A Microeconomic Scheduler for Parallel Computers

Ion Stoica, Hussein Abdel-Wahab, Alex Pothen *

Department of Computer Science,
Old Dominion University, Norfolk VA 23529-0162, USA
e-mail: {stoica, wahab, pothen}@cs.odu.edu

Abstract. We describe a scheduler based on the microeconomic paradigm for scheduling on-line a set of parallel jobs in a multiprocessor system. In addition to increasing the system throughput and reducing the response time, we consider *fairness* in allocating system resources among the *users*, and provide the user with *control* over the relative performances of his jobs. Every user has a *savings account* in which he receives money at a constant rate. To run a job, the user creates an *expense account* for that job to which he transfers money from his savings account. The job uses the funds in its expense account to obtain the system resources it needs. The share of the system resources allocated to the user is directly related to the rate at which the user receives money; the rate at which the user transfers money into a job expense account controls the job's performance.
We prove that starvation is not possible in our model. Simulation results show that our scheduler improves both system and user performances in comparison with two different variable partitioning policies. It is also effective in guaranteeing fairness and providing control over the performance of jobs.

1 Introduction

We describe a microeconomic approach for scheduling on-line a set of jobs in a parallel system with identical processors. This approach exploits the following similarity between the scheduling and the resource allocation problems in a computer system, and in a real economic system: Each system involves independent agents that compete for common resources in pursuing their goals. We adopt an open-market strategy which has proved to be successful in dealing with the enormous complexity of real economical environments.

The microeconomic approach has several advantages over other algorithms that have been developed for this scheduling problem [5]. The usual formulations of this problem seek to maximize the system throughput and minimize the user response time, but, in practice there are additional requirements that schedules

* Also affiliated with ICASE, MS 132C, NASA Langley Research Center, Hampton VA 23681-0001, USA. This author was supported by NSF grant CCR-9024954, by U. S. DOE grant DE-FG05-94ER25216, and by NASA Contract NAS1-19480.

must satisfy. The first of these is to ensure *fairness* in resource allocation among the users. A second requirement is to give the user flexibility in *controlling* the relative share of resources allocated among his jobs. We show that both these features can be incorporated into the microeconomic approach in a very natural way, while this is not true of many of the earlier scheduling algorithms.

Scheduling problems are usually formulated as optimization problems of minimizing the maximum completion time or the maximum lateness [1, 8, 9, 17]. Since even simplified formulations of scheduling problems are NP-hard in general [1, 8, 17], many sub-optimal algorithms have been proposed [3, 9]. The more complex scheduling problem considered here is also NP-hard, and the microeconomic approach leads to a heuristic algorithm for the problem. We show by simulation that this algorithm improves both system and user performances relative to two different variable partitioning policies [5].

The microeconomic paradigm has been applied to the resource allocation problem by Miller and Malone from MIT, Drexler and Huberman from Xerox, and others [4, 14, 15] at the end of the eighties. In the last few years, several schedulers based on this paradigm have been proposed [4, 19]. These schedulers use the auction mechanism to allocate resources among competing users. At the beginning of every time-slice, the resource initiates an auction in which the interested users participate by bidding monetary funds that increase over time. The client that offers the highest bid acquires the resource for the next time-slice. The price per time-slice is directly related to the level of competition for that resource; if the competition increases, the price also increases. In this way, as in real *economic* environments, the users are encouraged to maximize their profit, i.e., to devote their funds to resources that are more important for them. These schedulers were intended more for distributed systems in which resources are allocated in an un-correlated manner. Therefore, these systems were suited more for coarse grained asynchronous parallel applications, such as Monte-Carlo simulations [19]. In contrast, the majority of parallel scientific applications are highly synchronous, in that an application requires a specified number of processors to be available during the same interval of time. Another problem with these schedulers is that holding an auction at the beginning of every time-slice incurs a high overhead.

A microeconomic algorithm for balancing the load in distributed systems was suggested by Ferguson *et al.* [7]. Jobs are assumed to arrive independently at every processor in the system. Upon arrival, each job evaluates the cost to run locally or to migrate and execute on another processor. If a job migrates, it has to pay for the communication bandwidth required. Their experiments show that the algorithm is effective in allocating processors and communication resources.

