ENTERPRISE:
A MARKET-LIKE TASK SCHEDULER FOR DISTRIBUTED COMPUTING ENVIRONMENTS

Thomas W. Malone
Richard E. Fikes
Michael T. Howard

October 1983

CISR WP #111
Sloan WP #1537-84

Center for Information Systems Research

Massachusetts Institute of Technology
Sloan School of Management
77 Massachusetts Avenue
Cambridge, Massachusetts, 02139

ENTERPRISE:
A MARKET-LIKE TASK SCHEDULER FOR DISTRIBUTED COMPUTING ENVIRONMENTS

Thomas W. Malone
Richard E. Fikes
Michael T. Howard

October 1983

CISR WP #111
Sloan WP #1537-84

Also published as a working paper by the Intelligent Systems Laboratory, Xerox Palo Alto Research Center, October 1983.

Center for Information Systems Research
Sloan School of Management
Massachusetts Institute of Technology

Enterprise: A Market-like Task Scheduler for Distributed Computing Environments

Thomas W. Malone*
Richard E. Fikes
Michael T. Howard*

Working Paper
Cognitive and Instructional Sciences Group
Xerox Palo Alto Research Center

October 1983

Abstract:

This paper describes a system for sharing tasks among processors on a network of personal computers and presents an analysis of the problem of scheduling tasks on such a network. The system, called Enterprise, is based on the metaphor of a market: processors send out "requests for bids" on tasks to be done and other processors respond with bids giving estimated completion times that reflect machine speed and currently loaded files. The system includes a language independent Distributed Scheduling Protocol (DSP), and an implementation of this protocol for scheduling remote processes in Interlisp-D. The Enterprise implementation assigns processes to the best machine available at run-time (either remote or local) and includes facilities for asynchronous message passing among processes. In a series of simulations of different load conditions and network configurations, DSP was found to be substantially superior to both random assignment and a more complex alternative that maintained detailed schedules of estimated start and finish times.

*Now at Massachusetts Institute of Technology.

Enterprise:
A Market-like Task Scheduler for Distributed Computing Environments

Introduction

With the rapid spread of personal computer networks and the increasing availability of low cost VLSI processors, the opportunities for massive use of parallel and distributed computing are becoming more and more compelling. Parallel hardware can often increase overall program speed by concurrently executing independent subparts of an algorithm on different processors [1]. In some cases, parallel search algorithms can improve average execution time even more rapidly than the increase in the number of processors ([24], [28], [47]). With their potential for multiple redundancy, parallel systems can be made much more reliable than their single-processor counterparts. There are also important cases of inherently distributed problem solving where the initial information and resulting actions necessary to solve a problem occur in widely distributed locations (e.g., [3], [7], [39], [48]).

Providing appropriate programming facilities for specifying, controlling, and debugging parallel computations involves a new set of problems. For example, Jones & Schwarz [27] discuss three main classes of problems in multiprocessing systems: (1) *resource scheduling*--how to allocate processor time and memory space, (2) *reliability*--how to deal with various kinds of component failures, and (3) *synchronization*--how to coordinate concurrent activities that depend on each other's actions.

In this paper we describe a prototype system called Enterprise that addresses those problems and present the results of analyzing one aspect of the problem of resource scheduling, namely the scheduling of tasks on processors. This analysis was based on simulations of several alternative task scheduling algorithms operating under various network configurations and load conditions. Although we have focused on decentralized methods for scheduling tasks, our results on this topic are applicable to many forms of parallel computation, regardless of whether or not the processors are geographically separated and whether or not they share memory.

The Enterprise system schedules and runs processes on a local area network of high-performance personal computers. It is implemented in Interlisp-D and runs on the Xerox 1100 (Dolphin), 1108 (Dandelion), and 1132 (Dorado) Scientific Information Processors connected with an Ethernet.

OVERVIEW OF THE ENTERPRISE SYSTEM

A new philosophy for distributed processing

As summarized in Table 1, the traditional philosophy used in designing systems based on local area networks such as Ethernets [34] is to have dedicated personal workstations which remain idle when not used by their owners, and dedicated special purpose servers such as file servers, print servers, and various kinds of data base servers. A system like Enterprise that schedules tasks on the best processor available at run time (either remote or local) enables a new philosophy in designing such distributed systems. In this new philosophy, personal workstations are still dedicated to their owners, but during the (often substantial) periods of the day when their owners are not using them, these personal workstations become general purpose servers, available to other users on the network. "Server" functions can migrate and replicate as needed on otherwise unused machines (except for those such as file servers and print servers that are required to run on specific machines). Thus programs can be written to take advantage of the maximum amount of processing power and parallelism available on a network at any time, with little extra cost when there are few extra machines available.

System Architecture

In order to use this new philosophy, at least the following three facilities must be provided: (1) a way of *communicating* between processes on different machines, (2) a way of *scheduling* tasks on the best available machines (either remote or local), and (3) programming language *constructs* for dealing with remoteness.

As shown in Figure 1, the Enterprise system provides these facilities in three layers of software. The first layer provides an Inter-Process Communication (IPC) facility by which different processes, either on the same or different machines, can send messages to each other. When the different processes are on different machines, the IPC protocol uses internetwork datagrams called PUPs (see [4]) to provide reliable non-duplicated delivery of messages over a "best efforts" physical transport medium such as an Ethernet [34]. Enterprise uses a pre-existing protocol that is highly optimized for remote procedure calls ([2], [42]) in which messages are passed to remote machines as procedure calls on the remote machines.

The next layer of the Enterprise system is the Distributed Scheduling Protocol (DSP) which, using the IPC, locates the best available machine to perform a task (even if the best machine for a task turns out to be the one on which the request originated). Finally, the top layer is a Remote Process Mechanism, which uses both the DSP and IPC to create processes on different machines that can communicate with each other.

Table 1

A new distributed processing philosophy

1. Traditional distributed processing philosophy

    a. dedicated personal workstations

    b. unused workstations are idle

    c. dedicated special purpose servers

2. New distributed processing philosophy

    a. dedicated personal workstations

    b. unused workstations are general purpose servers

    c. special purpose servers only where required by hardware

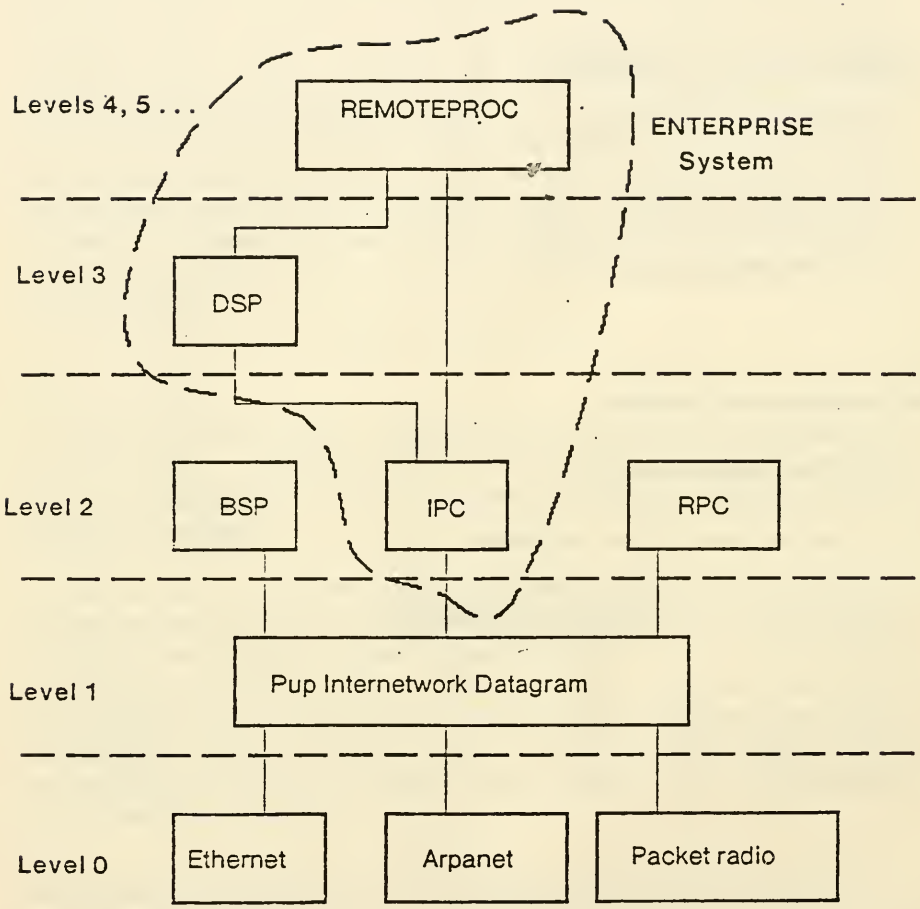

Figure 1. Protocol layers used in the Enterprise system.

