From: Conference on Future Trends in Distributed Computer Systems in the 90's; Sept. 14-16, 198; Hong Kong

A GLOBAL LOAD BALANCING STRATEGY FOR A DISTRIBUTED COMPUTER SYSTEM

Katherine M. Baumgartner and Benjamin W. Wah

Department of Electrical and Computer Engineering
and the Coordinated Sciences Laboratory
University of Illinois at Urbana-Champaign
1101 W. Springfield Avenue
Urbana, IL 61801

ABSTRACT

Distributed Computing Systems (DCSs) evolved to provide communication among replicated and physically distributed computers as hardware costs decreased. Interconnecting physically distributed computers allows better communication and improved performance through redistribution (or load balancing) of workload. In this paper, we describe a load balancing strategy for a computer system connected by multiaccess broadcast network. The strategy uses the existing broadcast capability of these networks to implement an efficient search technique for finding stations with the maximum and the minimum workload. The overhead of distributing status information in the proposed strategy is independent of the number of stations. This result is significant because the primary overhead in load balancing lies in the collection of status information. An implementation of the proposed strategy on a network of Sun workstations is presented. It consists of two modules that are executed at all participating computers: the distributed-search module that isolates the maximally and minimally loaded computers, and the job-migration module that places a job based on the load extremes.

INDEX TERMS: Broadcast, collision detection, distributed computer system, dynamic programming, load balancing, multiaccess networks.

1. INTRODUCTION

Early computer systems were centralized due to the cost of replicating hardware and additional staffing. As hardware costs dropped, it became possible for smaller organizations to own computer systems. Consequently, several computer installations could be present on a college or industrial campus, and local area networks (LANs) evolved to allow communication among the computer installations. The resulting collection of resources and the communications medium are Distributed Computer Systems (DCSs). This trend is even more prevalent now as networks of personal computers and workstations are common in the work place.

The networks which connect computers and workstations allow communication, but they also have the capability for allowing efficient sharing of resources. Since the demands for computing power are continually increasing, the network can be used for scheduling (or load balancing) operations during time when it is otherwise idle. DCSs can provide a cost-effective solution to increase the computing power available to a single user if jobs can be scheduled to exploit the potential parallelism. Livney and Melman [LiM82] have shown that in a system of n independent processors modeled as M/M/1 systems [Kle75], the condition in which a job is waiting for service at one processor while another processor is idle occurs 70% of the time for traffic intensities (the ratio of arrival rate to service rate) ranging from 0.5 to 0.8. This idle-while-waiting condition indicates the possibility of reducing the average job delay. With a global scheduling strategy for a DCS, the occurrence of this condition can be reduced and, consequently, the overall performance can be improved.

Research supported by National Aeronautics and Space Administration Contract NCC 2-481 and National Science Foundation Grant DMC 85-19649.

Conference on Future Trends in Distributed Computer Systems in the 90's; September 14-16, 1988; Hong Kong.

1.1 Scheduling Terminology

There is a considerable amount of conflict in the literature concerning the terminology used to describe the attributes of scheduling strategies. This section discusses terminology for classifying scheduling strategies. First, some previous classifications of scheduling strategies are reviewed. Using these results, the terminology used in this paper is described next.

Wang and Morris developed a classification of scheduling algorithms [WaM85]. Their criteria for classification is whether the strategy is source initiated or sink initiated, meaning whether overloaded resources seek to alleviate their load, or lightly loaded resources actively pursue more work. Additionally, the level of information dependency is a factor. Information dependency refers to the level at which a resource has information about the current state (or workload) of other resources.

The terminology used in global scheduling (or load balancing) [ChK79, NiH81, WaJ83, BaW85, EaL86, Ezz86] is varied and conflicting. Some features commonly discussed are whether the scheduling intelligence is centralized or distributed, and whether the scheduling decisions are static (independent of the current state of the system) or dynamic (dependent on the current state of the system). This characteristic is also referred to as being deterministic versus probabilistic, or adaptive versus non-adaptive (adaptability also refers to a different attribute as discussed below). These features are useful for comparing scheduling strategies and indicating the potential of a given scheduling algorithm.

The informal classification used for load balancing problems above was formalized by Casevant and Kuhl in an attempt to unify the diverse notation used [CaK88]. Their classification was designed for distributed computing systems. This classification consists of two parts: a hierarchical classification and a flat classification. The hierarchical classification is used to show where some characteristics are exclusive. The flat classification gives definitions of attributes that are not exclusive. Several observations about this taxonomy are as follows.

1. Adaptability refers to long-term algorithm state-dependency rather than short-term state-dependency for scheduling decisions. Adaptability is available with both static and dynamic scheduling decisions since the algorithm can be static or dynamic for one time interval, and then change for the next time interval.
2. Load balancing and optimality are considered characteristics of scheduling strategies, rather than requirements of the scheduler.
3. One-time reassignment and dynamic reassignment (corresponding to preemption in the one-processor case) are considered strategy characteristics rather than capabilities of the DCS.
4. Bidding in the flat portion of the classification and cooperation in the hierarchical portion are not distinct.

The terminology used here is not exclusively adopted from any of these sources for the following reasons. Wang and Morris' classification focuses on only two aspects of the strategy (initiation location and information dependency), so it not extensive enough. Casevant and Kuhl's taxonomy is not used exclusively because there is overlap between their classification of scheduling strategies and requirements of a scheduler.

Table 1 summarizes the terminology used in this paper. The first characteristic is the level of scheduling which indicates whether scheduling is among resources, or within a resource. Next is the rule basis, which

Table 1. Revised Classification of Scheduling Strategies

Characteristic	Values	Explanation
Level of Scheduling	Intra-resource vs. Inter-resource	Refers to scheduling within a node or resource as opposed to among nodes or resources. There may be multiple sub-levels of intra-resource scheduling. Also called global versus local.
Rule Basis	Static vs. Dynamic	Refers to the flexibility of scheduling rules to react to the current state of the system. A static schedule bases rules on unchanging system characteristics. A dynamic schedule bases rules on the current state of the system. Also called state-dependency.
Location of Control	Distributed vs. Hierarchical vs. Centralized	Describes where the responsibility for scheduling decisions lies. This applies primarily to dynamic rule basis since static rule basis implies a centralized decision. Hybrids are also possible.
Cooperation	Negotiated vs. Independent	Describes the interaction among locations of control. This applies to distributed or hierarchical control since centralized control does not have separate cooperating modules.
Initiation	Source vs. Sink vs. Both	Which processor initiates job movement: the overloaded processor (source initiated) or the underloaded processor (sink initiated).
Adaptability	Adaptable vs. Non-adaptable	Refers to flexibility of the algorithm, and whether the algorithm changes execution based on job characteristics of the system.

refers to what the scheduling decisions are based upon, the static system characteristics or the dynamic state of the system. Note that this is different from the static versus dynamic arrivals of jobs to a DCS. Next is the location of control. In case that job distribution can be controlled, the controlling processors can negotiate to make scheduling decisions, or function independently. Initiation and adaptability are the final two characteristics. The tradeoffs of these different characteristics are discussed in Section 1.2.

1.2. Previous Work

In this section, the results of previous studies are shown by grouping problems according to whether processing requirements of arriving jobs are specified stochastically (as a distribution) or deterministically (in exact amounts).

A great deal of research was done for scheduling tasks with exact a priori knowledge of execution requirements during the 1960s and 1970s. This work has been described in several books [CoM67, Bak74, Cof76] and survey papers [Gon77, GrL77]. This is a class of problems that frequently occur in a manufacturing environment and are scheduling problems at a inter-resource level. Scheduling strategies for this type of problems have been discussed extensively [CoM67, GrL77]. Solutions are optimal, near optimal, approximate, or heuristic. Schedules may be explicit time-event pairs or static rules, such as the shortest-job-first rule. Scheduling intelligence in this case is centralized.

As DCSs evolved, scheduling problems related to computing environment received more attention. Some early scheduling problems for the multiprocessor environment considered scheduling tasks with acyclic precedence requirements with no communication among tasks [RaC72, GoR72]. These studies assumed that a reasonable estimate of resource requirements could be obtained with a preprocessing phase of a program. Results of these studies included a method for determining the minimum computation time, and a method for determining the minimum number of processors to achieve the minimum computation time [RaC72]. Also, a comparison of a centralized versus a decentralized algorithm indicated that the decentralized algorithm performed better.