A market-based approach was proposed by Cheriton and Harty [2] for system memory allocation. In their system, the memory manager deposits money in a process account, proportional to the share of the resources that process has to receive. Unlike a real market, the resource prices are assumed to be fixed. When it has enough money in its account, the process "leases" the required amount of memory for a bounded interval of time. At the application level, this

approach proved to be effective in controlling the amount and the interval of time for which the memory is allocated, on uniprocessor and shared-memory multiprocessor systems.

The remainder of the paper is organized as follows. In the next section we present the model in detail. In Section 3, we prove that the starvation is not possible in our model. Section 4 describes the simulation results. Finally, in Section 5 we summarize our results and indicate some future directions for extending our work.

2 The Model

We consider a parallel computer consisting of N identical processors interconnected by a general communication network. We assume that the communication parameters for any pair of processors do not depend on their relative position, [2] and therefore the system may be arbitrarily partitioned. Every job specifies, upon its arrival, the number of processors p it needs, and the estimated computation time. Once processors are allocated, they are guaranteed to be exclusively used by the job for the entire duration of its execution. Also, the job is assumed to acquire or release all p processors at the same time.

The computation system is modeled as a microeconomic environment in which different *users* compete for obtaining system *resources* in order to run their *jobs*. To get the requested resources the user has to pay the price asked by the system. As in real life, the buyers (users) and the sellers (system) have antagonistic goals; the users wish to run their jobs as fast as possible with minimum expenses, while the system wants to maximize its income.

The flow of currency in the system is depicted in Figure 1. Every user has a *savings account* in which he receives money at a constant rate, as long as he has less than a specified amount of funds. Whenever a user decides to run a job, he creates an *expense account* for that job to which money from his savings account is transferred. The job uses this account to buy the resources it needs. Once the job is scheduled for execution, all of its money (and depending on the strategy, possibly all the money it receives until it terminates) is transferred to the *system account*. In order to maximize the system income, the scheduler applies a simple strategy: it allocates available resources to the job that offers the best price. In a loaded system, it is possible that not all p processors that were requested by a job become available at the same time. In this case, when the job is scheduled it is asked to pay for the wasted resources also. In this way resource fragmentation is discouraged.

For convenience, throughout this paper we refer to the monetary-unit as a *dollar* and to the time-unit as a *minute*. The notations used in this paper are summarized in Table 1.

[2] This is a reasonable assumption for many modern multiprocessor architectures (e.g., IBM SP-1/2, Intel Paragon).

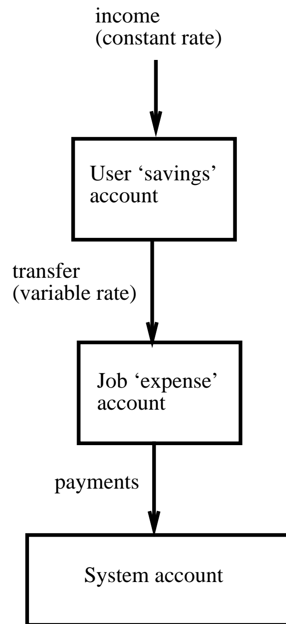


Fig. 1. The currency flow.

2.1 The User Savings Account

Every user has a *savings account* in which he accumulates funds for buying resources required by his jobs. The maximum amount of money the user i can deposit in his savings account is bounded by M_i. While the user has less than M_i dollars, he receives money at a constant rate R_i. Intuitively, this can be visualized as a system in which every user has a tank with capacity M_i where he saves his earnings for future consumption. While the tank is not full, the inlet-valve is open and the tank is filled at a constant rate R_i; once the tank is full, the inlet-valve is closed.

Limiting the maximum funds in a user's savings account is necessary to avoid disruption in system utilization. Suppose a user does not use the computer for a long period of time (e.g., during his holiday). Without this limitation it is possible for the user to acquire enough money to monopolize the system for an appreciable interval of time (e.g., several hours) when he returns, which will preclude other users from running their jobs.