Decentralized Task Scheduling

One of the most obvious solutions to the task scheduling problem is to simply assign a task to the first available processor. This approach, or one similar to it is taken in many existing distributed systems (e.g., [2]). As we will see in more detail below, there are three important conditions under which such a simple approach may be radically sub-optimal:

(1) when processors have different capabilities (e.g., different speeds or different files loaded into virtual memory),

(2) when tasks have different priorities (e.g., some subtasks in a search problem are more promising that others), and

(3) when network-wide demand for processor time is high.

The problem of scheduling tasks on processors is, of course, a well-known problem in traditional operating systems and scheduling theory, and there are a number of mathematical and software techniques for solving this problem in both single and multi-processor systems (e.g., [5], [8], [9], [27], [29], [30], [44]). Almost all the traditional work in this area, however, deals with *centralized* scheduling techniques, where all the information is brought to one place and the decisions are made there. In highly parallel systems where the information used in scheduling and the resulting actions are distributed over a number of different processors, there may be substantial benefits from developing *decentralized* scheduling techniques [33]. For example, when a centralized scheduler fails, the entire system is brought to a halt, but systems that use decentralized scheduling techniques can continue to operate with all the remaining nodes. Furthermore, much of the information used in scheduling is inherently distributed and rapidly changing (e.g., momentary system load). Thus, decentralized scheduling techniques can "bring the decisions to the information" rather than having to constantly transmit the information to a centralized decision maker.

Overview of the Distributed Scheduling Protocol

Since many of the problems of coordinating concurrent computer systems are isomorphic to problems in coordinating human organizations ([7], [13], [31], [39]), we have explicitly drawn upon organizational metaphors in the design of the Enterprise scheduling mechanism. In particular, the system's Distributed Scheduling Protocol (DSP) is based on the metaphor of a market (similar to the contract net protocol of Smith and Davis [11], [38], [39] and the scheduler in the Distributed Computing System ([12])). We use the term *clients* for the machines with tasks to be done and *contractors* for the remote server machines (as in [13]). The essential idea is that a client requests bids on a task he wants done, contractors who can do the task respond with bids, and the client selects a contractor from among the received bids. Figure 2 illustrates the messages used in this scheduling process. In the standard case, the following steps occur:

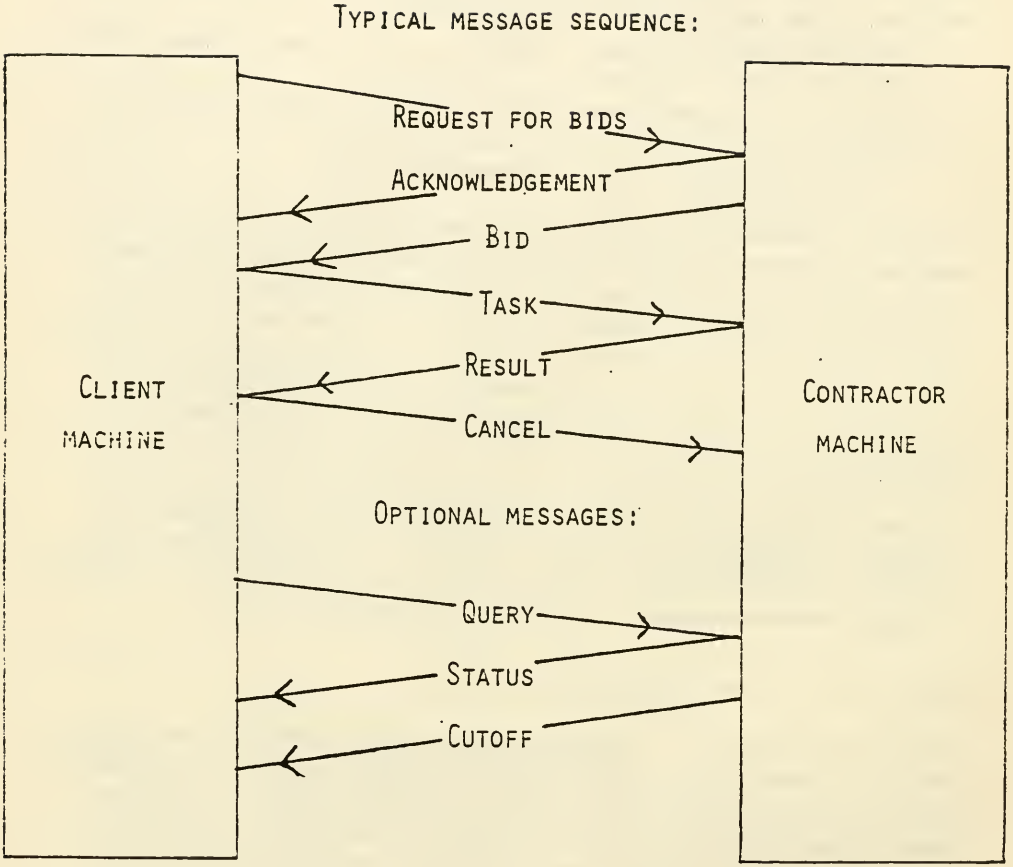

Figure 2. Messages in the Distributed Scheduling Protocol.

1. *The client broadcasts a "request for bids".* The request for bids includes the priority of the task, any special requirements, and a summary description of the task that allows contractors to estimate its processing time.

2. *Idle contractors respond with "bids" giving their estimated completion times. Busy contractors respond with "acknowledgements" and add the task to their queues (in order of priority).*

3. *When a contractor becomes idle, it submits a bid for the next task on its queue.*

4. *When the client receives a bid, it sends the task to the contractor who submitted the bid.* If a later bid is significantly better than the early one, the client cancels the task on the first bidder and sends the task to the later bidder. If the later bid is not significantly better (or if the task has side-effects and cannot be restarted), the client sends a cancel message to the later bidder.

5. *When a contractor finishes a task, it returns the result to the client.*

6. *When a client receives the result of a task, it broadcasts a "cancel" message so that all the contractors can remove the task from their queues.*

If a task takes much longer than it was estimated to take, the contractor aborts the task and notifies the client that it was "cut off." This cutoff feature prevents the possibility of a few people or tasks monopolizing an entire system.

Protection against processor failure. In addition to this bidding cycle, clients periodically query the contractors to which they have sent tasks about the status of the tasks. If a contractor fails to respond to a query (or any other message in the DSP), the client assumes the contractor has failed. Failures might result from hardware or software malfunctions or from a person preempting a machine for other uses. In any case, unless the task description specifically prohibits restarting failed tasks, the client automatically reschedules the task on another machine. Similarly, if a contractor fails to receive periodic queries from one of its clients, the contractor assumes the client has failed and the contractor aborts that client's task.