Scheduling tasks with intertask communication is a more difficult but more realistic problem. Improving performance requires limiting excessive communication and evenly distributing the workload among processors. If all tasks are scheduled on one processor, there is no communication cost, but there is no benefit from a multiplicity of processors. If the tasks are distributed such that processor utilization is completely uniform, full advantage of the multiplicity of processors is realized; however, communications costs may be so large that the benefits of concurrent execution are reduced. Since the goals of limiting communication and balancing load are directly conflicting, tradeoff must be made in obtaining the maximum performance.

Several studies were performed by Stone [Sto77, StB78, Sto78, RaS79] in which the relationship among jobs was represented using graphs. The nodes represent tasks with execution requirements, the edges are labeled with communication costs. Communication is required for one module to signal its successor on completion. An assignment of tasks is specified by a cut that divides the graph into as many sets as there are processors. The cost of the assignment is equal to the sum of the execution costs plus the sum of the communication costs among tasks that are not assigned to the same processor, that is, the sum of the weights of the edges on the cut. Consequently, the minimum cost assignment corresponds to the minimum cutset. The overhead of this method is sufficiently large so that it is limited to system with two to three processors. A similar mapping problem for larger systems was explored by Bokhari [Bok81] in which communicating modules were placed as much as possible on adjacent processors. Other approaches to this problem include graph matching [ChH80, RaS79], mathematical programming [Weg80, ChH80], branch and bound optimization [MaL82], and heuristics [ApI78, ChH80, KrH80, LoL81, Efe82]. An optimal solution for the n-processor case was studied by Chou and Abraham [ChA82].

The above strategies are for static sets of jobs, and the strategies themselves are static. Their execution time is too long for them to be effective for dynamically arriving tasks. Their utility is, therefore, useful during the planning phase of a system or for special-purpose systems in which timing is critical. Scheduling strategies for dynamically arriving jobs cannot use these techniques due to time constraints.

Issues that are important in developing strategies for dynamic scheduling problems with stochastic resource requirements include tradeoff between balanced load and communication overhead, location of control, status information used for the scheduling decisions, and the initiation point.

The tradeoff between balanced load and communication overhead has been discussed earlier. In deciding the location of control, the complexity of the scheduling algorithm and its benefits must be considered. A centralized location of control may allow the scheduling strategy to be simpler. However, the decision point has the potential of becoming a bottleneck and a critical failure point. If a distributed decision is made, the overhead of distributing status information can be so high that the benefit of load balancing is reduced. Trace driven simulation results by Zhou [Zho86] show that neither centralized or distributed strategies are always superior, and that communication overhead is important for both. The third consideration is what status information to use for scheduling decisions. Studies indicate that excessive status information is not only unnecessary, but can be detrimental [LiM82, WaM85, TaT85].

Several centralized, static scheduling strategies have been proposed. Proportional branching is a static, sink initiated strategy in which jobs are routed to processors with a probability determined by the relative power of the processors [ChK79]. Ni and Hwang found optimal, static, sink-initiated strategies for single and multiple job classes under centralized control

[NiH81]. An optimal static source-initiated strategy was found by Tantawi and Towsley for scheduling jobs that are modeled as independent tasks [TaT85]. Dynamic strategies have more potential than static strategies because they can react to changes in the system state. Chow and Kohler proposed three dynamic centralized strategies with a job dispatcher (sink initiated) and found the one that maximizes throughput gives the best performance.

Distributed dynamic strategies are more complex than centralized strategies due to the added task of coordinating independent actions. Typically, distributed and dynamic strategies require negotiation among participating processors, which involves communication of status information, and the selection of processors involved in the scheduling decision. The simplest method is to maintain a centralized table with load information, and processors can consult the table before sending jobs for remote execution [HwC82]. This method has the similar problems as the centralized dispatcher in the sense of reliability and boutleneck. Another technique is to have each processor broadcast its load and keep track of the loads of other processors. This, in effect, is equivalent to duplicating the workload table at each processor [LiM82, Zho86]. Alternatively, only significant load changes can be broadcast, resulting in a decrease in network traffic [LiM82, Zho86]. Other methods of exchange involve nearest neighbors [Wil83, KeL84]. Stankovic has proposed a strategy that includes a bidding phase in which negotiations take place [RaS84, Sta85].

Some significant results of these previous studies are as follows.

1. Load balancing is beneficial since load imbalance occurs frequently in a system with ten or more processors.
2. Excessive state information to make a load balancing decision is not necessary and may be detrimental.
3. Sink-initiated strategies have the potential for improved performance over source-initiated ones.
4. Dynamic strategies have greater potential that non-adaptive ones.
5. Centralized strategies may have reliability and bottleneck problems.
6. Communication of scheduling information should not interfere with regular message transfers.
7. There is a mismatch between the capabilities of the network and the communications required for scheduling operations.

1.3. Objectives and Organization

A global scheduling strategy is studied in this paper, with an objective of finding the maximum attainable performance of such a strategy. A system with a broadcast bus is chosen because a bus connection is common in many local DCSs and workstations. This paper describes the proposed strategy, the motivation for its development, its implementation, and its performance. Section 2 describes a scheduling strategy that transfers workload from the maximally loaded processor to the minimally loaded processor. The sequence of operations, the technique for isolating the processors, a characterization of load distribution, and performance based on simulations are detailed. In Section 3, the implementation of the proposed strategy is described. An overview of the software is presented, followed by a detailed description of each program module. Next, the performance of the strategy on a network of Sun workstations is shown. Conclusions and suggestions for future work are drawn in Section 4.

2. SCHEDULING STRATEGY

In this section an overview of the scheduling problem on a broadcast bus is presented. Figure 1 shows a diagram of the system under consideration. There are multiple identical processors connected by a broadcast bus. Each processor can have arrivals external to the system or from the bus. Jobs are modeled as independent tasks. If jobs are migrated to a processor across the bus, the results must be returned to the originating processor when execution is completed. Moreover, the queue at each processor is finite: only a limited number of jobs may be waiting for execution.

2.1. Development of the Strategy

The first step in defining a procedure is to find those system states in which job redistribution can result in improved performance. Obviously the occurrence of the idle-while-waiting condition must be decreased as must

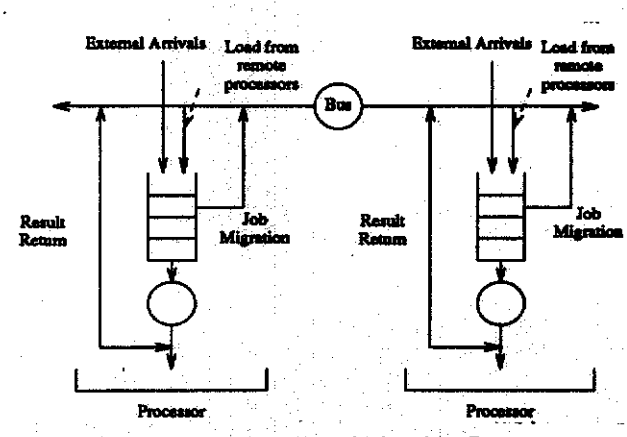

Figure 1. Model of a Broadcast Bus

the occurrence of any state that makes idle-while-waiting more likely. In a batch processing system, idle-while-waiting will not occur when there is at least one job at each processor at any time. Hence, the likelihood of idle-while-waiting can be minimized if if jobs are evenly distributed. In a multiprogramming systems, assuring that processors are busy is not sufficient to minimize idle-while waiting. Since the response time for each job is degraded as more jobs are added to the active queue, it is important to distribute all available jobs evenly, while attaining reasonable response time.

An important point here is that the number of jobs at a processor, while frequently a good reflection of load, is not always adequate. Other factors that may contribute to the workload at a processor include physical differences of processors (such as speed or size of main memory), paging activity, and the ratio of processing activities and input/output activities in jobs.

The strategy discussed here uses the queue length of active jobs at a processor as a metric to indicate workload. A queue length imbalance will make the idle-while-waiting condition more likely. Hence, a load redistribution action is needed when there is significant difference between queue lengths, such that the estimated total overhead of migrating a job, queueing delay at a remote processor and later returning results is less than the delay a job would experience at its source queue.

