower threshold and an upper threshold — to classify the nodes. A node is a sender if its queue length is greater than its upper threshold, a receiver if its queue length is less than its lower threshold, and OK otherwise.

Location policy. The location policy has two components: the *sender-initiated component* and the *receiver-initiated component.* The sender-initiated component is triggered at a node when it becomes a sender. The sender polls the node at the head of the receivers list to determine whether it is still a receiver. The polled node removes the sender node ID from the list it is presently in, puts it at the head of its senders list, and informs the sender whether it is currently a receiver, sender, or OK. On receipt of this reply, the sender transfers the new task if the polled node has indicated that it is a receiver. Otherwise, the polled node's ID is removed from the receivers list and is put at the head of the OK list or the senders list based on its reply.

Polling stops if a suitable receiver is found for the newly arrived task, if the number of polls reaches a poll limit (a parameter of the algorithm), or if the receivers list at the sender node becomes empty. If polling fails to find a receiver, the task is processed locally, though it may later be preemptively transferred as a result of receiver-initiated load sharing.

The goal of the receiver-initiated component is to obtain tasks from a sender node. The nodes polled are selected in the following order:

(1) *Head to tail in the senders list.* The most up-to-date information is used first.
(2) *Tail to head in the OK list.* The most out-of-date information is used first in the hope that the node has become a sender.
(3) *Tail to head in the receivers list.* Again, the most out-of-date information is used first.

The receiver-initiated component is triggered at a node when the node becomes a receiver. The receiver polls the selected node to determine whether it is a sender. On receipt of the message, the polled node, if it is a sender, transfers a task to the polling node and informs it of its state after the task transfer. If the polled node is not a sender, it removes the receiver node ID from the list it is presently in, puts it at the head of the receivers list, and informs the receiver whether the polled node is a receiver or OK. On receipt of this reply, the receiver node removes the polled node ID from whatever list it is presently in and puts it at the head of its receivers list or OK list, based on its reply.

Polling stops if a sender is found, if the receiver is no longer a receiver, or if the number of polls reaches a static poll limit.

Selection policy. The sender-initiated component considers only newly arrived tasks for transfer. The receiver-initiated component can use any of the approaches discussed under "Selection policy" in the "Issues in load distributing" section.

Information policy. The information policy is demand driven, as polling starts when a node becomes either a sender or a receiver.

Discussion. At high system loads, the probability of a node's being underloaded is negligible, resulting in unsuccessful polls by the sender-initiated component. Unsuccessful polls result in the removal of polled node IDs from receivers lists. Unless receiver-initiated polls to these nodes fail to find senders, which is unlikely at high system loads, the receivers lists remain empty. This scheme prevents future sender-initiated polls at high system loads (which are most likely to fail). Hence, the sender-initiated component is deactivated at high system loads, leaving only receiver-initiated load sharing (which is effective at such loads).

At low system loads, receiver-initiated polls are frequent and generally fail. These failures do not adversely affect performance, since extra processing capacity is available at low system loads. In addition, these polls have the positive effect of updating the receivers lists. With the receivers lists accurately reflecting the system's state, future sender-initiated load sharing will generally succeed within a few polls. Thus, by using sender-initiated load sharing at low system loads, receiver-initiated load sharing at high loads, and symmetrically initiated load sharing at moderate loads, the stable symmetrically initiated algorithm achieves improved performance over a wide range of system loads and preserves system stability.

A stable sender-initiated adaptive algorithm. This algorithm[7] uses the sender-initiated load-sharing component of the previous approach but has a modified receiver-initiated component to attract future nonpreemptive task transfers from sender nodes. An important feature is that the algorithm performs load sharing only with nonpreemptive transfers, which are cheaper than preemptive transfers. The stable sender-initiated algorithm is very similar to the stable symmetrically initiated algorithm. In the following, we point out only the differences.

In the stable sender-initiated algorithm, the data structure (at each node) of the stable symmetrically initiated algorithm is augmented by an array called the *state vector.* Each node uses the state vector to keep track of which list (senders, receivers, or OK) it belongs to at all the other nodes in the system. For example, *statevector* [*nodeid*] says to which list node *i* belongs at the node indicated by *nodeid.* As in the stable symmetrically initiated algorithm, the overhead for maintaining this data structure is small and constant, irrespective of the number of nodes in the system.