The "remote or local" decision. In the protocol as described above, if the local machine submits bids for its own tasks (i.e., the client machine offers to be its own contractor), then the local machine would presumably always be the first bidder and would therefore receive every task. To prevent this from happening, the client waits for other bids during a specified interval before processing its own bid. Since contractor machines are assumed to be processing tasks for only one user at a time, the client machine's own bid is also inflated by a factor that reflects the current load on the client machine. (Human users of a processor can express their willingness to have tasks scheduled locally by setting either of these two parameters.)

Comparison with the contract net protocol. Like the contract net protocol ([11], [38], [39]), DSP uses an "announcement, bid, award" sequence. This allows for *mutual selection* of clients and contractors; that is, contractors choose which clients to serve and clients choose which contractors to use. It also allows for *anonymous invocation* where programmers merely describe the requirements of a task without specifying which processor is to perform the task.

The most important difference between DSP and contract nets is that DSP restricts the basis for mutual selection by clients and contractors to two primary dimensions: (1) contractors select clients' tasks in the order of numerical *task priorities*, and (2) clients select contractors on the basis of *estimated completion times* (from among the contractors that satisfy the minimum requirements to perform the job). As we will see below, this specialization of the contract net protocol allows us, among other things, to make some very nice connections with results from traditional scheduling theory about optimality.

Remote Processes Rather Than Remote Procedures

One of the most important issues in designing a system for distributed parallel processing is the choice of language constructs available to the programmer. Most of the proposals for such constructs fall into one of two general classes [35]: (1) procedure-call constructs ([2], [35], [46]), and (2) message-passing constructs ([19], [20], [25]). In most--but not all--cases, message passing systems assume that the objects that receive and send messages are concurrent *processes* with separate "threads" of control, while procedure call systems include the notion of transfer of control from the caller to the callee.

Though there are certainly situations in which remote procedures are a useful programming construct (e.g., we used them as the basis for our message passing primitive), we believe that processes are a more appropriate and powerful abstraction for programming many distributed computations. First, *remoteness is inherently parallel.* Thus, it is appropriate to use the same language constructs for remoteness as for parallelism. For example, process mechanisms typically provide facilities for interprocess communication and for user interaction with processes (e.g., deleting a running process, querying the status of a process, providing input for a process, or receiving output from a process). Those facilities, extended to include remote processes, are crucial to the programmer's ability to specify, debug, and control distributed computations. In addition, they allow a program to be designed without concern for the physical location of each process activation. Second, if a system provides remote processes, it is a trivial matter for users to implement their own remote procedures on top of remote processes, but the reverse is certainly not true. The primary argument in favor of remote procedures seems to be that their simplicity allows them to be implemented more efficiently than remote processes. However, the control and communication facilities available for remote processes make them the construct of choice in most situations.

THE TASK SCHEDULING PROBLEM

DSP is only one of a large class of potentially effective task scheduling protocols, and it is not immediately clear which of the possible schedulers is most desirable. For example, a protocol that allows a contractor to bid on one task while performing another one might be more efficient than DSP. Or, random scheduling might be almost as good as a sophisticated scheduler in most commonly occuring situations. In the development of DSP we analyzed the objectives of the task scheduler and used a simulation program to compare the performance of promising alternative scheduling protocols. In this section we present the results of that study.

Global Scheduling Objectives

Traditional schedulers for centralized computing systems often use *list scheduling* as a basis for layering the design of a system (e.g., [8]). In this approach, one level of the system sequences jobs according to their order in a priority list while the policy decisions about how priorities are assigned are made at a higher level in the system (see [30]). DSP allows precisely the same kind of separation of policy and mechanism. The DSP protocol itself is concerned only with sequencing jobs according to priorities assigned at some higher level. By assigning these priorities in different ways, the designers of distributed systems can achieve different global objectives. For example, it is well known that in systems of identical processors, the average waiting time of jobs is minimized by doing the shortest jobs first [9]. Thus, by assigning priorities in order of job length, the completely decentralized decisions based on priority result in a globally optimal sequencing of tasks on processors.

Optimality results for mean flow time and maximum flow time. Traditional scheduling theory (e.g., [9]) has been primarily concerned with minimizing one of two objectives: (1) the average flow time of jobs (F_{ave})--the average time from availability of a job until it is completed, and (2) the maximum flow time of jobs (F_{max})--the time until the completion of the last job. Minimizing F_{max} also maximizes the utilization of the processors being scheduled [8]. (A third class of results from scheduling theory, involving the "tardiness" of jobs in relation to their respective deadlines, appears to be less useful in most computer system scheduling problems.) The most general forms of both these problems are NP-complete ([6], [23]), so much of the literature in this field has involved comparing scheduling heuristics in terms of bounds on computational complexity and "goodness" of the resulting schedules relative to optimal schedules (e.g., [10], [26]).

A number of results suggest the value of using two simple heuristics, shortest processing time first (SPT) and longest processing time first (LPT), to achieve the objectives F_{ave} and F_{max}, respectively. First, we consider cases where all jobs are available at the same time and their processing times are known exactly. In these cases, if all the processors are identical, then SPT exactly minimizes F_{ave} [9] and LPT is guaranteed to produce an F_{max} that is no worse than 4/3 of the minimum possible value [17]. If some processors are uniformly faster than others, then the LPT heuristic is guaranteed

to produce an F_{max} no worse than twice the best possible value [16]. Next, we consider cases where all jobs are available at the same time but their exact processing times are not known in advance. Instead the processing times have certain random distributions (e.g., exponential) with different expected values for different jobs. In these cases, if the system contains identical processors on which preemptions and sharing are allowed, then SPT and LPT exactly minimize the expected values of F_{ave} and F_{max}, respectively, ([45], [15]). Finally, we consider cases where the jobs are not all available at the same time but instead arrive randomly and have exponentially distributed processing times. In these cases, if the processors are identical and allow preemption, then LPT exactly minimizes F_{max} [43].

Other scheduling objectives. DSP can be used to achieve many other possible objectives besides the traditional ones of minimizing mean or maximum flow time for independent jobs. For example:

(1) *Parallel heuristic search.* Many artificial intelligence programs use various kinds of heuristics for determining which of several alternatives in a search space to explore next. For example, in a traditional "best first" heuristic search, the single most promising alternative at each point is always chosen to be explored next [36]. By using the heuristic evaluation function to determine priorities for DSP, a system with n processors available can be always exploring the n most promising alternatives rather than only one. Furthermore, if the processors have different capabilities, each task will be executing on the best processor available to it, given its priority.

(2) *Arbitrary market with priority points.* Another obvious use of DSP is to assign each human user of the system a fixed number of priority points in each time period. Users (or their programs) can then allocate these priority points to tasks in any way they choose in order to obtain the response times they desire (see [41] for a similar--though non-automated--scheme).

(3) *Incentive market with priority points.* If the personal computers on a network are assigned to different people, then a slight modification of the arbitrary market in (2) can be used to give people an incentive to make their personal computers available as contractors. In this modified scheme, people accumulate additional priority points for their own later use, every time their machine acts as a contractor for someone else's task.

Alternative Scheduling Protocols

For comparison purposes, consider the following two alternative protocols. The first protocol is a scheme designed to be the most logical extension of the traditional techniques from scheduling theory (e.g., [9]). (In fact, it is the protocol we initially thought would provide the best performance.) The second is a random assignment method that provides a comparison with designs where no attention is given to the scheduling decision.