During a time interval Δt, user i receives at most $R_i \Delta t$ dollars and spends at most $M_i + R_i \Delta t$ dollars (provided he has M_i dollars at the beginning of the time interval, and spends all his savings and earnings during the interval Δt). Notice that for a sufficiently large interval of time ($\Delta t \rightarrow \infty$) the amount $R_i \Delta t$ is the dominant term in the money spent by user i. Thus, over large intervals of time, R_i dictates how much money the user i can spend on the average for

M_i	The maximum amount of funds user i can have in his savings account.
R_i	The rate at which user i receives income, when he has less than M_i dollars in his savings account.
$m_i(t)$	The amount of money user i has in his savings account at time t.
R	The income rate over all users in the system; $R = \sum_{i=1}^{m} R_i$.
J_{ik}	The kth job started by user i.
$r_{ik}(t)$	The rate at which user i transfers money into job J_{ik}'s expense account at time t.
$m_{ik}(t)$	The amount of money job J_{ik} has in its expense account at time t.
N_{ik}	The number of processors requested by J_{ik}.
T_{ik}	The estimated service time required by job J_{ik} when N_{ik} processors are used.
E_{ik}	The estimated cumulative computation time for J_{ik}, i.e., $E_{ik} = N_{ik}T_{ik}$.

Table 1. The notations used in this paper.

acquiring system resources.

Next, notice that R_i is directly related to the share of the system resources that user i receives; the higher R_i is, the more resources the user can buy. Moreover, if two users compete for resources at the same time, they will get a share that is roughly proportional to their expenditures (since the resource prices will be the same). If both users spend at the same rate as they receive money, then the ratio of their share of the resources would be roughly proportional to their income rates. This discussion of "fairness" assumes that users compete at the same time. Otherwise, it is possible for a user with less money to buy more resources than another user with more money. Consider the user who runs his jobs at night, when the system is lightly loaded and resource prices are low, rather than during the day when the system is heavily loaded.

Although the income rate R_i determines the maximum spending rate over large intervals of time, a user with a lower income rate should be able to execute urgent tasks when needed. This is possible in our model since for short intervals of time a user can spend much more than his income. Specifically, let m_i ($m_i \leq M_i$) be the amount of funds user i has in his savings account at the beginning of the time interval dt. Then, the user can spend $m_i + R_i dt$ dollars during the interval dt, and therefore the average spending rate $(m_i/dt) + R_i$ could be much higher than R_i.

2.2 The Job Expense Account

When a user wants to run a job he has to specify its estimated running time and the number of processors needed. At the same time, for every job he wishes to run, the user creates an *expense account* to which he begins to transfer funds from his savings account. In contrast with the user's income rate which is constant, the rate at which money is transferred into the savings account of a job is variable,

and is specified by the user. These funds are used to buy the resources required by the job. In this way, the user has the flexibility to adjust his expenses according to the number and relative importance of his jobs. This is similar to the real-life situation in which people receive a fixed salary per month but have the freedom to spend their money according to their needs.

When a user submits a job to be executed, an expense account is created for it, and the job is inserted into a list called the *ready-list*. Whenever a set of processors becomes idle, the scheduler scans the *ready-list* and selects the job that offers the best price (see the next section for details). If there are enough idle processors available, then the selected job could be executed immediately. Two approaches are possible for the manner in which the scheduler computes the funds that a job could afford to spend for acquiring the resources at a time t (denoted by $m'_{ik}(t)$). In one, it considers only the current funds in the expense account of the job; in the other, it considers the future earnings of the job also. More specifically, let J_{ik} be a job in the *ready-list*, belonging to the user i, that at time t has $m_{ik}(t)$ dollars in its expense account and receives money at a rate $r_{ik}(t)$. Then, in the first approach, the scheduler evaluates J_{ik} to have $m_{ik}(t)$ dollars. In the second approach, the scheduler finds that the job can spend