The sender-initiated load sharing is augmented with the following step: When a sender polls a selected node, the sender's state vector is updated to show that the sender now belongs to the senders list at the selected node. Likewise, the polled node updates its state vector based on the reply it sent to the sender node to reflect which list it will belong to at the sender.

The receiver-initiated component is replaced by the following protocol: When a node becomes a receiver, it informs only those nodes that are misinformed about its current state. The misinformed nodes are those nodes whose receivers lists do not contain the receiver's ID. This information is available in the state vector at the receiver. The state vector at the receiver is then updated to reflect that it now belongs to the receivers list at all those nodes that were misinformed about its current state.

There are no preemptive transfers of partly executed tasks here. The sender-initiated load-sharing component will do any task transfers, if possible, on the arrival of a new task. The reasons for this algorithm's stability are the same as for the stable symmetrically initiated algorithm.

Performance comparison

In this section, we discuss the general performance trends of some of the example algorithms described in the pre-

Example systems

Here we review several working load-distributing algorithms.

V-system. The V-system[1] uses a state-change-driven information policy. Each node broadcasts (or publishes) its state whenever its state changes significantly. State information consists of expected CPU and memory utilization and particulars about the machine itself, such as its processor type and whether it has a floating-point coprocessor. The broadcast state information is cached by all the nodes. If the distributed system is large, each machine can cache information about only the best *N* nodes (for example, only those nodes having unused or underused CPU and memory).

The V-system's selection policy selects only newly arrived tasks for transfer. Its relative transfer policy defines a node as a receiver if it is one of the *M* most lightly loaded nodes in the system, and as a sender if it is not. The decentralized location policy locates receivers as follows: When a task arrives at a machine, it consults the local cache and constructs the set containing the *M* most lightly loaded machines that can satisfy the task's requirements. If the local machine is one of the *M* machines, then the task is scheduled locally. Otherwise, a machine is chosen randomly from the set and is polled to verify the correctness of the cached data. This random selection reduces the chance that multiple machines will select the same remote machine for task execution. If the cached data matches the machine's state (within a degree of accuracy), the polled machine is selected for executing the task. Otherwise, the entry for the polled machine is updated and the selection procedure is repeated. In practice, the cache entries are quite accurate, and more than three polls are rarely required.[1]

The V-system's load index is the CPU utilization at a node. To measure CPU utilization, a background process that periodically increments a counter is run at the lowest priority possible. The counter is then polled to see what proportion of the CPU has been idle.

Sprite. The Sprite system[2] is targeted toward a workstation environment. Sprite uses a centralized state-change-driven information policy. Each workstation, on becoming a receiver, notifies a central coordinator process. The location policy is also centralized: To locate a receiver, a workstation contacts the central coordinator process.

Sprite's selection policy is primarily manual. Tasks must be chosen by users for remote execution, and the workstation on which these tasks reside is identified as a sender. Since the Sprite system is targeted for an environment in which workstations are individually owned, it must guarantee the availability of the workstation's resources to the workstation owner. To do so, it evicts foreign tasks from a workstation whenever the owner wishes to use the workstation. During eviction, the selection policy is automatic, and Sprite selects only foreign tasks for eviction. The evicted tasks are returned to their home workstations.

In keeping with its selection policy, the transfer policy used in Sprite is not completely automated:

(1) A workstation is automatically identified as a sender only when foreign tasks executing at that workstation must be evicted. For normal transfers, a node is identified as a sender manually and implicitly when the transfer is requested.

(2) Workstations are identified as receivers only for transfers of tasks chosen by the users. A threshold-based policy decides that a workstation is a receiver when the workstation has had no keyboard or mouse input for at least 30 seconds and the number of active tasks is less than the number of processors at the workstation.

The Sprite system designers used semiautomated selection and transfer policies because they felt that the benefits of completely automated policies would not outweigh the implementation difficulties.

To promote fair allocation of computing resources, Sprite can evict a foreign process from a workstation to allow the workstation to be allocated to another foreign process under the following conditions: If the central coordinator cannot find an idle workstation for a remote execution request and it finds that a user has been allocated more than his fair share of workstations, then one of the heavy user's processes is evicted from a workstation. The freed workstation is then allocated to the process that had received less than its fair share. The evicted process may be automatically transferred elsewhere if idle workstations become available.

For a parallelized version of Unix "make," Sprite's designers have observed a speedup factor of five for a system containing 12 workstations.