An ideal redistribution of jobs, given that the metric used is the queue length at each processor, is to have equal number of jobs at each processor. Since a single bus is used to connect all the computers, only one job can be migrated at any one time, and it is not possible to perform such an ideal redistribution in one step. The movement with the most impact is to take a job from the maximally loaded processor, and send it to the minimally loaded processor. Such a strategy is both source- and sink-initiated. Additionally minimal status information needed because it is only necessary to identify the maximally and the minimally loaded processors.

Three scheduling operations are required for this redistribution on a bus network: identification of the maximally and the minimally loaded processors, job migration, and result return. Migrating jobs and returning results are straightforward because existing communication facilities can be utilized; however, identifying processors with the load extremes efficiently is more difficult. Such an operation should have very low complexity, preferably independent of the number of processors connected to the bus. Any centralized scheduling algorithm, such as polling, is not suitable here.

2.2. Scheduling Sequence

There are three steps to developing the scheduling sequence: identifying the operations using the network, determining their relative priorities, and organizing the execution of these operations according to their priorities. There are four types of tasks that require the use of the network: regular message transfers, identifying the maximally and the minimally loaded processors (max/min identification), job migration, and result return. Regular message transfer is assigned the highest priority, since it is the original purpose of the network. The priorities of the remaining tasks are determined by considering the relative overheads in terms of the additional total system delay (the sum of the delays of all jobs) incurred.

First, the relative priority between identifying the i^{th} maximally/minimally-loaded-processor pair and the migration of the job between the $(i-1)^{th}$ pair is determined. Two cases are considered. The first is when there are idle processors. If job migration is done first, then the migrated job can begin execution immediately upon arrival at the destination processor, so its delay is increased by the time required to send it across the network. In contrast, if the max/min *identification* is done first, then the same job will incur the delay of that operation as well. Clearly, migrating the job first is better. The second case is when there are no idle processors. The ordering of the tasks is not critical in this case, since migrating the job first does not immediately contribute to reducing the job delay, as the job may not begin execution upon arrival at the destination. This result is true for any job migration and max/min identification, and performing the migration first will result in a total delay equal to or smaller than performing the max/min identification first. Consequently, job migration should have higher priority over max/min identification.

Next, the relative priority between result return and job migration is considered. When there are no idle processors, delay is added directly to the job waiting for result return. Since the job waiting for job migration will not be able to begin execution immediately upon arrival at the destination, result return should take precedence. When there are idle processors, both the delay for job migration and the delay for result return will reduce the overall delay. Performing the result return first always improves the overall performance as much or more than performing job migration first. It is also easy to see that result return should always have precedence over max/min identification by a similar argument.

In summary, the priority ordering for tasks using the bus network is (1) regular message transfer, (2) result return, (3) job migration, and (4) max/min identification.

The scheduling strategy consists of two steps that are executed repeatedly. The first is to determine which of the current tasks has the highest priority, and the second step is to execute that task. Due to the ordering of the priorities, only one job will ever be waiting for job migration, but potentially more than one may be waiting for result return.

One consideration associated with priorities is overhead. The discussion above assumes that each of the steps has similar overhead. If the overheads associated with tasks are considerably different, priority enforcement changes. A specific case is processors sharing secondary storage. Job migration and result return have lower overhead on such a system than on one without a shared disk. File transfer is not explicitly needed since all processors have access to a common secondary storage. Further, if the overhead of resolving priorities is large, then it is more efficient not to schedule according to priorities.

2.3. Max/Min Identification

Carrier-sense-multiaccess networks with *collision detection* (CSMA/CD) are a type of local-area network with packet switching and a bus topology [Tan81]. CSMA/CD networks evolved from CSMA networks that have *listen-before-talk* protocols to avoid overlapping transmissions. The collision-detection ability of CSMA/CD networks allows processors to additionally *listen-while-talk*, so collisions resulting from simultaneous transmissions can be detected and stopped immediately.

There are three types of contention resolution protocols for CSMA/CD networks. *Collision-free protocols* strictly schedule bus accesses, so no collisions occur. *Contention protocols* function at the other extreme allowing processors to transmit whenever they find the bus idle. When collisions occur because of simultaneous transmissions, processors stop transmitting, wait for some prescribed amount of time, and try again. The backoff algorithm of Ethernet [MeB76] is an example in this class. The disadvantage of collision-free protocols lies in the overhead of waiting for transmission, while the disadvantage of contention protocols is the time wasted during collisions. The third type of contention-resolution protocol is the *limited-contention protocol*. This type of protocol chooses a processor for transmission from among those waiting to transmit based on a priori information, such as the channel load. The Virtual-Window Protocol proposed by Wah and Juang [WaJ83, JuW84] is an example of a limited-contention protocol.

```
procedure virtual_window_protocol;
/* choose_window()        function returning the upper bound for the next window
 * contention_state()     procedure returning the state of the network
 * contend()              procedure to contend and broadcast argument
 * lb_window              lower bound for window to be chosen
 * ub_window              upper bound for window to be chosen
 * window_bd_up           actual upper bound of window chosen
 */
contending = true;
lb_window = L;
ub_window = U;
while contending do {
        /* operations performed in one contention slot */
        window_bd_up = choose_window(lb_window, ub_window);
        if (contention_parameter $<=$ window_bd_up)
                        and (contention_parameter > lb_window) then {
                contend(contention_parameter);
                contention_state(state);
                if (state = idle) then
                        lb_window = window_bd_up;
                else if (state = collision) then
                        ub_window = window_bd_up;
                else if (state = success) then
                        contending = false;
        } else
                contending = false;
}
```

Figure 2. The Virtual-Window Protocol

The *Virtual-Window Protocol* (VWP) uses a three-state collision-detection mechanism. After each attempted broadcast, there are three possible outcomes: *collision* (more than one broadcast), *idle* (no broadcast), and *success* (exactly one broadcast). Stations wishing to transmit packets participate in a *contention period* that consists of a number of *contention slots*. Each station generates a random number called a *contention parameter* that is used for the entire contention period. The parameter is in an interval with upper and lower bounds U and L, respectively. Successive choices of smaller intervals in each contention slot attempt to isolate the minimum contention parameter. The steps performed by each station in a contention period are shown in Figure 2.

For regular message transfers, each station has equal chance of being chosen for transmission, so the contention parameters are random numbers generated from a uniform distribution on the interval (0,1]. The stations maintain a common window (interval) for contention. In a contention slot, stations having contention parameters within the window broadcast a short signal to contend for the channel. If a collision or no transmission occurs, the window boundaries are adjusted in parallel at all stations for the next contention slot. Stations having contention parameters outside the window stop contending and wait for the next contention period. The above steps are repeated until a single station is isolated in the window. This station is the winner and is allowed to transmit its packet. The distribution of the contention parameters and an estimate of the channel load are used to update the window efficiently, so the number of contention slots is kept to a minimum.

The windows chosen in these examples are not the optimal windows but are chosen to illustrate the characteristics of the protocols. w_1, the upper bound for the first window chosen, is 0.51. All stations with contention parameters less that or equal to 0.51 are allowed to broadcast, in this case stations 1, 3, and 4. The result of this contention slot is a collision, the interval to be searched is updated to (0, 0.51], and stations 2 and 5 are eliminated from the contention. w_2, the upper bound for the next window, is 0.25. The result of the second contention slot is idle (no broadcast), so the interval is updated to (0.25, 0.51]. No stations were eliminated as a result of this contention slot. For the third contention slot, the upper bound of the window is chosen to be 0.32. The result is a successful transmission, and station 4 is isolated and "wins" the contention.

The window-selection process is formulated as a dynamic programming problem, and details have been shown elsewhere [WaJ85]. Analyses and simulations have shown that contention can be resolved in an average of 2.4 contention slots, independent of the number of contending stations and the distribution function of the contention parameters, if the parameters are independent and identically distributed [WaJ83].

The VWP locates the extremum of a set of independent contention parameters using information gathered during three-state collision detection

in a contention slot. Many existing networks do not make the three-state collision-detection information available to the applications software because a contention slot is a small amount of time (50 to 100 microseconds) relative to the time required to propagate information through all levels of software to the applications level (hundreds of microseconds). As a result, the VWP cannot be implemented easily at the applications level in software, and additional hardware modification to the network interface is required.

To implement an efficient max/min-search protocol at the applications level of an existing system without modifying the hardware, two important steps must be taken. First, the algorithm must be simplified to compensate for the additional overhead for communicating status information to the applications level. Second, since information concerning outcome of contention is not readily available at the applications level, the protocol must be redesigned to use only readily available information.