$$m'_{ik}(t) = m_{ik}(t) + \int_{t}^{t_f} r_{ik}(t')dt', \qquad (1)$$

where t_f is J_{ik}'s estimated finishing time (t_f is the current time t plus the estimated waiting time plus the estimated running time). Equation (1) has only a theoretical importance, since in practice it is hard to estimate how r_{ik} will vary in the future. This depends both on the user's strategy and the current set of jobs he has to run. A guaranteed lower bound r'_{ik} on the rate at which J_{ik} will receive money in its expense account could be used to obtain a simple estimate of the future income. Then, at time t, the scheduler can assume that J_{ik} can spend at least $m'_{ik}(t) = m_{ik}(t) + r'_{ik}(t_f - t)$ dollars. Notice that if $r'_{ik} = 0$ (user i does not guarantee any future money transfer for the job), then this reduces to the first approach since $m'_{ik}(t) = m_{ik}(t)$.

For simplicity, the transfer of money between a user's savings account and the job expense account is unidirectional in that money cannot be transferred back into the savings account. For example, if a job buys some resources for a certain interval of time but finishes earlier than predicted, then the balance cannot be returned back to the user. On the other hand, if a job fails to finish at the predicted time, then it will be allowed to continue for some time while being charged for the additional time, as long as the user can afford to pay; otherwise it will be terminated. This simple solution motivates the user to provide accurate estimates for the job service time.

2.3 The Price of Computation

We have considered two strategies for establishing the price of computation.

The first approach is similar to the one used in other microeconomic systems [4, 19]. In this approach, time is assumed to be divided into intervals called

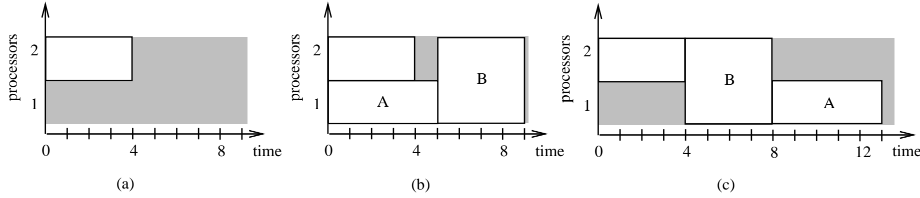


Fig. 2. The execution time diagram for two processors. The shaded area next to a processor indicates that the processor is free while the white area indicates that it is busy. At time 0 (Figure (a) processor 1 is free while processor 2 is busy for 4 minutes. Also, at time 0 there are two jobs A and B in the ready-list; A needs one processor for 5 minutes, while B requires two processors for 4 minutes. Figures (b) and (c) show two possible schedules.

time-slices. At the beginning of every time-slice the scheduler computes the prices offered by all the jobs in the *ready-list*. If the job that uses the resource has not finished yet, then it is allowed to execute for the current time-slice if it can continue to pay at the current price; otherwise the job that has offered the highest price is scheduled to run for the current time-slice. Since the price is evaluated at every time-slice, this scheme accurately reflects the market trends in prices (e.g., when competition increases, the price also tends to increase). Unfortunately, this approach has several drawbacks. First, evaluating the highest offer at every time-slice incurs a high overhead. Second, a job would not know at the beginning *how much* it has to pay to complete execution, and could run out of money before termination due to unexpected price changes.

The second strategy is to negotiate a price that is constant for the entire period of the computation. The main disadvantage of this strategy is that for large intervals of time the price may no longer reflect the level of competition for the resources. For example, if a user starts several jobs early in the morning before other users submit their jobs, he can get all the resources at zero cost since there is no competition. But, if his jobs take several hours to complete, then no other user can run jobs during this time. This would compromise our objective to ensure *fairness*. A common solution to this problem is to gather statistics and predict the price per minute for future process utilization. Since the prediction is more accurate over large intervals of time, we use a weighted function in order to establish the price for the next Δt minutes at time t. More precisely, $p(t)$, the price at time t, is

$$p(t) = p_e(t, \Delta t) + (p_a(t) - p_e(t, \Delta t))e^{-\alpha \Delta t}, \qquad (2)$$

where $p_a(t)$ represents the current highest offer at time t, $p_e(t, \Delta t)$ is the estimated price for the next Δt minutes and α is a positive constant. When $\Delta t \to 0$ the price $p(t)$ goes to $p_a(t)$, while for large values of Δt ($\Delta t \to \infty$) the price $p(t)$ tends to the estimated value $p_e(t, \Delta t)$. Thus, every job that is scheduled to

start at time t and run for the next Δt minutes is asked to pay at least $p(t)$ dollars/minute.