Condor. Condor[3] is concerned with scheduling long-running CPU-intensive tasks (background tasks) only. Condor is designed for a workstation environment in which the total availability of a workstation's resources is guaranteed to the user logged in at the workstation console (the owner).

Condor's selection and transfer policies are similar to Sprite's in that most transfers are manually initiated by users. Unlike Sprite, however, Condor is centralized, with a workstation designated as the controller. To transfer a task, vious sections. In addition, we compare their performance with that of a system that performs no load distributing (a system composed of *n* independent M/M/1 systems) and that of an ideal system that performs perfect load distributing (no unshared states) without incurring any overhead in doing so (an M/M/K system). The results we present are from simulations of a distributed system containing 40 nodes, interconnected by a 10-megabit-per-second token ring communication network.

For the simulation we made the following assumptions: Task interarrival times and service demands are independently exponentially distributed, and the average task CPU service demand is one time unit. The size of a polling message is 16 bytes, and the CPU overhead to either send or receive a polling

a user links it with a special system-call library and places it in a local queue of background tasks. The controller's duty is to find idle workstations for these tasks. To accomplish this, Condor uses a periodic information policy. The controller polls each workstation at two-minute intervals to find idle workstations and workstations with background tasks waiting. A workstation is considered idle only when the owner has not been active for at least 12.5 minutes. The controller queues information about background tasks. If it finds an idle workstation, it transfers a background task to that workstation.

If a foreign background task is being served at a workstation, a local scheduler at that workstation checks for local activity from the owner every 30 seconds. If the owner has been active since the previous check, the local scheduler preempts the foreign task and saves its state. If the workstation owner remains active for five minutes or more, the foreign task is preemptively transferred back to the workstation at which it originated. The task may be transferred later to an idle workstation if one is located by the controller.

Condor's scheduling scheme provides fair access to computing resources for both heavy and light users. Fair allocation is managed by the "up-down" algorithm, under which the controller maintains an index for each workstation. Initially the indexes are set to zero. They are updated periodically in the following manner: Whenever a task submitted by a workstation is assigned to an idle workstation, the index of the submitting workstation is increased. If, on the other hand, the task is not assigned to an idle workstation, the index is decreased. The controller periodically checks to see if any new foreign task is waiting for an idle workstation. If a task is waiting, but no idle workstation is available and some foreign task from the lowest priority (highest index value) workstation is running, then that foreign task is preempted and the freed workstation is assigned to the new foreign task. The preempted foreign task is transferred back to the workstation at which it originated.

Stealth. The Stealth Distributed Scheduler[4] differs from V-system, Sprite, and Condor in the degree of cooperation that occurs between load distributing and local resource allocation at individual nodes. Like Condor and Sprite, Stealth is targeted for workstation environments in which the availability of a workstation's resources must be guaranteed to its owner. While Condor and Sprite rely on preemptive transfers to guarantee availability, Stealth accomplishes this task through preemptive allocation of local CPU, memory, and file-system resources.

A number of researchers and practitioners have noted that even when workstations are being used by their owners, they are often only lightly utilized, leaving large portions of their processing capacities available. The designers of Stealth[4] observed that over a network of workstations, this unused capacity represents a considerable portion of the total unused capacity in the system — often well over half.

To exploit this capacity, Stealth allows foreign tasks to execute at workstations even while those workstations are used by their owners. Owners are insulated from these foreign tasks through prioritized local resource allocation. Stealth includes a prioritized CPU scheduler, a unique prioritized virtual memory system, and a prioritized file-system cache. Through these means, owners' tasks get the resources they need, while foreign tasks get only the leftover resources (which are generally substantial). In effect, Stealth replaces an expensive global operation (preemptive transfer) with a cheap local operation (prioritized allocation). By doing so, Stealth can simultaneously increase the accessibility of unused computing capacity (by exploiting underused workstations, as well as idle workstations) and reduce the overhead of load distributing.

Under Stealth, task selection is fully automated. It takes into account the availability of CPU and memory resources, as well as past successes and failures with transferring similar tasks under similar resource-availability conditions. The remainder of Stealth's load-distributing policy is identical to the stable sender-initiated adaptive policy discussed in the "Example algorithms" section of the main text. Because it does not need preemptive transfers to assure the availability of workstation resources to their owners, Stealth can use relatively cheap nonpreemptive transfers almost exclusively. Preemptive transfers are necessary only to prevent starvation of foreign tasks.