Eager assignment

The first alternative protocol overcomes the possible deficiency of DSP that no estimates of completion times are provided by processors that are not ready to start immediately. That is, clients using DSP may start a task on a machine that is available immediately (possibly their own local machine), only to find that another much faster machine becomes available soon. If the task is canceled and restarted, all the processing time on the first machine is wasted. If not, the job finishes later than it could have. If reasonable estimates of completion times on currently busy machines could be made, then clients would know enough to wait for faster machines that were not immediately available. These estimates might also be useful to the human users who, when DSP is used, have no idea of when their tasks will finish until the tasks actually begin execution.

In this alternative, tasks are assigned to contractors as soon as possible after the tasks become available and then reassigned as necessary when conditions change. In this way, each contractor maintains a schedule of tasks it is expected to do, along with their estimated start and finish times, and so the contractor can make estimates of when it could complete any new task that is submitted. By analogy with "lazy" evaluation of variables ([14], [18]) the original DSP could be called "lazy assignment" because clients defer assigning a task to a specific contractor until the contractor is actually ready to start. This alternative protocol, therefore, will be called "eager assignment," since it assigns tasks to contractors as soon as possible.

In this protocol, all contractors bid on all tasks even if they are currently busy. A contractor estimates its starting time for a task by finding the first time in its schedule at which no task of higher priority is scheduled. Then the client picks the best bid and sends the task to the contractor who submitted it. When new tasks are added to a contractor's schedule, or when a task takes longer than expected to complete, the contractor notifies the owners of later tasks in its schedule that their reservations have been "bumped." These clients may then try to reschedule their tasks on other contractors.

It is important to note that even in cases where there is a lot of bumping, this scheduling process is guaranteed to converge. Since tasks can only bump the reservations of other tasks of lower priority, the scheduling of a new task can never cause more than a finite number of bumps. To reduce the finite (but possibly large) amount of rescheduling in rapidly changing situations, rescheduling can occur only when a task is bumped by a large amount.

Random assignment

In the second alternative protocol, clients pick the first contractor who responds to their request for bids and contractors pick the first tasks they receive after an idle period. Contractors do not bid at all when they are executing a task, and they answer all requests for bids when they are idle. If a

client does not receive any bids, it continues to rebroadcast the request for bids periodically. When contractors receive a task after already beginning execution of another one, the new task is rejected (with a "bump" message) and the client who submitted it continues trying to schedule it elsewhere. In the simulations discussed below, the selection of the first bidders when more than one machine is available, and of the first task when more than one task is waiting, are both modeled as random choices since the delay times for message transmission and processing are presumably random. (In reality, fast contractor machines might often respond more quickly to requests for bids than slow ones and so would be more likely to be the first bidders. Thus the performance of this scheduling mechanism in a real system might be somewhat better than the simulated performance.)

Simulation Results: Minimizing Mean Flow Time

Since minimizing the mean flow time of independent jobs is likely to be the primary objective in many real distributed scheduling environments, and since analytic results about this topic are so scarce, it is appropriate to use simulations to compare alternative strategies for achieving this objective. In this section we summarize the results of a series of simulation studies of the three distributed scheduling alternatives outlined above: (1) *lazy assignment* (the original DSP), (2) *eager assignment*, and (3) *random assignment*. In both the eager and lazy alternatives, priorities are determined according to the shortest processing time first (SPT) heuristic. In the random alternative, priorities are not used. These simulation studies are described in more detail by Howard [21].

Method

To simulate the performance of the alternative scheduling strategies, most of the code for the operational Enterprise system was used, with some of the functions (primarily those for sending messages between machines) redefined so that a complete network was simulated on one machine. The completion of jobs was simulated using elapsed time of the simulation clock, with faster machines completing simulated jobs proportionally more quickly than slow ones.

Job loads. For all the simulations, "scripts" of random job submissions were created. All jobs were assumed to be independent of each other and required to be run remotely. The job arrivals were assumed to be a Poisson process and the amount of processing in each job was assumed to be exponentially distributed. This means that, at every instant, there was a constant probability that a new job would arrive in the system, and also that, for each job currently executing, there was a constant probability that, at any instant, it would end. By varying the parameters of the random number generators, we created scripts with average loads of 0.1, 0.5, and 0.9, where average load is defined as the expected amount of processing requested per time interval divided by the total amount of processing power in the system. Thus, 0.1 represents a light load and 0.9 a heavy load.

Estimation errors. In addition to the actual amount of processing in each job, the scripts also included an estimated amount of processing for each job (i.e., the estimate a user might have made of how long the job would take). In order to examine extreme cases, these estimates were either perfect (0 percent error) or very inaccurate (± 100 percent error). In the case of inaccurate estimates, the errors were uniformly randomly distributed over the range.

Machine configurations. Nine different configurations of machines on the network were defined. In all configurations, a total of 8 units of processing power was available, but in different cases this was achieved in different ways: a single machine of speed 8; or 8 machines of speed 1; or 1 machine of speed 4 and 2 machines of speed 2; etc.

Communications. In order to simulate "pure" cases of the different scheduling mechanisms, communication among machines was assumed to be perfectly reliable and instantaneous. In real situations where communication delays are negligible relative to job processing times, this assumption of instantaneous communications is appropriate. In other cases, different assumptions about communications delays might change the trade-offs among scheduling mechanisms.

Bumping, restarting after late bids, and rebroadcasting. In keeping with the spirit of simulating "pure" scheduling methods, jobs in the eager simulations are rescheduled every time their scheduled start time is delayed at all. In a real system, jobs would ordinarily have to be bumped by more than some tolerance before being rescheduled. (After a job begins execution, it is not subject to being bumped.) In the lazy simulations, jobs are never restarted when late bids are received from fast contractors. In other words, the performance of the eager method could only get worse if fewer bumps were made, but the performance of the lazy method might improve if jobs were sometimes restarted after receiving late bids.

Similarly, in the random assignment simulations, clients rebroadcast their requests for bids in every time interval of the simulation until the job is successfully assigned to a contractor. Thus, this simulates the best scheduling performance the random method could achieve; if rebroadcasting occurred less often, the performance could only get worse.

Replications. For each load average, five different random scripts of job submissions were generated. Then these same five scripts were used for each of the three scheduling methods, each of the two accuracies (0 and ± 100 percent), and each of the nine machine configurations. By using the same scripts for all the different methods, accuracies, and configurations, we obtained a much more powerful comparison of the differences due to the factors in which we were interested than if the job submissions had been generated randomly in each different case.

Results

The results presented below are averages of mean flow time for jobs over five scripts and over several machine configurations in each case. There are three configurations of multiple identical machines, five configurations of multiple non-identical machines, and one configuration of a single machine. Strictly speaking, since the different configurations are not representative of any particular population and since there are large differences between different configurations in the same category, one should be hesitant about averaging them in this way. Therefore, we also normalized the flow times for the lazy and eager methods in each configuration by dividing by the flow time of the random method. The averages of these normalized values showed exactly the same relative patterns as the averages of the original flow times, so the original flow times are shown here.

Effect of scheduling method. Somewhat surprisingly, Figure 3 shows that the "lazy" assignment method is at least as good as, and in some cases, much better than the more complicated and expensive "eager" assignment method.

Effect of system load. With perfect estimates of processing amounts, both eager assignment and lazy assignment are consistently better than random assignment. With heavy loads, these differences are substantial (5%-50%). Even with light loads in the case of non-identical processors, there is approximately a 40% advantage for either of the two non-random assignment methods. This sizable advantage under light loads appears to be because the two non-random assignment methods consistently assign jobs to the faster machines while the random method does not. At light loads, a fast machine is almost always available so this is a significant advantage. At moderate loads, fast machines are often busy so this assignment preference does not matter nearly as much.