At the applications level, each station has an independent *search parameter*, which reflects the local workload. An iteration at this level is a *broadcast slot*, which consists of resolving contention at the network interface, followed by a broadcast of a message to all stations. A broadcast slot has two possible outcomes, *idle* (no stations attempt to broadcast), or *transmission* (one or more stations attempt to broadcast resulting in contention resolution, and one station broadcasts its search parameter). A broadcast slot may consist of a number of contention slots; however, information about each contention slot is not sent to the applications level.

There are a number of differences between searching at the applications level and the network level.

1. The contention parameters are the search parameters for the VWP, which is not the case for the window protocol at the applications level.
2. An iteration of the VWP is a contention slot with three possible outcomes: idle, collision, and success. An iteration of the window protocol at the applications level is a broadcast slot with two possible outcomes: transmission and idle.
3. An iteration of the VWP takes less time than an iteration of the window protocol at the applications level. Normally, a contention slot takes tens of microseconds, while broadcasting a short message takes hundreds of microseconds.

Since the information available for window selection is different at the applications level, the decision process has to be modified. Three possible window-search strategies for identifying the minimum are the one-broadcast strategy, the two-broadcast strategy, and the combined strategy. The identification of the maximum is similar and is not described. The algorithm, the technique for making window choices, and an implementation are shown for the one-broadcast strategy. The other two strategies are described briefly here, as their performance is inferior [BaW87]. In contrast to the VWP, dynamic programming methods to optimize window choices are not used because the Principle of Optimality is not satisfied. The performance of these strategies are compared using the number of broadcast slots they require to isolate the minimum search parameter.

The one-broadcast strategy allows a maximum of one broadcast slot per iteration. Starting with an interval (L,U], each station has a search parameter x_i in the interval. The stations maintain a global window on the interval. Stations with parameters within the window attempt to broadcast their search parameters, and if there are one or more parameters in the window, there will be a contention resolution followed by a broadcast of one of the search parameters. In that case the upper bound of the interval will be updated to the value broadcast. If there are no parameters within the window, the lower bound of the interval is updated to the upper bound of the window used, and the protocol continues. The minimum is identified when the lower bound of the interval is equal to the upper bound. The steps each station performs are outlined in Figure 3.

An example of the one-broadcast strategy is shown in Figure 4. There are five processors contending, and station i has contention parameter x_i. In this example, $x_1=0.48$, $x_2=0.90$, $x_3=0.35$, $x_4=0.30$, and $x_5=0.75$. These contention parameters were chosen arbitrarily, but for different purposes they may reflect processor loads or priorities. For the first iteration, the upper bound of the window chosen is 0.51. Stations 1, 3, and 4 attempt to broadcast their parameters. Suppose that station 3 is the winner and

```
procedure one_broadcast_strategy;
/* choose_window()        function returning choice for the upper bnd of next window
 * broadcast_state()      procedure returning the results of the broadcast slot
 * broadcast()            procedure to contend and broadcast argument
 * parameter_broadcast    search parameter b'cast if broadcast_state returns transmission
 * lb_window              lower bound for window to be chosen
 * ub_window              upper bound for window to be chosen
 * window_bd_up           actual upper bound of window chosen
 */
searching = true;
lb_window = L;
ub_window = U;
while searching do {
        window_bd_up = choose_window(lb_window, ub_window);
        if (search_parameter $<=$ window_bd_up) and
                        (search_parameter > lb_window) then {
                broadcast(search_parameter);
                broadcast_state(state, parameter_broadcast);
                if (state = idle) then
                        lb_window = window_bd_up
                else if (state = transmission) then
                        ub_window = parameter_broadcast;
        } else
                searching = false;
        if (ub_window = lb_window) then
                searching = false;
}
```

Figure 3. The One-Broadcast Strategy

transmits. The next interval to be searched is $(0,x_3)$. Let the upper bound of the next window chosen be 0.33. Only station 4 trys to transmit its parameter, and x_4 is broadcast. The search has not concluded even though x_4 is the minimum because the fact that it was the only station broadcasting is not available to station 4 or to the other stations. The next window chosen is 0.25. There is, of course, no broadcast. This process will continue until the bounds of the window isolate x_4, and the minimum is globally known.

The choice of the window in each broadcast slot is based on the probabilities of the two states, transmission and idle, which are dependent on previous broadcasts. If a previous broadcast slot resulted in the transmission of a value, say x_{b1}, then any subsequent transmissions must be less than x_{b1}. This implies that any subsequent x_is broadcast were eligible to broadcast during the iteration that x_{b1} was broadcast, but lost the contention. The probability of the subsequent transmissions must be conditioned on the fact that any x_is in the current window did not broadcast when they were eligible during previous iterations. The choice of the window is, thus, dependent on previous broadcasts; hence, the Principle of Optimality is violated.

Assume that station i has an independent search parameter x_i with distribution F(x) and density f(x). The following definitions are used to formulate the problem of choosing the upper bound of the next window as a recurrence after k broadcasts.

$N_{B1}(a,b,v^k,q^k)$: the minimum expected number of broadcast slots to isolate the minimum x_i using a one-broadcast strategy, given that the current interval under consideration is (a,b], that there is at least one x_i in (a, b], and that there have been k previous broadcasts with values and corresponding upper bounds of windows stored in the k-element arrays v^k and q^k, respectively;

$\phi_B(a,b,w,v^k,q^k)$: the probability of a transmission on the interval (a,w], given that there have been k previous broadcasts with values and corresponding upper bounds of windows stored in the k-element arrays v^k and q^k, respectively;

$\theta_B(a,b,w,v^k,q^k)$: the probability of idle on the interval (a,w], given that there have been k previous broadcasts with values and corresponding upper bounds of windows stored in the k-element arrays v^k and q^k, respectively.

The notation v^k and q^k indicates a set of k values broadcast and the corresponding upper bounds of windows used. v_k and q_k are the k^{th} value and the corresponding upper bound of window used. It follows directly from the above definitions that

$$\phi_B(a,b,w,v^k,q^k) + \theta_B(a,b,w,v^k,q^k) = 1.0. \qquad (1)$$

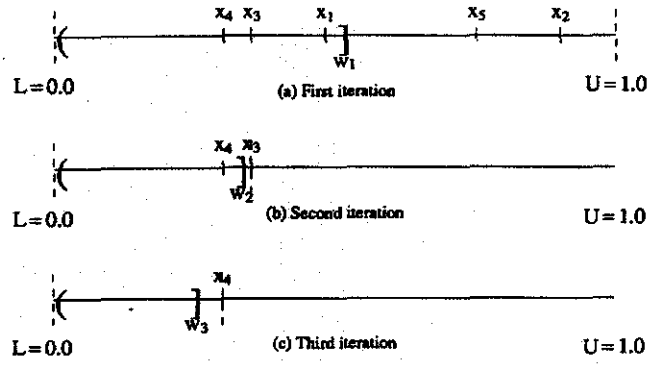

Figure 4. Example illustrating the One-Broadcast Strategy

After k successful broadcasts, there are 2(k+1) subintervals on the interval (a,U]. They are (a, w], (w, v_k], (v_k, q_k], (q_k, v_{k-1}], ..., (v_1, q_1], and (q_1, U]. For reference, they can be numbered from left to right and from 1 to 2(k+1). Let s be a set of elements { s_i }, where s_i is the number of x_is in the i^{th} subinterval. Let S be the set of s that are possible with the previous windows and values broadcast, and let I be a subset of S such that s_1=0. The set I is the subset of S that corresponds to a distribution of x_is, such that there will be no x_is in (a, w] and that the result of the broadcast slot is idle. Then

$$\theta_E = \frac{\sum_{s\in I}\left[\Pr[\text{arrangement } s]\sum_{i=1}^{k}\Pr[v_i \text{ broadcast with } w = q_i \mid s]\right]}{\sum_{s\in S}\left[\Pr[\text{arrangement } s]\sum_{i=1}^{k}\Pr[v_i \text{ broadcast with } w = q_i \mid s]\right]}. \quad (2)$$

The probability of a given arrangement is found using the distribution function F(x). Let b(a,b,i) = [F(b) − F(a)]^i then

$$\Pr[\text{arrangement } s] = \binom{n}{s_1} b(a,w,s_1)\binom{n-s_1}{s_2} b(w,v_k,s_2)\cdots \binom{n-\sum_{i=1}^{2k} s_i}{s_{2k+1}} b(v_1,q_1,s_{2k+1})b(q_1,U,s_{2k+2}). \quad (3)$$