References

1. M. Stumm, "The Design and Implementation of a Decentralized Scheduling Facility for a Workstation Cluster," *Proc. Second Conf. Computer Workstations*, IEEE CS Press, Los Alamitos, Calif., Order No. 810, 1988, pp. 12-22.

2. F. Douglis and J. Ousterhout, "Transparent Process Migration: Design Alternatives and the Sprite Implementation," *Software — Practice and Experience*, Vol. 21, No. 8, Aug. 1991, pp. 757-785.

3. M.J. Litzkow, M. Livny, and M.W. Mutka, "Condor — A Hunter of Idle Workstations," *Proc. Eighth Int'l Conf. Distributed Computing Systems*, IEEE CS Press, Los Alamitos, Calif., Order No. 865, 1988, pp. 104-111.

4. P. Krueger and R. Chawla, "The Stealth Distributed Scheduler," *Proc. 11th Int'l Conf. Distributed Computing Systems*, IEEE CS Press, Los Alamitos, Calif., Order No. 2144, 1991, pp. 336-343.

message is 0.003 time units. A nonpreemptive task transfer incurs a CPU overhead of 0.02 time units, and a preemptive transfer incurs a 0.1 time unit overhead. Transfer overhead is divided evenly between the sending and the receiving nodes. The amount of information that must be communicated for a nonpreemptive transfer is 8 Kbytes, while a preemptive transfer requires 200 Kbytes.

While the specific performance values we present are sensitive to these assumptions, the performance trends we observe are far less sensitive and endure across a wide range of distributed systems. Errors in the results we present are less than 5 percent at the 90 percent confidence level.

Figure 4 plots the performance of the following:

- M/M/1, a distributed system that performs no load distributing;
- a sender-initiated algorithm with a random location policy, assuming that a task can be transferred at most once;
- a sender-initiated algorithm with a threshold location policy;
- a symmetrically initiated algorithm (sender- and receiver-initiated algorithms combined);
- a stable sender-initiated algorithm;
- a receiver-initiated algorithm;
- a stable symmetrically initiated algorithm; and
- M/M/K, a distributed system that performs ideal load distributing without incurring overhead for load distributing.

A fixed threshold of T = upper threshold = lower threshold = 1 was used for each algorithm.

For these comparisons, we assumed a small fixed poll limit (5). A small limit is sufficient: If P is the probability that a particular node is below threshold, then the probability that a node below threshold is first encountered on the ith poll is $P(1 - P)^{i-1}$.[6] (This result assumes that nodes are independent, a valid assumption if the poll limit is small relative to the number of nodes in the system.) For large P, this expression decreases rapidly with increasing i. The probability of succeeding on the first few polls is high. For small P, the quantity decreases more slowly. However, since most nodes are above threshold, the improvement in systemwide response time that will result from locating a node below threshold is small. Quitting the search after the first few polls does not carry a substantial penalty.

Main result. The ability of load distributing to improve performance is intuitively obvious when work arrives at some nodes at a greater rate than at others, or when some nodes have faster processors than others. Performance advantages are not so obvious when all nodes are equally powerful and have equal workloads over the long term. Figure 4a plots the average task response time versus offered system load for such a homogeneous system under each load-distributing algorithm. Comparing M/M/1 with the sender-initiated algorithm (random location policy), we see that even this simple load-distributing scheme provides a substantial performance improvement over a system that does not use load distributing. Considerable further improvement in performance can be gained through simple sender-initiated (threshold location policy) and receiver-initiated load-