Effect of machine configuration. There are three important effects of machine configuration: (1) As just discussed, there is a sizable advantage of the non-random scheduling methods at light loads with non-identical machines. (2) As Figure 4 shows, the benefits of non-random scheduling (averaged over all loads) increase as the range of processor speeds in the network increases. (3) There is a clear advantage at all load levels of having one fast machine rather than a combination of slower machines with the same total processing power. This last result can also be derived analytically (see [33]).

Effect of accuracy of processing time estimates. The lazy assignment method is quite robust in the face of poor processing time estimates, with its overall performance degrading only a few percent even when the estimates are off by as much as 100 percent. The eager assignment method, on the other hand, often does much worse with bad estimates than with good ones, and in some cases it even does worse with bad estimates than purely random assignment.

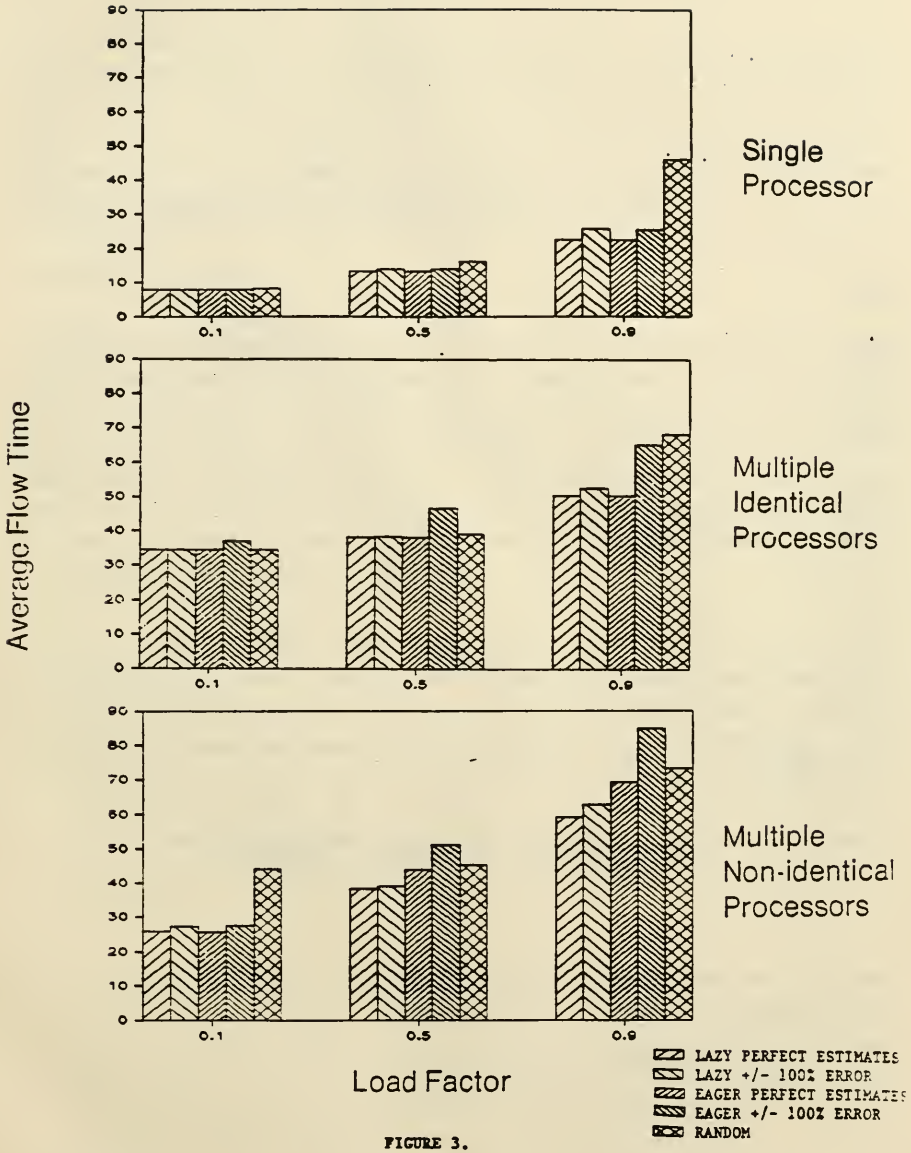

FIGURE 3.

Performance of 3 scheduling methods (lazy assignment, eager assignment, random assignment) under various loads and accuracies of processing time estimates.

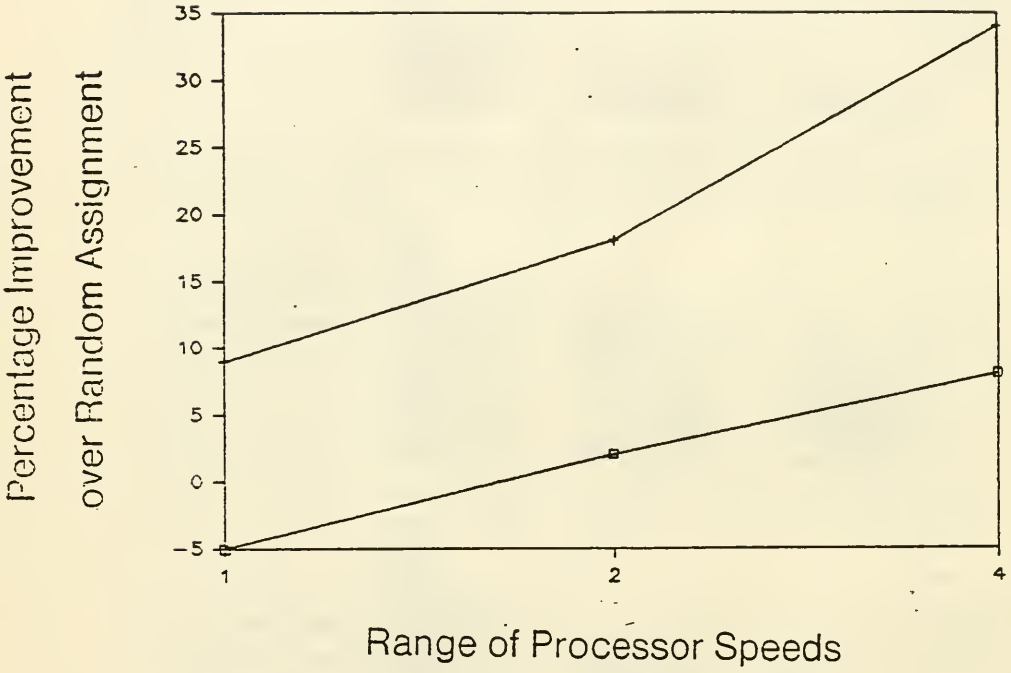

FIGURE 4.

Improvement in average flow time of non-random assignment methods compared to random assignment. "Range of processor speeds" is defined as ratio of speed of fastest to slowest contractor in network.

Amount of message traffic. Figure 5 shows the number of directed and broadcast messages used by the two non-random scheduling methods with perfect processing time estimates for jobs. While the lazy method requires almost the same number of messages per job at both heavy and light loads, the eager method requires many more messages as loads increase. Though they are not shown here, similar results were obtained for poor estimates of job processing times (± 100% error). In these "pure" simulations, communication was treated as instantaneous and free in order to maximize scheduling performance. In designing an actual system, one would presumably sacrifice some scheduling performance in order to reduce the number of messages.