$\Pr(v_k$ broadcast with a window upper bound $q_k \mid s)$ is easily determined because each station in the subinterval searched has equal probability of winning and broadcasting in a broadcast slot, so

$$\Pr(v_i \text{ broadcast with a window upper bound } q_i \mid s) = \frac{1}{\sum_{j=1}^{2(k-i+1)} s_j}. \quad (4)$$

Using a conditional density function

$$f_c(a,w,x_b) = \frac{f(x_b)}{\Pr(a<x_b\le w)} = \frac{f(x_b)}{F(w)-F(a)}, \quad (5)$$

the choice of the upper bound of the next window is formulated as a recurrence. Let

$$\gamma(a,w,v^k,q^k) = \int_a^w f_c(a,w,x_b)N_{E1}(a,x_b,v^{k+1},q^{k+1})dx_b \quad (6)$$

then

$$N_{E1}(a,b,v^k,q^k) = \min_{a<w<b}\left\{1 + \phi_E(a,b,w,v^k,q^k)\gamma(a,w,v^k,q^k) + \theta_E(a,b,w,v^k,q^k)N_{E1}(w,b,v^k,q^k)\right\}, \quad (7)$$

with

$N_{E1}(a,b,v^k,q^k) = 1$ for all b=a.

The first term on the right hand side of Eq. 7 counts the current broadcast. The second term is the expected number of additional broadcast slots to isolate the minimum if the current broadcast slot results in a transmission. γ is the weighted average number of broadcast slots for x_b, the value broadcast, and the probability that this value was broadcast. The third term is the number of additional broadcast slots if the current broadcast slot is idle.

Boundary conditions must be set to terminate the evaluations after a reasonable number of broadcast slots. In practice, the x_is may represent indistinguishable physical measures when their difference is less than δ. It is assumed that when the window size is smaller than δ, the probability that two stations have generated parameters in this interval is so small that contention can always be resolved in one step. The boundary condition becomes

$N_{E1}(a,b,v^k,q^k) = 1$ for all $(b-a) < \delta$.

Using a δ=1/(10n), the evaluation of the recurrence equation is complex.

The data structure shown in Figure 5 is used to store window choice information. The top of the structure contains pointers to initial windows for different numbers of processors. For a given n_i, there is an initial window $w_{1,1}$, and two pointers to substructures corresponding to the two outcomes, transmission and idle. Note that the substructure for a transmission contains windows for each of the possible values that can be transmitted in the subinterval.

The data structure shows the final window choices, but during computation of the best window choice for each decision point, all possible choices of windows have to be tried. For each possible window, there can either be a transmission or no transmission. If there is a transmission, all possible values within the window must be considered as the possible value broadcast. Each level of the structure indicates the outcome of an iteration. For every decision in the exact solution, the entire structure above the current decision point, which is determined by v^k and q^k, must be taken into consideration in computing the next set of branching probabilities. With δ = 1/(10n), and n=5, there are 69,007,690 decision points, and for n=6, the number increases to 8,501,194,726. The number of decision points increases so rapidly that the problem becomes intractable. Fortunately, reasonable results can be obtained using a heuristic decision based on the current upper and lower bounds only.

In the approximate solution, the probabilities of transmission and idle are assumed to be independent of previous broadcasts and are computed without information from previous broadcasts. The following definitions are used.

$N_{A1}(a,b)$: the minimum expected number of broadcast slots to isolate the minimum x_i using an approximate solution of the one-broadcast strategy, given that all x_is are in (a,U], and that at least one x_i is in (a,b];

$\phi_A(a,b,w)$: the probability of a transmission on the interval (a,w], given that all x_is are in (a,U], and that at least one x_i is in (a,b];

$\theta_A(a,b,w)$: the probability of no transmission on the interval (a,w], given that all x_is are in (a,U], and that at least one x_i is in (a,b].

It is obvious that

$$\phi_A(a,b,w) + \theta_A(a,b,w) = 1.0. \quad (8)$$

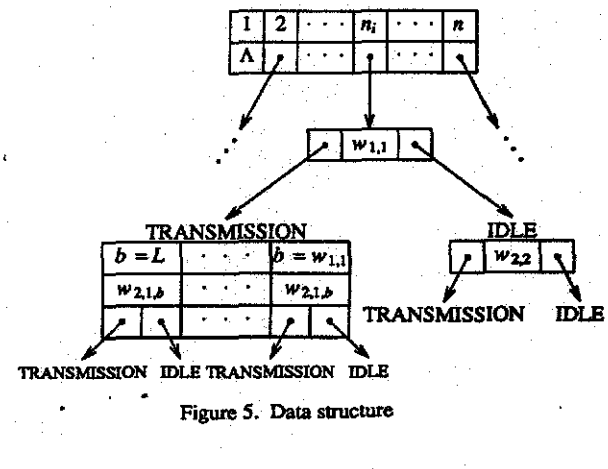

Figure 5. Data structure

There are two cases to consider when calculating $\theta_A(a,b,w)$, namely, $b=U$ and $b \neq U$. When $b=U$, it is uncertain whether there is a x_i at b, and all arrangements of the n x_is must be considered.

$$\theta_A(a,U,w) = \frac{(F(U)-F(w))^n}{(F(U)-F(a))^n}. \quad (9)$$

When $b \neq U$, there must be a station at b, since b is only updated to a value of x_i in the event of a transmission. In this case we are only concerned with the placement of at most (n−1) of the x_is.

$$\theta_A(a,b,w)\Big|_{b\neq U} = \frac{(F(U)-F(w))^{n-1}}{(F(U)-F(a))^{n-1}}. \quad (10)$$

The recurrence for choosing the window is

$$N_{A1}(a,b) = \min_{a<w<b}\left\{ 1+\phi_A(a,b,w)\left[\int_a^b f_c(a,w,x_b)N_{A1}(a,x_b)dx_b\right] + \theta_A(a,b,w)N_{A1}(w,b)\right\}. \quad (11)$$

Again, the three terms on the right hand side of the above equation count the current broadcast slot, additional broadcast slots in the event of a transmission, and additional broadcast slots if the current broadcast slot is idle.

The assumption that contention can be resolved in one step when the window size is smaller than δ holds, so the boundary condition

$$N_{A1}(a,b) = 1 \quad \text{for all } (b-a) < \delta$$

is used again. The data structure for storing the windows is simply a two-dimensional array. The number of decision points for the approximate solution is determined by the values of a and b. The total number of unique nodes with $\delta=1/(10n)$ is $((10n)^2+30n)/2$, which is determined by counting the decision points indicated by the above recurrences. For $n=5$ and $n=6$, the numbers of decisions points are 1325 and 1890, respectively, and the complexity of the solution is considerably reduced from the exact solution.

The two-broadcast and combined strategies differ from the one-broadcast case in the number of broadcast slots per iteration. For the two-broadcast strategy, two broadcast slots are allowed per iteration. The first broadcast slot uses a window choice similar to the one-broadcast strategy The second broadcast slot is used only in the event that there was a transmission on the first broadcast slot; if the result of the first broadcast slot is idle, the search continues to the next iteration. All stations with x_i less than the value broadcast during the first broadcast slot are allowed to broadcast. If there is no transmission during the second broadcast slot, the global minimum is the value transmitted during the first broadcast slot; if there is a transmission on the second broadcast slot, the search continues to another iteration. The objective in investigating the two-broadcast strategy is to truncate the search in the event that the minimum is broadcast in the first broadcast slot. The combined strategy uses both the one-broadcast and two-broadcast strategies. In each iteration, the appropriate strategy along with the window is selected. The objective here is to achieve the benefits of both strategies.

The simulation results for the distributed window search are shown in Figure 6. The windows were generated using the equations derived above. The broadcast parameters were generated from a uniform distribution in (0,1], and sufficient cases were simulated until a confidence interval of 0.95 was reached. The number of broadcast slots is bounded by 2.6 for the one-broadcast strategy. Results of simulating the two-broadcast and combined strategies are also shown in Figure 6. The performance of the two-broadcast strategy is bounded by 2.7 broadcast slots, so it is not as good as the one-broadcast strategy. The performance of the combined strategy is the same as the one-broadcast strategy, but it has higher overhead. In summary, the one-broadcast strategy is superior in performance.