We chose the second approach for two reasons: first, the completion time for computation bounded jobs can be predicted with good accuracy; and second, the algorithm is simpler and more efficient to implement.

When the scheduler scans the *ready-list*, it computes the price per minute offered by every job J_{ik} as a function f of: the predicted service time T_{ik}, the number of requested processors N_{ik}, and the estimated expenses $m'_{ik}(t)$. We describe the details in the next two paragraphs.

First, consider a job J_{ik} that needs only one processor ($N_{ik} = 1$). In this case, the price offered by J_{ik} is computed as $f(1, T_{ik}, m'_{ik}(t)) = \frac{m'_{ik}(t)}{T_{ik}}$. Next, consider a job J_{ik} that requires N_{ik} processors, where $1 < N_{ik} \leq n$. If at least N_{ik} processors become free at the same time then the price offered by J_{ik} is computed as $f(N_{ik}, T_{ik}, m'_{ik}(t)) = \frac{m'_{ik}(t)}{N_{ik}T_{ik}} = \frac{m'_{ik}(t)}{E_{ik}}$. When the first N_{ik} processors to become free finish at different times (as is more probable), deciding what job to run next in order to maximize the system income is difficult. To see why, consider the example shown in Figure 2(a). The system consists of two processors such that when the first processor becomes free, the second one requires 4 minutes to process its current task. Now, assume that there are two jobs: A requires one processor for 5 minutes and offers 3 dollars/minute and B requires two processors for a total of 8 minutes (4 minutes on each processor) and offers to pay 4 dollars/minute. What job must be scheduled first in order to maximize the system income ? The second job offers a higher price per minute but cannot start as long as the second processor is busy, while although the first job offers a lower price, it can start immediately. The following examples show that there is no unique answer. If the next job to be executed requests two processors, then clearly, scheduling A first (Figure 2(b)) is better since both processors are free after 9 minutes. On the other hand, if the next job to be executed arrives at $t = 1$, requires exactly 3 minutes, and pays 6 dollars/minute, then it can be immediately scheduled on processor 1, and therefore scheduling B first maximizes the system income (Figure 2(c)).

Our solution to this problem is the following. In computing the price for a job, the scheduler takes into account not only the effective cumulative computation time (E_{ik}), but also the computation time that is wasted while waiting for other processors (requested by the job) to be available. In the example, when B is scheduled it wastes four minutes of processor 1 unless there is another job in the *ready-list* that can fit in the space. Consequently, the scheduler asks the job to pay also for the potentially wasted four minutes and B is estimated to require 12 minutes ($=$ 4 minutes $\times$ 2 processors $+$ 4 wasted minutes). Hence, the *real price per minute* offered by B is scaled proportionally, i.e., $4 \cdot \frac{8}{12} = 2.66\ldots$. With this modification, the scheduling algorithm will continue to select the job that offers the highest real price per minute (in this example, A). Thus, in this case we compute the price offered by J_{ik} as being

$$f(N_{ik}, T_{ik}, m'_{ik}(t)) = \frac{m'_{ik}(t)}{W_{ik} + N_{ik}T_{ik}} = \frac{m'_{ik}(t)}{W_{ik} + E_{ik}}, \tag{3}$$

where W_{ik} is the wasted computation time in scheduling J_{ik} to run on the first N_{ik} processors that become available. Notice that asking parallel jobs to pay for potentially wasted resources discourages fragmentation in processor allocation.

2.4 The User Strategy

Generally, the user can implement any mechanism for allocating funds to his jobs in our model. Unfortunately, this freedom makes it very hard to analyze and even simulate such a model. Hence we propose a simple strategy that we consider to be flexible enough for practical use. As in other scheduling policies, the idea is to group jobs into different classes. But, while in other policies this classification is done at the central level from the system point of view (e.g., based on the resource requirements), in our case the classification is done at the user level. For example, the user can classify his jobs based on their urgency, their resource requirements, etc.