Discussion

The similarities of this scheduling problem to that of job shop scheduling might lead one to believe that obvious extensions of traditional scheduling theory algorithms such as the eager assignment method would provide the best performance for scheduling of distributed tasks. Also, early simulation results by Conway, Maxwell, and Miller [9] of a job shop environment showed that schedules based on very inaccurate estimates performed almost as well as schedules based on perfect estimates and much better than random schedules. Why, then, should the eager assignment method do so poorly in our examples when it is given inaccurate processing time estimates? We believe that two primary factors account for this result:

(1) *"Stable world illusion."* In the eager assignment method, each job is assigned to the machine that estimates the soonest completion time. But if jobs of higher priority arrive later and are assigned to the same machine, then they will keep "bumping" the first job back to later and later times. In other words, jobs are assigned to machines on the assumption that no more jobs will arrive (i.e., that the world will remain stable). Even though in the simulation, jobs are rescheduled every time any new jobs arrive that delay their estimated start time, by the time the job is rescheduled, it may already have missed a chance to start on another machine that could have completed it before it will now be completed.

In some of our simulations (not included here), the bids included an extra factor to correct for this effect, that is, bids included an estimate of how long the starting time of the job would be delayed by jobs that had not yet arrived, but could be expected to arrive before the job began execution. (See [28] for the derivation of this correction factor.) Even though the inclusion of this correction factor did improve the performance of the eager assignment method somewhat, the changes were not substantial.

(2) *Unexpected availability.* When a job takes longer than expected, or when higher priority jobs arrive at a processor, all the clients who submitted jobs scheduled later on that

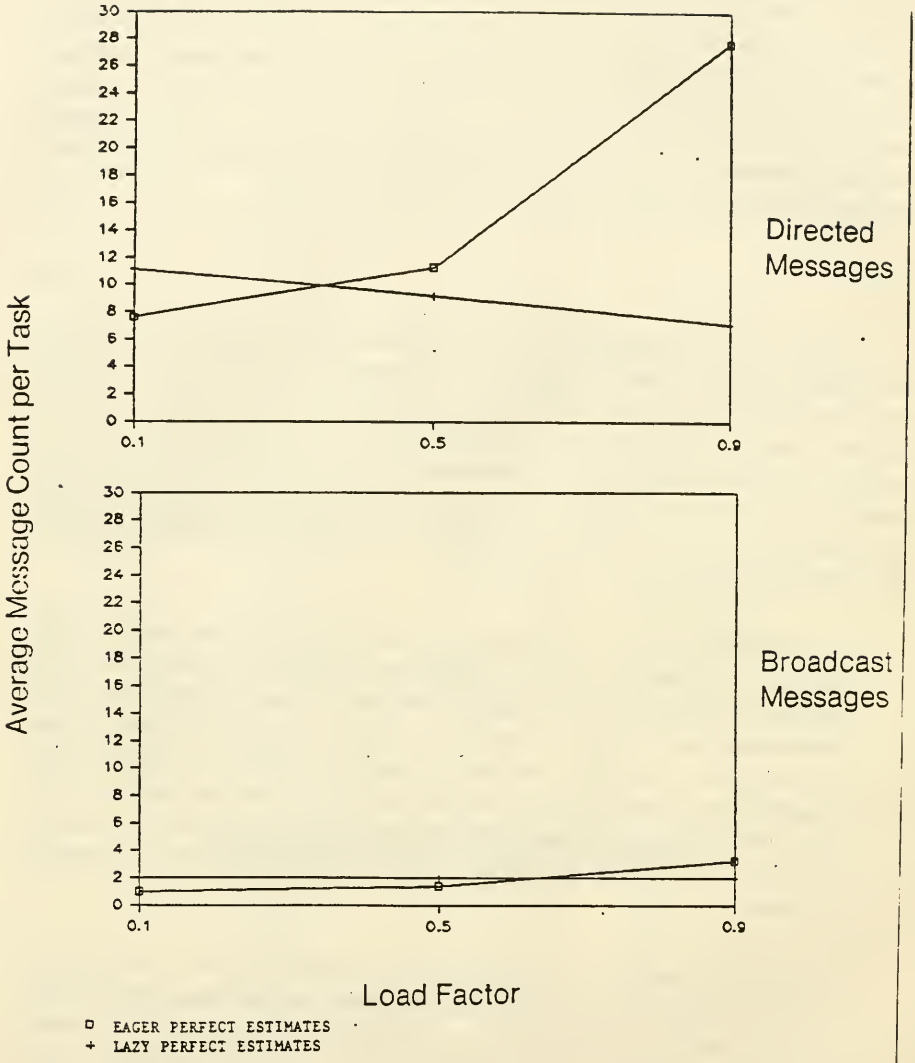

Figure 5. Average number of broadcast and directed messages used by eager and lazy assignment methods to schedule each task under various loads.

processor are notified with "bump" messages and given a chance to reschedule their jobs. When a job takes less time than expected or when jobs scheduled on a processor are canceled, the processor may become available sooner than expected, but in these cases, the clients who submitted jobs that were scheduled elsewhere but who might now want to reschedule on the newly available machine are never notified. There can thus be situations where fast processors are idle while high priority jobs wait in queues on slower processors. This appears to be a serious weakness of the eager assignment method. We have specified, but not implemented, an addition to the protocol that notifies all clients of such situations and allows them to reschedule their tasks. The cost of this addition would be even greater message traffic and system complexity, and we believe it unlikely that the resulting performance would be significantly better than the much simpler lazy assignment method.

Implications. The most important and surprising result of these simulations is the superiority of the lazy assignment method. Even though the lazy method is simpler to implement and often requires substantially less message traffic, it never performs much worse than the alternative methods, and it usually performs much better.

There are two important qualifications on the generality of these results: First, the introduction of communication delays might change the trade-offs among scheduling methods in a few cases. For example, in heavily loaded systems with very long communication delays, the eager assignment method transmits jobs to contractors in advance and thus removes one communication lag from the round trip flow time. Also, these simulations were only attempting to minimize average flow time, not any of the other possible global objectives. Nevertheless, we believe that the superiority of deferring assignment of tasks to machines in multi-machine scheduling environments is a principle that is likely to be very widely applicable, and that is not yet widely known.

DISTRIBUTED SCHEDULING PROTOCOL DEFINITION

In this section, we present a detailed definition of the Distributed Scheduling Protocol (DSP) that is used to implement the lazy assignment method. This protocol can be used by cooperating machines in a network even when the machines use different programming languages and operating systems. This section (and the Interlisp implementation of the protocols) describe all the message formats as lists of fields with no field widths or data types specified. To use the protocol with other languages, field widths and data types should be specified as well.

The message definitions in Figures 6 and 7 use a modified form of BNF specification. Nonterminal symbols are enclosed by angle brackets ("< >"), terminal symbols are written without delimiters, ellipses (". . .") are used to indicate lists containing an arbitrary number of 0 or more terms, and square brackets ("[ ]") indicate comments.

Figure 6
IPC Message Format

<IPCmessage> → (<sender> <receiver> <body> IPCControlInfo)

<sender> → <processID>

<receiver> → <processID>

<processID> → (hostName processName)

Figure 7
DSP Message Formats

<DSPmessage> → <requestforbids> | <bid> | <acknowledgement> | <task> | <result> | <query> | <status> | <bump> | <cancel> | <cutoff>

<requestforbids> → (REQUESTFORBIDS <taskID> priority <earliestStartTime> <requirements> <taskSummary>)

<bid> → (BID <taskID> <startTime> <completionTime>)

<acknowledgement> → (ACK <taskID>)

<task> → (TASK <taskID> priority <taskSummary> taskDescription)

<result> → (RESULT <taskID> <resultStatus> <runtime> result)

<query> → (QUERY <taskID>)

<status> → (STATUS <taskstatus> <startTime> <completionTime>)

<bump> → (BUMP <startTime> <completionTime>)

<cancel> → (CANCEL <taskID>)

<cutoff> → (CUTOFF <taskID> <runtime>)

<taskID> → (hostName taskName taskCreationTime lastMilestoneTime)

<taskCreationTime> → systemTime

<lastMilestoneTime> → systemTime

<earliestStartTime> → systemTime

<startTime> → systemTime

<completionTime> → systemTime

<resultStatus> → NORMAL | ERROR

<runtime> → (elapsedTime machineType)