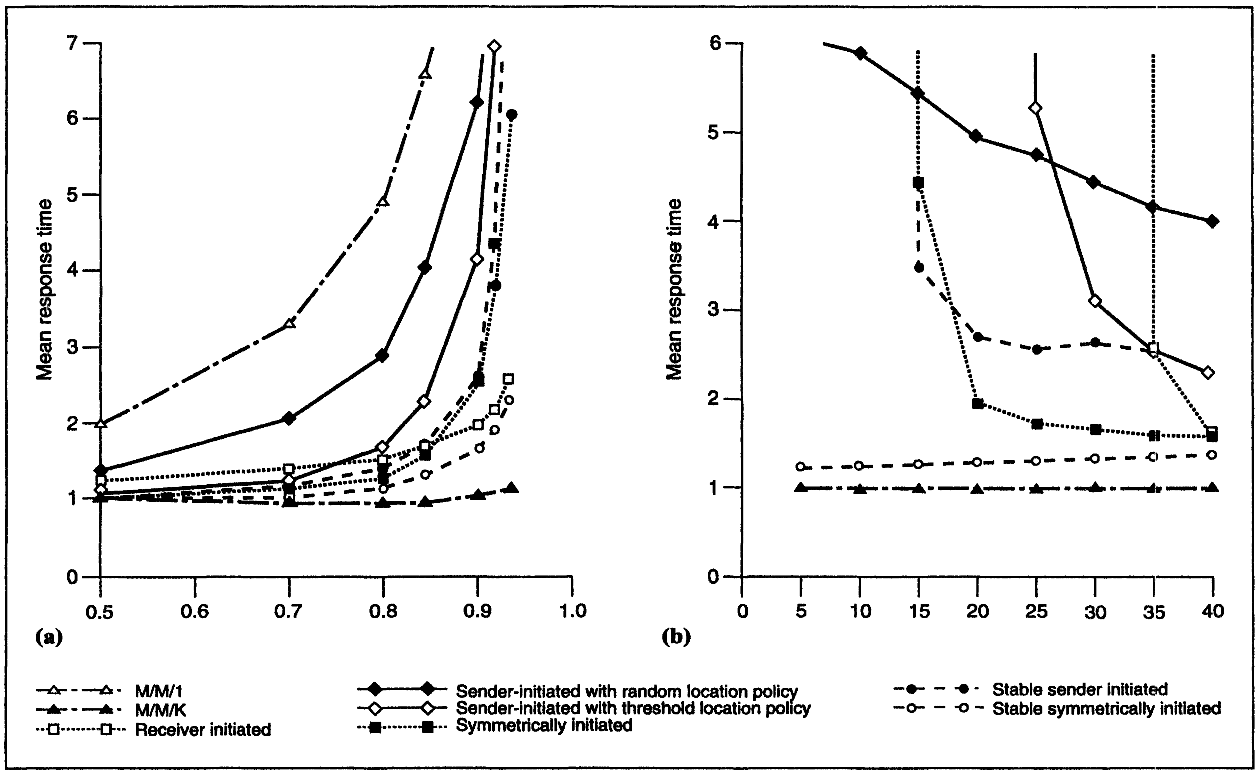


Figure 4. Average response time versus system load (a) and number of load-generating machines (b).

sharing schemes. The performance of the best algorithm — the stable symmetrically initiated algorithm — approaches that of M/M/K, though this optimistic lower bound can never be reached, since it assumes no load-distribution overhead.

Receiver- versus sender-initiated load sharing. Figure 4a shows that the sender-initiated algorithm with a threshold location policy performs marginally better than the receiver-initiated algorithm at light to moderate system loads, while the receiver-initiated algorithm performs substantially better at high system loads (even though the preemptive transfers it uses are much more expensive than the nonpreemptive transfers used by the sender-initiated algorithm). Receiver-initiated load sharing is less effective at low system loads because load sharing is not initiated at the time that one of the few nodes becomes a sender, and thus load sharing often occurs late.

In robustness, the receiver-initiated policy has an edge over the sender-initiated policy. The receiver-initiated policy performs acceptably over the entire system load spectrum, whereas the sender-initiated policy causes system instability at high loads. At such loads, the receiver-initiated policy maintains system stability because its polls generally find busy nodes, while polls due to the sender-initiated policy are generally ineffective and waste resources in efforts to find underloaded nodes.

Symmetrically initiated load sharing. This policy takes advantage of its sender-initiated load-sharing component at low system loads, its receiver-initiated component at high system loads, and both at moderate system loads. Hence, its performance is better than or matches that of the sender-initiated algorithm with threshold location policy at all levels of system load, and is better than that of the receiver-initiated policy at low to moderate system loads. Nevertheless, this policy also causes system instability at high system loads because of the ineffective polling by its sender-initiated component at such loads.

Stable load-sharing algorithms. The performance of the stable symmetrically initiated algorithm matches that of the best of the algorithms at low system loads and offers substantial improvements at high loads (greater than 0.85) over all the nonadaptive algorithms. This performance improvement results from its judicious use of the knowledge gained by polling. Furthermore, this algorithm does not cause system instability.