Let $C_{i1}, C_{i2}, \ldots C_{is}$ be s classes to which the jobs of user i may belong. We associate a coefficient α_{il} with each job class C_{il}, chosen such that $\sum_{l=1}^{s} \alpha_{il} = 1$. Recall that E_{ik} is the cumulative computation time requested by a job J_{ik} and let $\overline{E_i}$ be weighted sum over all the estimated cumulative computation times of all jobs of user i that are in the *ready-list*. Hence we have

$$\overline{E_i} = \sum_{l=1}^{s} \alpha_{il} \Big(\sum_{J_{ik} \in C_{il}} E_{ik} \Big). \tag{4}$$

Then the transfer rate to the expense account of J_{ik} is given by the following formula:

$$r_{ik} = \alpha_{il} R_i \frac{E_{ik}}{\overline{E_i}}. \tag{5}$$

Notice that if user i has at least one job, then the sum of the transfer rates into the expense accounts of his jobs is equal to his income rate R_i.

In this strategy the classification reflects the importance of the jobs; the higher the coefficient α_{il}, the higher the price increase a job belonging to the class C_{il} can afford to pay.

This strategy can be further refined by allowing the coefficients to be dynamically changed in order to achieve certain objective functions (see Section 4 for details).

2.5 Implementation Issues

The overhead introduced by the scheduler is as important as the scheduler performance itself. Therefore, in this section we briefly describe some implementation issues.

The information in the *ready-list* is modified in one of the following cases: a new job arrives in the list, a job terminates and its processors become available, and the rate r_{ik} (at which a job receives money from its user) changes. Since the first case is trivial (the job is appended to the *ready-list*), we discuss only the other two.

When a subset of processors becomes free and there is no other job that is scheduled to be executed, the job in the *ready-list* that offers the highest price is scheduled. If there are enough available processors, the job is executed immediately; otherwise it has to wait until enough processors become free. If a job is already scheduled for execution, then the scheduler checks whether there are enough free processors for that job. If so, the job is loaded, and the scheduler scans the *ready-list* to schedule a new job. The complexity of finding the next job to schedule is linear in the number of jobs in the *ready-list*, since the list is scanned *only* once to schedule a job.

When $r_{ik}(t)$, the rate at which user i transfers money into J_{ik}'s expense account, changes, the amount of money in the account, $m_{ik}(t)$, is updated. For simplicity of exposition, assume that $r_{ik}(t)$ is constant between two subsequent changes (the case when $r_{ik}(t)$ is an arbitrary function can be treated similarly). Suppose that at $t = t_1$, the rate $r_{ik}(t)$ was changed and the expense account was updated accordingly. Then, when $r_{ik}(t)$ is changed again at $t = t_2$, we have $m_{ik}(t_2) = m_{ik}(t_1) + r_{ik}(t_1)(t_2 - t_1)$. Thus the scheduler can compute $m_{ik}(t)$ at any future time $t > t_2$ before the rate changes again. Recall from the previous section that in our scheme the rate $r_{ik}(t)$ is changed *only* when a new job arrives in the *ready-list*, or a job finishes execution. Then the rates are changed for *all* jobs belonging to user i. In the worst case all the jobs in the *ready-list* belong to user i, and hence the complexity of the updates caused by a change in these transfer rates is linear in the number of jobs in the list.

If the *ready-list* is too large for an algorithm that is linear in the number of jobs to be satisfactory, then a variant of the algorithm in which only the first $\overline{n}$ jobs from the list are considered for scheduling can be implemented, where $\overline{n}$ is a parameter that can be specified.