<taskstatus> → LOCAL | BIDDING | SCHEDULED | DELIVERED | RUNNING | NORMAL | ERROR | DELETED | CUTOFF

<requirements> → (<requirement> . . . <requirment>)

<requirement> → REMOTE | (HOSTS <hostName>. . .<hostName>) | [other terms to be added]

<taskSummary> → (<summaryterm> . . . <summaryterm>)

<summaryterm> → (TIME timeEst) | (FILES <fileDescriptionList>) | [other terms to be added]

<fileDescriptionList> → <fileDescription> | <fileDescription> <fileDescriptionList>

<fileDescription> → (fileName fileCreationDate fileLoadTimeEst)

Inter-Process Communication Protocol

DSP is based on an Inter-Process Communication protocol (IPC) for the transport of messages. Messages in the IPC are assumed to be reliably delivered without duplication. If the IPC guarantees that messages are received in the order they are sent, then some redundant processing will be avoided, but the DSP does not require this guarantee. As specified in Figure 6, messages are assumed to contain at least the following information: *sender*, *receiver*, and *body*. The *sender* and *receiver* are globally unique processIDs for the sending and receiving processes. The body may be a DSP message, as defined below, or some other message. The messages may also contain other IPC control information such as a message identifier, time sent, and information about acknowledgements and duplicates.

DSP Message Formats

Figure 7 specifies the formats for the different DSP messages. The exact formats for the *taskDescription*, *taskSummary*, *priority*, *requirements*, and *result* fields are implementation dependent, with the only restriction being that contractors that recognize and satisfy the *requirements*, must also be able to recognize and deal with the specified *priority* and the descriptions in the *taskSummary* and the *taskDescription*. Both the *requirements* and the *taskSummary* are intended to have new terms added as the protocol is used for different types of machines and different types of tasks.

*Task1D*s are task identifiers that are guaranteed to be unique across time and space. Since a given task can be restarted during its lifetime (e.g., because of a processor failure), it is also necessary to distinguish between these different "incarnations" of the same process [35]. In order to do this, a timestamp of the most recent "milestone event" in the life of the process is included in the *task1D*. Milestone events are the sending of either a request for bids or a task message concerning the task. Both these events render obsolete all previous DSP messages concerning the task. Before responding to DSP messages about a particular task, therefore, both clients and contractors check to be sure the message concerns the most recent incarnation of the task. (*Task1D*s serve the same purpose as the call identifiers used by Birrell and Nelson [2]). In Enterprise, both *process1D*s and *task1D*s use the host's network address as the *hostName*.

The *systemTime* is assumed to be the same (within small limits) across machines. *ElapsedTime* is measured in the same units as *systemTime*, and *machineType* is an implementation dependent encoding of the different types of machines on the network. Bidders who are ready to start on a task immediately indicate that fact by using the *earliestStartTime* specified in the request for bids as the *startTime* in their bid, even if that time has already passed. Contractors indicate that they cannot accept a task at all by returning a bump message whose *startTime* is 0 or NIL. The *fileDescriptions* include a *fileCreationDate* so contractors can determine if the version of a file they have loaded is the correct one.

The general constraints on message processing in DSP are described above in the overview of the scheduling process. Contractors are assumed to bid on or acknowledge only tasks whose *requirements* they satisfy. A contractor can only process one remote task at a time, but once a task has begun execution, it can create any number of other local (or remote) tasks. Different implementations of DSP can use different policies for priority setting, estimating finish times for bids, canceling and restarting after late bids, evaluating a client's own bid, and cutting off excessively long tasks. The only restrictions on these policies stem from the fact that consistent biases (e.g., "lying") on the part of some clients or contractors in setting priorities or submitting bids may lead to radically suboptimal schedules. Choices about restarting and cutoff policies also involve tradeoffs between the amount of computation and communication devoted to scheduling and the efficiency of the schedules.

Scheduling in Enterprise

The initial version of the Enterprise system makes the following choices in implementing the DSP:

(1) *Task descriptions* and *results* are character strings consisting of arbitrary Interlisp forms "dereferenced" to the "print names" of the forms.

(2) *Task summaries* consist of (a) the estimated processing time for the task on a "standard" processor (in our case a Dolphin processor), and (b) the names and lengths of the files that must be loaded before the processing can begin. The estimated processing times for a task are supplied by the programmer, or if no estimate is supplied, a default value is used.

(3) *Priorities* are simply the estimated processing time (including the estimated time to load all files) on a standard processor, with low numbers signifying high priority.

(4) *Requirements* can include (a) "REMOTE" (as opposed to the default which is "REMOTEORLOCAL") and/or (b) a list of acceptable contractors. Tasks that are required to be local never use the DSP.

(5) *Bids* take into account the processor speed of the contractor submitting the bid (relative to a "standard" processor) and the file loading time for all required files not already loaded on the contractor's machine. File loading time is estimated as being proportional to file length.

(6) "Late" bids from bidders who were ready to start as soon as they received the request for bids are accepted (and the task is canceled on the early bidder's machine) if they are better than the earlier bid by an amount greater than BidTolerance. All other late bids are rejected.

(7) A client machine's bid on its own task is processed after a delay of OwnBidDelay. As in interim measure, its own processing time is inflated by a factor equal to the number of other processes active on the client machine.

(8) A task (and any subtasks it has created) is *cutoff* if it exceeds its estimated time by a factor greater than CutoffFactor.

"Gaming" the system. If people supply their own estimates of processing times for their tasks and these time estimates are also used to determine priority, there is a clear incentive for people to bias their processing time estimates in order to get higher priority. This incentive to give biased estimates is counteracted in the current system by the possibility of a job being cutoff if it greatly exceeds its estimated time. In general, this issue of "incentive compatibility" [22] is an important one in designing any organization that involves human actors.

Conclusion

We believe that this paper has made three primary contributions:

First, any designer of a parallel processing computing system, whether the processors are geographically distributed or not, must solve the problem of scheduling tasks on processors. We presented a simple heuristic method for solving this problem, and demonstrated with simulation studies its superiority to two plausible alternatives. The simulation studies highlighted the benefit of deferring as long as possible the actual assignment of tasks to processors.

The scheduling heuristic we presented has the additional advantage of lending itself very naturally to a decentralized implementation in which separate decisions made by a set of geographically distributed processors lead to a globally coherent schedule. To aid future implementers of such distributed systems, we formalized a language-independent protocol (the Distributed Scheduling Protocol) for coordinating decentralized scheduling decisions.

Finally, whether scheduling decisions are centralized or decentralized, the Enterprise system points the way toward a new generation of distributed computing environments in which programmers can easily take advantage of the maximum amount of processing power and parallelism available on a network at any time, with little extra cost when there are few extra machines available.

Acknowledgements

The authors would like to thank Michael Cohen, Randy Davis, Larry Masinter, Mike Rothkopf, Henry Thompson, and Bill van Melle for helpful discussions about the design of the Enterprise system.

The final stages of this work were supported in part by the Center for Information Systems Research, MIT.

References

[1] Berhard, R. Computing at the speed limit. *IEEE Spectrum*, July 1982, 26-31.

[2] Birrell, A. D., and Nelson, B. J. *Implementing remote procedure calls.* Technical report no. CSL-83-7, Xerox Palo Alto Research Center, Palo Alto, Calif., October 1983.

[3] Birrell, Andrew D., Levin, Roy, Needham, Roger M., Schroeder, Michael D., Grapevine: An Excercise in Distributed Computing. *Communications of the ACM* 25(4), April 1982.