The stable sender-initiated algorithm yields as good or better performance than the sender-initiated algorithm with threshold location policy, with marked improvement at loads greater than 0.6, and yields better performance than the receiver-initiated policy for system loads less than 0.85. And it does not cause system instability. While it is not as good as the stable symmetrically initiated algorithm, it does not require expensive preemptive task transfers.

Heterogeneous workload. Heterogeneous workloads are common in distributed systems.[8] Figure 4b plots mean response time against the number of load-generating nodes in the system. All system workload is assumed to initiate at this subset of nodes, with none originating at the remaining nodes. A smaller subset of load-generating nodes indicates a higher degree of heterogeneity.

We assume a system load of 0.85. Without load distributing, the system becomes unstable even at low levels of heterogeneity under this load. While we do not plot these results, instability occurs for M/M/1 when the number of load-generating nodes is less than or equal to 33.

Among the load-distributing algorithms, Figure 4b shows that the receiver-initiated algorithm becomes unstable at a much lower degree of heterogeneity than any other algorithm. The instability occurs because random polling is unlikely to find a sender when only a few nodes are senders. The sender-initiated algorithm with a threshold location policy also becomes unstable at relatively low levels of heterogeneity. As fewer nodes receive all the system load, they must quickly transfer tasks. But the senders become overwhelmed as random polling results in many wasted polls.

The symmetrically initiated algorithm also becomes unstable, though at higher levels of heterogeneity, because of ineffective polling. It outperforms the receiver- and sender-initiated algorithms because it can transfer tasks at a higher rate than either. The stable sender-initiated algorithm remains stable for higher levels of heterogeneity than the sender-initiated algorithm with a threshold location policy because it is able to poll more effectively. Its eventual instability results from the absence of preemptive transfers, which prevents senders from transferring existing tasks even after they learn about receivers. Thus senders become overwhelmed.

The sender-initiated algorithm with a random location policy, the simplest algorithm of all, performs better than most algorithms at extreme levels of heterogeneity. By simply transferring tasks from the load-generating nodes to randomly selected nodes without any regard to their status, it essentially balances the load across all nodes in the system, thus avoiding instability.

Only the stable symmetrically initiated algorithm remains stable for all levels of heterogeneity. Interestingly, it performs better with increasing heterogeneity. As heterogeneity increases, senders rarely change their states and will generally be in the senders lists at the nonload-generating nodes. The nonload-generating nodes will alternate between the OK and receiver states and appear in the OK or receivers lists at the load-generating nodes. With the lists accurately representing the system state, nodes are often successful in finding partners.

Over the past decade, the mode of computing has shifted from mainframes to networks of computers, which are often engineering workstations. Such a network promises higher performance, better reliability, and improved extensibility over mainframe systems. The total computing capacity of such a network can be enormous. However, to realize the performance potential, a good load-distributing scheme is essential.

We have seen that even the simplest load-distributing algorithms have a great deal of potential to improve performance. Simply transferring tasks that arrive at busy nodes to randomly chosen nodes can improve performance considerably. Performance is improved still more if potential receiving nodes are first polled to determine whether they are suitable as receivers. Another significant performance benefit can be gained, with little additional complexity, by modifying such an algorithm to poll only the nodes most likely to be suitable receivers. Each of these algorithms can be designed to use nonpreemptive transfers exclusively. As a result, all carry relatively low overhead in software development time, maintenance time, and execution overhead. Among these algorithms, an algorithm such as the stable sender-initiated algorithm plotted in Figure 4 provides good performance over a wide range of conditions for all but the most "extreme" systems.

By extreme systems we mean systems that may experience periods during which a few nodes generate very heavy workloads. Under such conditions, a load-distributing algorithm that uses preemptive transfers may be necessary. We recommend an algorithm such as the stable symmetrically initiated algorithm, which can initiate transfers from either potential sending or potential receiving nodes, and targets its polls to nodes most likely to be suitable partners in a transfer. Such an algorithm provides larger performance improvements — over a considerably wider range of conditions — than any other algorithm discussed in this article. ■

Acknowledgments

We are deeply grateful to the referees, whose comments helped to improve the presentation. Portions of this material are based on work supported by the National Science Foundation under Grant No. CCR-8909072.