The proposed scheme is practical as a result of the constant expected number of broadcast slots. The time required for a contention slot is approximately 50 microseconds, and the time required to broadcast a search parameter is estimated at approximately 100 microseconds. If it takes 120 microseconds to resolve contention and 100 microseconds to transmit a 1-Kbyte packet, then each load balancing decision to identify the maximally and the minimally loaded processors requires less than 1 millisecond.

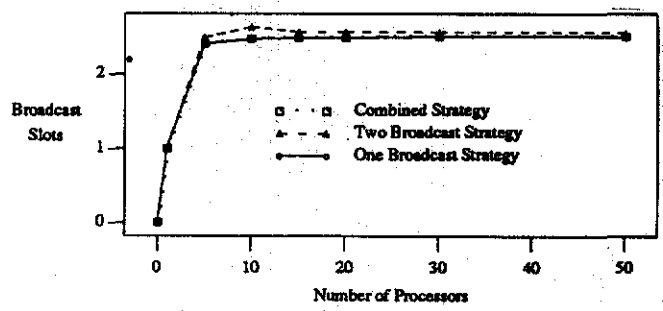

Figure 6. Results of the Protocol Simulation

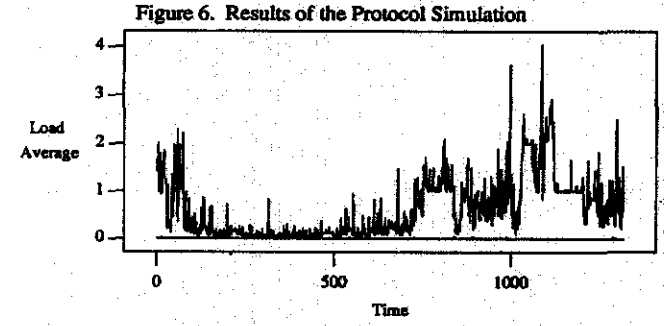

Figure 7. Load Maximums and Minimums for One Day

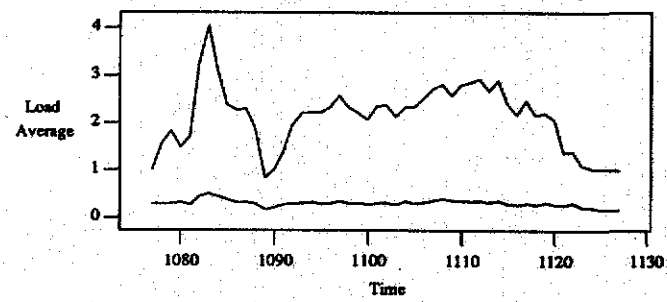

Figure 8. Load Maximums and Minimums for One Hour

2.4. Distribution of Load Averages

The knowledge on the distribution of workload (or load averages) is needed in the distributed search in order to choose the windows. In this section, the load averages on a system experiencing a real workload are studied to determine its characteristics and whether global scheduling has potential to improve performance.

The study consisted of measuring the load on a system of 10 Sun workstations (servers and clients). Every sixty seconds, the one-minute load average was measured and logged. The load data was analyzed using an adjusted Komolgorov-Smirnov test [LaK82, Tri82]. This goodness-of-fit test can be used to detect differences between a normal distribution and the empirical distribution indicated by the measured data. The agreement was measured over time. The results indicate that the distribution of load averages can be estimated using a normal distribution, as 80 percent of the time that the system is active, the distribution is within 0.215 of a normal distribution.

The results of the study are shown in Figures 7 and 8. Figure 7 shows the maximum and minimums load averages over time. The minimum is almost always zero, and the maximum varies. The peak utilization is between sample number 700 and 1400 which reflects the load from 1 p.m. until midnight. From 1 a.m. until 11 a.m., the loads measured were uniformly low (≤ 1). There is a potential benefit from global scheduling during that time. Figure 8 shows one hour of the minimum, average, and maximum load averages.

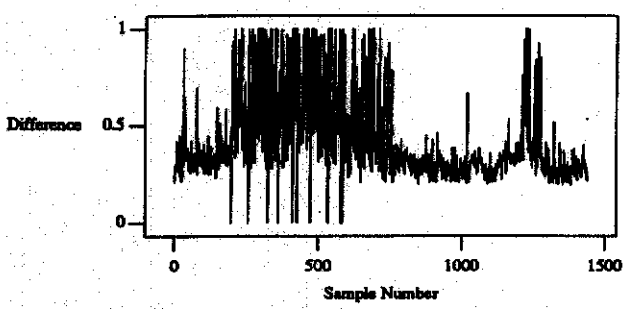

Figure 9. Difference Between the Empirical and Normal Distributions

3. IMPLEMENTATION

The Sun system on which the global scheduling strategy has been implemented consists of servers and clients connected by Ethernets. The servers have secondary disk storage, and the clients do not. A client can access a server's disk via the network, and is allocated a portion of the server's disk for swap space. Swapping over the network is a part of the regular message transfer. A network file system (NFS) allows transparent access to remote file systems. This mechanism allows uniform access by the clients to secondary storage.

The Sun system is multiprogrammed. When a process is initiated on a server or a client, a core image of that process containing run-time information exists in the swap space associated with that server or client. This core image may be quite large (on the order of several megabytes). Since it is not possible to copy from one swap space to another, the only way to transfer the core image is over the network. Moreover, due to the size of the core image, this transfer would have high overhead (on the order of seconds). Owing to this overhead, the current implementation migrates jobs only at their entry point. Future enhancements to the global scheduling strategy will study preemption strategies.

There are two program modules (daemons) that comprise the scheduling strategy in the current implementation: the searcher, and the job migrator. The search daemon periodically participates in a search for the minimum. The search at a processor is initiated in one of two ways. First, when an alarm signal occurs, the daemon determines if its local load is in $(L, w_1]$. If it is, then the job is transmitted. Second, the daemon is activated when a packet arrives from another processor. The search daemon obtains load information (in the current implementation the load average) directly from the kernel. Job migration is performed by sending the necessary information of a job at entry point to a remote processor. Result return is performed when execution of the migrated job is completed. Priorities of the various phases in load balancing are not enforced due to the high overhead of explicit priority resolution (as discussed in Section 2).

3.1. Implementation of the Distributed Search

The search for the minimum load normally takes 2.7 broadcast slots on the average (see Section 2). Table 2 shows the simulation results of the search truncated at one broadcast slot. This table shows that, under this condition, the absolute minimum is located 70 to 78 percent of the time, and there are 18.6 to 23.1 percent of the unresolved cases in which all search parameters are in the interval $(w_1, U]$ and no workload information is broadcast. Unresolved searches are not critical here because they reflect a condition in which no processors are lightly loaded enough to accept additional jobs. For cases that are resolved, those that do not find the absolute minimum have a one to two percent difference from the absolute minimum. Since workload information is heuristic in nature, small errors in identifying processors for load balancing are not critical. An important tradeoff we have achieved is that reasonable results are obtained at substantially lower overhead.

The minimum search as implemented on the Sun network is shown in Figure 10. Execution is initiated, as mentioned above, by an alarm signal, or by a packet arriving from another search module. When an alarm is received, the processor's current load is compared to the lower bound of the window. If the load is small than the lower bound, it is broadcast with a timestamp and processor address. If a packet is received, the load is

```
/* INTERVAL constant indicating time interval at which search is performed
 * packet{      information contained in the packets exchanged
 *      load    processor load
 *      source  processor address
 * }
 * load_status { information contained in load status
 *      timestamp           timestamp of the load status
 *      minimum_load        load at the minimally loaded processor
 *      minimum_location    location of the minimally loaded processor
 * }
 * current_time()      function returning the current time
 * current_load()      function returning the current load at a processor
 * set_alarm()         function to set an alarm signal ARGUMENT seconds from now
 * on_alarm()          function that sets a call to the argument when an alarm occurs
 * reset_alarm()       function to reset the alarm signal to ARGUMENT seconds from now
 * wait_arrival()      function that waits for the arrival of a packet
 * write()             writes to a file
 */
structure load_status load_status;      /* global structure containing load status */
procedure main;
        on_alarm(send_packet());
        set_alarm(INTERVAL);
        while (TRUE)
                receive_packet();
        endwhile
end
procedure send_packet;
        if (current_load() < window) then
                packet.load = current_load();
                packet.source = MY_ADDRESS;
                broadcast(packet);
        endif
        set_alarm(INTERVAL);
end
procedure receive_packet();
        wait_arrival(packet);
        reset_alarm(INTERVAL);
        if (current_time() > load_status.time) then
                load_status.minimum_load = packet_load;
                load_status.minimum_location = packet_source;
                load_status.timestamp = current_time();
                write(load_status);
        endif
end
```

Figure 10. Procedures for minimum search

Table 2. Results of Simulation for the Truncated Search

no. of procs	% of srches unreslvd	% reslvd srches finding min	av rank of result	% diff of result from min
3	20.0	78.00	0.194	2.790
4	18.6	76.04	0.224	2.192
5	20.1	73.97	0.268	2.086
6	21.8	73.66	0.252	1.665
7	21.4	72.26	0.279	1.506
8	20.5	70.57	0.292	1.495
9	23.1	72.82	0.256	1.151
10	21.8	74.04	0.256	0.959

accepted as the minimum and is stored with the current time. If more than one processor sends a load packet due to their both receiving an alarm simultaneously, the minimum load is accepted. If the loads are identical, the processor address is used as a tie breaker. The alarm is set when packets are received, so the process is loosely synchronized.