[4] Boggs, David R., Shoch, John F., Taft, Edward A., Metcalfe, Robert M., Pup: An Internetwork Architecture *IEEE Transactions on Communications*, Volume COM-28 Number 4, April 1980.

[5] Brinch Hansen, P., *Operating Systems Principles* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1973.

[6] Bruno, J., Coffman, E. G., & Sethi, R. Scheduling independent tasks to reduce mean finishing time. *Communications of the ACM*, 1974, *17*, 382-387.

[7] Chandrasekaran, B., Natural and Social System Metaphors for Distributed Problem Solving: Introduction to the Issue. *IEEE Transactions on Systems, Man, and Cybernetics.* 1981 (January). *SMC-11*, 1-4.

[8] Coffman, Edward G., Jr., and Denning, Peter J., *Operating Systems Theory* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1973.

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

[10] Davis, E. and Jaffe, J. M. Algorithms for scheduling tasks on unrelated processors. *Journal of the ACM*, 1981 (October), *28*, 721-736.

[11] Davis, R., and Smith, R. G., *Negotiation as a Metaphor for Distributed Problem Solving* Artificial Intelligence Volume 20 Number 1, January 1983.

[12] Farber, D. J. and Larson, K. C. The structure of the distributed computing system--Software. In J. Fox (Ed.). *Proceedings of the Symposium on Computer-Communications Networks and Teletraffic*, Brooklyn, NY: Polytechnic Press, 1972, pp. 539-545.

[13] Fikes, Richard E., A Commitment-Based Framework for Describing Informal Cooperative Work, *Cognitive Science*, 1982, 6, 331-347.

[14] Friedman, D. & Wise, D. CONS should not evaluate its arguments. *Automata, Languages and Programming*. Edinburgh University Press, 1976, 257-284.

[15] Glazebrook, K. D. Scheduling tasks with exponential service times on parallel processors. *Journal of Applied Probability*, 1979, *16*, 685-689.

[16] Gonzales, T., Ibarra, O. H., and Sahni, S. Bounds for LPT schedules on uniform processors. *SIAM Journal of Computing*, 1977, *6*, 155-166 (as cited by [Jaffee]).

[17] Graham, R. L. Bounds on multiprocessing timing anomalies. *SIAM Journal of Applied Mathematics*, 1969 (March), *17*, 416-429 (summarized in Coffman and Denning, pp. 100-106.).

[18] Henderson, P. & J. Morris, Jr. A lazy evaluator. *Record Third Symposium on Principles of Programming Languages*, 1976, 95-103.

[19] Hewitt, C. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 1977, *8*, 323-364.

[20] Hoare, C. A. R. Communicating sequential processes. *Communications of the ACM*, 1978, *21*, 666-677.

[21] Howard, M. T. Tradeoffs in distributed scheduling. Unpublished M.S. thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, forthcoming.

[22] Hurwicz, L. The design of resource allocation mechanisms. *American Economic Review Papers and Proceedings*, 58 (May, 1973), 1-30 (Reprinted in K. J. Arrow and L. Hurwicz (Eds.) *Studies in Resource Allocation Processes*. Cambridge: Cambridge University Press, 1977, pp. 3-40).

[23] Ibarra, O. H. and Kim, C. E. Heuristic algorithms for scheduling independent tasks on nonidentical processors. *Journal of the ACM*, 1977 (April), *24*, 280-289.

[24] Imai, M., Yoshida, Y., and Fukumura, T. A parallel searching scheme for multiprocessor systems and its application to combinatorial problems. *Proceedings of the Sixth International Joint Conference on Artificial Intelligence*, Tokyo, Japan, August 20-23, 1979.

[25] Ingalls, D. H. The Smalltalk-76 programming system: Design and implementation. *Conference record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, Tucson, AZ, January 1978, pp. 9-16.

[26] Jaffe, J. M. Efficient scheduling of tasks without full use of processor resources. *Theoretical*

Computer Science, 1980. *12*, 1-17.

[27] Jones, Anita K., and Schwarz, Peter. *Experience Using Multiprocessor Systems - A Status Report* Computing Surveys, Volume 12 Number 2, June 1980.

[28] Kornfeld, W. A. Combinatorially implosive algorithms. *Communications of the ACM*, 1982 (October). *25*, 734-738.

[29] Kriebel, C. H., & Mikhail, O. I. Dynamic pricing of resources in computer networks. In C. H. Kriebel, R. L. Van Horn, & J. T. Heames (Eds.). *Management Information Systems: Progress and Perspectives*. Pittsburgh: Carnegie Press, 1971. pp. 105-124.

[30] Lampson, B. W. A scheduling philosophy for multiprocessing systems. *Communications of the ACM*, 1968 (May). *11*, 347-359.

[31] Malone, Thomas W. *Organizing Information Processing Systems: Parallels between Human Organizations and Computer Systems* Working Paper, Xerox Palo Alto Research Center, Cognitive and Instructional Sciences Group, August 1982.

[32] Malone, T. W. and Rothkopf, M. H. Strategies for scheduling parallel processing computer systems. Paper in preparation. Sloan School of Management, MIT.

[33] Malone, T. W. and Smith, S. A. Trade-offs in designing organizations. Paper in preparation, Sloan School of Management, MIT.

[34] Metcalfe, R. M., and Boggs, D. R., *Ethernet: distributed packet switching for local computer networks* Communications of the ACM 19 (7), July 1976.

[35] Nelson, Bruce Jay, *Remote Procedure Call* Xerox Palo Alto Research Center, CSL-81-9, May 1981.

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

[37] Shoch, John F., Hupp, Jon A., *The WORM Programs - Early Experience with a Distributed Computation* Communications of the ACM 25(3), March 1982.

[38] Smith, R. G., *The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver* IEEE Transactions on Computers Volume C-29 Number 12, December 1980.

[39] Smith, R. G. and Davis, R., *Frameworks for Cooperation in Distributed Problem Solving* IEEE Transactions on Systems, Man, and Cybernetics Volume SMC-11 Number 1, January 1981.

[40] Sproull, R. F., and Cohen, D. High-level protocols. *Proceedings of the IEEE*. 66 (11): 1371-86. November, 1978.

[41] Sutherland, I. E. A futures market in computer time. *Communications of the ACM*. 1968 (June). *11*, 449-451.

[42] Thompson, H. *Remote Procedure Call*. Unpublished documentation for Interlisp-D system. Xerox Palo Alto Research Center. Palo Alto, CA: January, 1983.

[43] Van der Heyden, L. Scheduling jobs with exponential processing and arrival times on identical processors so as to minimize the expected makespan. *Mathematics of Operations Research*. 1981, *6*, 305-312.

[44] Van Tilborg, A. M., & Wittie, L. D. Distributed task force scheduling in multi-microcomputer networks. *Proceedings of the National Computer Conference*. 1981, *50*, 283-289.

[45] Weber, R. R. Scheduling jobs with stochastic processing requirements on parallel machines to minimize makespan or flowtime. *Journal of Applied Probability*, 1982, *19*, 167-182.

[46] White, J. E.; A high-level framework for network-based resource sharing. *AFIPS Proceedings of the National Computer Conference*, June 1976, pp. 561-570.

[47] Weide, B. W. *Statistical methods in algorithm design and analysis*. Ph.D. dissertation, Carnegie-Mellon University, Pittsburgh, PA, August, 1978 (as cited in Jones, A. K. & Schwarz, P. Experience using multiprocessor systems--A status report. *Computing Surveys*, 1980 (June), *12*, 121-165).

[48] Wesson, R., Hayes-Roth, F., Burge, J. W., Stasz, C., and Sunshine, C. A. Network structures for distributed situation assessment. *IEEE Transactions on Systems, Man, and Cybernetics*. 1981 (January), *SMC-11*, 5-23.

BARCODE ON BACK COVER