References

1. M. Livny and M. Melman, "Load Balancing in Homogeneous Broadcast Distributed Systems," *Proc. ACM Computer Network Performance Symp.*, 1982, pp. 47-55. Proceedings printed as a special issue of *ACM Performance Evaluation Rev.*, Vol. 11, No. 1, 1982, pp. 47-55.

2. T. Kunz, "The Influence of Different Workload Descriptions on a Heuristic Load Balancing Scheme," *IEEE Trans. Software Eng.*, Vol. 17, No. 7, July 1991, pp. 725-730.

3. Y. Artsy and R. Finkel, "Designing a Process Migration Facility: The Charlotte Experience," *Computer*, Vol. 22, No. 9, Sept. 1989, pp. 47-56.

4. F. Douglis and J. Ousterhout, "Transparent Process Migration: Design Alternatives and the Sprite Implementation," *Software — Practice and Experience*, Vol. 21, No. 8, Aug. 1991, pp. 757-785.

5. D.L. Eager, E.D. Lazowska, and J. Zahorjan, "A Comparison of Receiver-Initiated and Sender-Initiated Adaptive Load Sharing," *Performance Evaluation*, Vol. 6, No. 1, Mar. 1986, pp. 53-68.

6. D.L. Eager, E.D. Lazowska, and J. Zahorjan, "Adaptive Load Sharing in Homogeneous Distributed Systems," *IEEE Trans. Software Eng.*, Vol. 12, No. 5, May 1986, pp. 662-675.

7. N.G. Shivaratri and P. Krueger, "Two Adaptive Location Policies for Global Scheduling," *Proc. 10th Int'l Conf. Distributed Computing Systems*, IEEE CS Press, Los Alamitos, Calif., Order No. 2048, 1990, pp. 502-509.

8. P. Krueger and R. Chawla, "The Stealth Distributed Scheduler," *Proc. 11th Int'l Conf. Distributed Computing Systems*, IEEE CS Press, Los Alamitos, Calif., Order No. 2144, 1991, pp. 336-343.

9. T.L. Casavant and J.G. Kuhl, "Effects of Response and Stability on Scheduling in Distributed Computing Systems," *IEEE Trans. Software Eng.*, Vol. 14, No. 11, Nov. 1988, pp. 1,578-1,587.

10. P. Krueger and M. Livny, "The Diverse Objectives of Distributed Scheduling Policies," *Proc. Seventh Int'l Conf. Distributed Computing Systems*, IEEE CS Press, Los Alamitos, Calif., Order No. 801 (microfiche only), 1987, pp. 242-249.

Niranjan G. Shivaratri is a PhD student in the Department of Computer and Information Science at Ohio State University. From 1983 to 1987, he worked as a systems programmer for the Unisys Corporation, concentrating on network and operating systems software development. His research interests include distributed systems and performance modeling. He and Mukesh Singhal coauthored *Advanced Concepts in Operating Systems*, to be published by McGraw-Hill.

Shivaratri received a BS degree in electrical engineering from Mysore University, India, in 1979, and an MS degree in computer science from Villanova University in 1983. He is a member of the IEEE Computer Society.

Phillip Krueger is an assistant professor of computer science at Ohio State University. He directs the Stealth Distributed Scheduler Project at Ohio State, focusing on the design and construction of a distributed scheduler targeted for workstation-based distributed systems. His research interests include operating systems, concurrent and distributed systems, real-time systems, simulation, and performance analysis.

Krueger received his BS degree in physics, and his MS and PhD degrees in computer science from the University of Wisconsin - Madison. He is a member of the IEEE Computer Society and the ACM.

Mukesh Singhal has been a member of the faculty of the Department of Computer and Information Science at Ohio State University since 1986 and is currently an associate professor. His research interests include distributed systems, distributed databases, and performance modeling. He and Niranjan Shivaratri coauthored *Advanced Concepts in Operating Systems*, to be published by McGraw-Hill. He also served as co-guest editor of the August 1991 special issue of *Computer* on distributed computing systems.

Singhal received a bachelor of engineering degree in electronics and communication engineering with high distinction from the University of Roorkee, India, in 1980, and a PhD in computer science from the University of Maryland in 1986.

Readers can contact the authors at the Dept. of Computer and Information Science, Ohio State University, Columbus, OH 43210. Their e-mail addresses are {niran philk singhal}@cis.ohio-state.edu.