The contents of received packets are written to file to reflect the current status of the network. As the status (the minimum load, and the location of the processor with the minimum load) is stored, the current time is also stored. When the status information is read from the file, the time stamp is checked and is used to determine if the load value is out of date. If the status information is out of date, the processor considers itself ineligible for job migration until it receives another status packet. This time stamping allows an unreliable communication mechanism to be used to communicate load information (broadcast datagrams) which reduces network traffic in the form of acknowledgements. In case that load information is lost by one processor, that processor does not migrate packets. This has minimal affect on the overall performance of the scheduling strategy. Moreover, since packet loss is a relatively rare occurrence, acknowledgements are an unnecessary price to pay.

The search was tested on two Sun systems: one with a server (Aquinas) and two clients (Calvin and Hobbes), and one with a server (Dwarfs) and ten clients. On Aquinas, the search was resolved in 50 to 80 milliseconds, and on Dwarfs, the searches were resolved in 150 to 180 milliseconds. The performance on Dwarfs can be considered the worst case because the Ethernet cable the clients are connected with is at the allowable length limit. The results for both Aquinas and Dwarfs were consistent with the simulation results in which the absolute minimum was located 70 percent of the time for resolved searches.

The search for the maximum load is not performed explicitly. It was observed that if a processor's load was above the initial window in the minimum search, its load was the maximum the majority of the time. As a result, processors with loads above the initial window can migrate the current job if the minimum load is current, and an explicit search of the maximally loaded processor is not needed.

A potential problem with not explicitly identifying the maximally loaded processor is that a lightly loaded processor may be swamped by jobs from more heavily loaded ones. There are two solutions to resolve this problem. First, a processor may only be allowed to migrate jobs if it has a load higher than the upper bound of the initial window and a new arrival. Second, a processor may be allowed to migrate at most one job between searches. Our performance data indicate that swamping is not a problem for a moderate number of participating processors. However, if preemption were implemented, it would be necessary to identify a unique maximum, as a large fraction of the processors may be preempting jobs at any time.

Another obvious technique for distributing status information is to broadcast it periodically, as is done with the **rwhod** daemon in Unix. To determine the savings of using the proposed method as opposed to using the technique of the **rwhod** daemon, the resource utilization of both was measured for the period of one hour. The overhead is summarized in Table 3. For comparison, a daemon that only broadcasts the load average value is studied (as opposed to the **rwho** daemon which broadcasts other information as well). These results indicate that both **rwhod** and the simple broadcast daemon introduce considerably more communication overhead than the search daemon proposed here.

3.2. Implementation of Job Migration and Result Return

In the general case, result return and job migration require that input file (and executable files if necessary) be sent over the network from the source processor to the destination processor, and that output files be sent back. In the Sun environment, it is not necessary to send files from the source to the destination because all processors and clients have access to the same secondary storage. In this environment, job migration requires sending the command from the source to the destination, and result return requires sending any error information back.

The job migration decision is made as shown in Figure 11. First the current load of the processor is compared to the first window for the maximum search. If the load is above the first window, the global minimum location and its timestamp are read from a local file. If the timestamp indicates the minimum is current, the job is executed on the processor with the minimum load using a remote shell. This causes any error messages associated with the remote execution to be sent back directly. The remote shell uses a reliable communication mechanism (TCP/IP protocol), so execution of the job is guaranteed.

3.3. Performance of the Scheduling Strategy

In this section, the response time of a system with global scheduling is compared to the response time of a system without global scheduling. The system used was the server, Aquinas, with the two clients, Calvin and Hobbes.

First, workload was generated by processes that either initiated a CPU-bound job or slept for the amount of time the job consumed during its last execution, with equal probability. As these processes were executed, the load and the amount of time consumed by executing processes was tabulated. Also, a history of the initiation of jobs was created. During the second portion of the test, the global scheduling strategy was enabled, and the jobs were initiated according to the history. Again the load was monitored, and the execution time tabulated.

Table 3. Comparison of Overhead for Status Distribution

Deamon	System Time Seconds	User Time Seconds	Number of Broadcasts
search	0.1094	0.0137	65
broadcast	0.2593	0.0316	181
rwho	0.2633	0.0613	181

Table 4. Performance of the Global Scheduling Strategy

Name	Job Time w/out Global Sched.	Job Time with Global Sched.	% Diff
Aquinas (server)	18963	18540	2.23
Calvin (client)	12143	10430	14.10
Hobbes (client)	7319	5912	19.22
Totals	38425	34882	9.2

```
/* INTERVAL constant indicating time interval at which search is performed
 * THRESHOLD      constant indicating minimum difference between loads for load balancing
 *
 * load_status{
 *      timestamp          timestamp of the load status
 *      minimum_load       load at the minimally loaded processor
 *      minimum_location   location of the minimally loaded processor
 * }
 * current_time()     function returning the current time
 * current_load()     function returning the current load at a processor
 * read()             function to read from a file
 * execute_locally()  function to execute a command at the current processor
 * execute_remotely() function to execute a command at a remote processor
 */
procedure migrate(command);
        read(load_status);
        if (current_time() > (load_status.timestamp + INTERVAL) then
                execute_locally(command);
        else if (current_load() > $w sub 1 sub max$) and
                        (current_load() > (load_status.load + THRESHOLD)) then
                execute_remotely(command, load_status.location);
        endif
end
```

Figure 11. Procedure for job migration

The results as summarized in Table 4 show a small improvement for the server, and a much larger improvement for the clients. This is expected as the server is a much faster computer than the clients, and has direct access to the secondary storage through the VME bus rather than the Ethernet. As a result, the clients will benefit more by sending jobs to the server.

4. CONCLUDING REMARKS

The problem of global scheduling on distributed computing systems has been studied. An efficient distributed technique for determining the extremum of a set of numbers has been developed, and can find the extremum in 2.7 iterations on the average, independent of the number of participating stations. The search technique can be used to efficiently distribute status information for scheduling decisions. A load balancing strategy using the distributed search was designed and implemented on a network of Sun workstations.

Areas for future research include adding preemption capabilities, investigating the problem of an accurate load measure, and predicting job requirements. Preemption would allow redistribution when load imbalance occurs between arrivals, especially in the case of longer jobs. This would increase the utility of the load balancing strategies. The problem of an accurate load measure is important to the potential improvement of global scheduling strategies. Although the number of processes in the run queue is a good measure, other factors contribute to the load such as physical characteristic of the participating processors and local activities, such as page faults, percentage of the CPU that is utilized, and percentage of the main memory that is utilized. Another problem that has not been adequately addressed to date is that of estimating the future requirements of a job. This will allow scheduling decisions to be based on future activity rather than solely upon past events.

ACKNOWLEDGEMENT

We gratefully acknowledge the help of Ralph Kling, who has participated in developing the implementation on the network of Sun computers.

REFERENCES

[ApI78] W. F. Appelbe and M. R. Ito, "Scheduling Heuristics in a Multiprogramming Environment," *IEEE Trans. Comp.*, July 1978, pp. 628-637.

[BaW85] K. M. Baumgartner and B. W. Wah, "The Effects of Load Balancing on Response Time for Local Computer Systems with a Multiaccess Network," *Proc. Int'l. Conf. Communications*, June 1985, pp. 10.1.1-10.1.5.

[BaW87] K. M. Baumgartner and B. W. Wah, "Window Protocols for Load Balancing on a System with a Local Multiaccess Network," *Proc. Int'l. Conf. Parallel Processing*, Aug. 1987, pp. 851-858.

[Bak74] K. A. Baker, *Introduction of Sequencing and Scheduling*, John Wiley and Sons, Inc., New York, 1974.

[Bok81] S. H. Bokhari, "On the Mapping Problem," *IEEE Trans. Comp.*, Mar. 1981, pp. 207-214.

[CaK88] T. Casevant and J. Kuhl, "A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems," *IEEE Trans. on Software Engineering*, Vol. 14, Feb. 1988, pp. 141-154.

[ChA82] T. C. K. Chou and J. A. Abraham, "Load Balancing in Distributed Systems," *IEEE Trans. Software Engineering*, Vol. SE-8, July 1982, pp. 401-412.

[ChH80] W. W. Chu, L. J. Holloway, M. T. Lan, and K. Efe, "Task Allocation in Distributed Data Processing," *IEEE Computer*, Nov. 1980, pp. 57-68.

[ChK79] Y. C. Chow and W. Kohler, "Models for Dynamic Load Balancing in a Heterogeneous Multiple Processor System," *IEEE Trans. Comp.*, Vol. C-28, May 1979, pp. 354-361.

[CoM67] R. W. Conway, W. L. Maxwell, and L. W. Miller, *Theory of Scheduling*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1967.

[Cof76] E. G. Coffman, Jr., *Computer and Job-Shop Scheduling Theory*, John Wiley and Sons, Inc., New York, 1976.

[EaL86] D. L. Eager, E. D. Lazowska, and J. Zahorjan, "Adaptive Load Sharing in Homogeneous Distributed Systems," *IEEE Trans. Software Engineering*, Vol. SE-12, May 1986, pp. 662-675.

[Efe82] K. Efe, "Heuristic Models of Task Assignment Scheduling in Distributed Systems," *IEEE Computer*, June 1982, pp. 50-56.

[Ezz86] A. K. Ezzat, "Load Balancing in Nest: A Network of Workstations," *Int'l Conf. on Distributed Computing Systems*, IEEE, 1986, pp. 1138-1149.

[GoR72] M. J. Gonzalez, Jr. and C. V. Ramamoorthy, "Parallel Task Execution in a Decentralized System," *IEEE Trans. Comp.*, Vol. c-21, Dec. 1972, pp. 1310-1322.

[Gon77] M. J. Gonzalez, Jr., "Deterministic Processor Scheduling," *ACM Computing Surveys*, Vol. 9, Sept. 1977, pp. 173-204.

[GrL77] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan, "Optimization and Approximation in Deterministic Sequencing and Scheduling," *Proceedings of Discrete Optimization*, Aug. 1977, pp. 47-98.

[HwC82] K. Hwang, W. Croft, B. W. Wah, F. A. Briggs, W. R. Simmons, and C. L. Coates, "A Unix-Based Local Computer Network With Load Balancing," *IEEE Computer*, Vol. 15 , Apr. 1982, pp. 55-66.

[JuW84] J. Y. Juang and B. W. Wah, "Optimal Scheduling Algorithms for Resource Sharing Interconnection Networks," *Computer Software and Applications Conference*, IEEE, Nov. 1984, pp. 34-43.

[KeL84] R. M. Keller and F. C. H. Lin, "Simulated Performance of Reduction-Based Multiprocessors," *IEEE Computer*, July 1984, pp. 70-82.

[Kle75] L. Kleinrock, *Queueing Systems Volume 1: Theory*, John Wiley and Sons, Inc., New York, N.Y., 1975.

[KrH80] A. Kratzer and D. Hammerstrom, "A Study of Load Leveling," *Proc. IEEE COMPCON*, Fall 1980, pp. 647-654.

[LaK82] A. M. Law and D. W. Kelton, *Simulation Modeling and Analysis*, McGraw-Hill, New York, 1982.

[LiM82] M. Livney and M. Melman, "Load Balancing in Homogeneous Broadcast Distributed Systems," *Proc. ACM Computer Network Performance Symposium*, 1982, pp. 47-55.

[LoL81] V. Lo and J. W. S. Liu, "Task Assignment in Distributed Multiprocessor Systems," *Int'l. Conf. Parallel Processing*, Aug. 1981, pp. 358-360.

[MaL82] P. Y. R. Ma, E. Y. S. Lee, and M. Tsuchiya, "A Task Allocation Model for Distributed Computing Systems," *IEEE Trans. Comp.*, Vol. C-31, Jan. 1982, pp. 41-47.

[MeB76] R. M. Metcalfe and D. R. Boggs, "Ethernet : Distributed Packet Switching for Local Computer Networks," *Communications of the ACM*, Vol. 19, July 1976, pp. 395-404.

[NiH81] L. M. Ni and K. Hwang, "Optimal Load Balancing Strategies for a Multiple Processor System," *Proc. Int'l. Conf. Parallel Processing*, Aug. 1981, pp. 352-357.

[RaC72] C. V. Ramamoorthy, K. M. Chandy, and M. J. Gonzalez, Jr., "Optimal Scheduling Strategies in a Multiprocessor System," *IEEE Trans. Comp.*, Vol. C-21, Feb. 1972, pp. 137-146.

[RaS79] G. S. Rao, H. S. Stone, and T. C. Hu, "Assignment of Task in a Distributed Processor System with Limited Memory," *IEEE Trans. Comp.*, Vol. C-28, Apr. 1979, pp. 291-299.

[RaS84] K. Ramamritham and J. A. Stankovic, "Dynamic Task Scheduling in Hard Real-Time Distributed Systems," *IEEE Software*, Vol. 1, July 1984, pp. 65-75.

[StB78] H. S. Stone and S. H. Bokhari, "Control of Distributed Processes," *IEEE Computer*, July 1978, pp. 97-105.

[Sta85] J. A. Stankovic, "An Application of Bayesian Decision Theory to Decentralized Control of Job Scheduling," *IEEE Trans. Comp.*, Vol. C-34, Feb. 1985, pp. 117-130.

[Sto77] H. S. Stone, "Multiprocessor Scheduling with the Aid of Network Flow Algorithms," *IEEE Trans. Software Engineering*, Vol. SE-3, Jan. 1977, pp. 85-93.

[Sto78] H. S. Stone, "Critical Load Factors in Two-Processor Distributed Systems," *IEEE Trans. Software Engineering*, Vol. SE-4, May 1978, pp. 254-258.

[TaT85] A. N. Tantawi and D. F. Towsley, "Optimal Static Load Balancing in Distributed Computer Systems," *Journal of the ACM*, Vol. 32, Apr. 1985, pp. 445-465.

[Tan81] A. S. Tanenbaum, *Computer Networks*, Prentice Hall Inc., N.J., 1981.

[Tri82] K. S. Trivedi, *Probability and Statistics with Reliability, Queuing, and Computer Science Applications*, Prentice-Hall Inc., Englewood Cliffs, N.J., 1982.

[WaJ83] B. W. Wah and J. Y. Juang, "An Efficient Protocol For Load Balancing on CSMA/CD Networks," *Proc. Eighth Conference on Local Computer Networks*, IEEE, Oct. 1983, pp. 55-61.

[WaJ85] B. W. Wah and J. Y. Juang, "Resource Scheduling for Local Computer Systems with a Multiaccess Network," *IEEE Trans. Comp.*, Dec. 1985, pp. 1144-1157.

[WaM85] Y. T. Wang and R. J. T. Morris, "Load Sharing in Distributed Systems," *IEEE Trans. Comp.*, Vol. C-34, Mar. 1985, pp. 204-217.

[Weg80] J. Welgarz, "Multiprocessor Scheduling with Memory Allocation--A Deterministic Approach," *IEEE Trans. Comp.*, Aug. 1980, pp. 703-709.

[Wil83] E. Williams, "Assigning Processes to Processors in Distributed Systems," *Proc. Int'l. Conf. Parallel Processing*, 1983, pp. 404-406.

[Zho86] S. Zhou, "A Trace Driven Study of Dynamic Load Balancing," *University of California - Berkeley Technical Report #UCB/CSD 87/305*, , Sept. 1986.