3 Non-Starvation

Every scheduling algorithm has to address the fundamental problem of *starvation*, the situation where a job waits indefinitely to acquire the resources it needs to run. In this section we prove that starvation is not possible in our model. We make two assumptions: first, the running time of every job is bounded above by T_{max}, and second, there exists a lower bound r_{min} on the transfer rate from the

user savings account to every job's expense account. Also, we assume that the number of users m is bounded. [3]

Let us consider a job J that requires p processors for an estimated service time T on a parallel computer with N identical processors. For convenience denote the time at which J enters the *ready-list* by $t = 0$. After Δt minutes, J has in its expense account at least $r_{min}\Delta t$ dollars. In order to be scheduled, a job has to offer the highest price per minute during the sum of the required computation time pT and the time the job spends in waiting for p processors to become free. The largest amount of money that job J has to pay is when there are $p-1$ free processors and the remaining $N-p+1$ processors finish after exactly T_{max} minutes. Therefore, to be scheduled the job J has to pay for at most $pT + (p-1)T_{max}$ minutes. Let Δt_1 be the time interval at which the following equality is true:

$$\frac{r_{min}\Delta t_1}{pT + (p-1)T_{max}} = \frac{R}{N-p+1} + \delta, \tag{6}$$

where R represents the sum over all the income rates received by all users and δ is an arbitrary positive constant. In words, Equation (6) says that after Δt_1 minutes the job J can pay at least $\frac{R}{N-p+1} + \delta$ dollars/minute. Next, let t_1 be the time at which Equation (6) becomes true. Clearly, at some time between $t = t_1$ and $t = t_1 + T_{max}$ all the jobs that were running at t_1 will finish and therefore other jobs will be scheduled to run. If J is not scheduled in this interval then there are at least $N-p+1$ processors that receive more dollars/minute than what J could offer.

Let $M = \sum_{i=1}^{m} M_i$ denote the total funds that all users have in their savings accounts at time t_1. Then between t_1 and some future time t_2 all the other jobs, excepting J, can spend at most $M + (R - r_{min})(t_2 - t_1)$ dollars for acquiring the resources. As observed before, if J is not scheduled by $t_1 + T_{max}$, then there are other jobs that have paid a higher price for at least $N-p+1$ processors. Let Δt_2 be the time interval such that

$$\begin{aligned}\frac{M + (R - r_{min})\Delta t_2}{(N-p+1)\Delta t_2} &= \frac{M}{(N-p+1)\Delta t_2} + \frac{R - r_{min}}{N-p+1} = \\ &= \frac{R}{N-p+1} + \delta.\end{aligned} \tag{7}$$

Since the first term on the left-hand side monotonically decreases with the interval Δt_2, this interval represents the *maximum* interval of time for which the $(N-p+1)$ processors can be paid at a rate greater than $\frac{R}{N-p+1} + \delta$ dollars/minute. Hence the job J will be scheduled by $t = \Delta t_1 + T_{max} + \Delta t_2$, since then it can pay more than $\frac{R}{N-p+1} + \delta$ dollars/minute (from Equation (6)).

[3] This is a realistic assumption since the total number of active users is bounded by the total number of users who have accounts on that computer.

4 Experimental Results

We have implemented a simple simulator in which we consider a parallel computer with $N = 128$ identical processors and 10 independent users, to validate our model. We assume that jobs of any user belong to only one of three classes (see Table 2). The jobs are assumed to come from a single Poisson source with mean arrival rate λ (measured in jobs/minute). By the decomposition property of a single Poisson process into m output streams ([18], Sec. 6.4), we can divide the initial job stream into ten independent streams, and therefore every user i can be modeled as an independent Poisson source from which jobs arrive with a mean rate $p_i\lambda$ (where p_i is the probability that a job comes from user i). Further, we denote by q_{i1}, q_{i2} and q_{i3} the probability that a job that comes from user i belongs to class 1, 2 and 3, respectively. Thus, the mean arrival rate of a job from user i belonging to class j is $q_{ij}p_i\lambda$.

The job service time is assumed to have a biphase hyperexponential distribution [13]. The relative values for the average service time and coefficient of variation for each class (see Table 2) are derived from the observed workload on an Intel iPSC/860 hypercube at NASA Ames, reported by Feitelson and Nitzberg [6]. [4]

In the following discussion, for ease of notation, we number all the jobs in the system during the simulation from J_1 to J_n. Let T_i represent the execution time of J_i, using the number of processors requested by the job. Let s_i represent the *system response time* for job J_i, the difference between the time when the job completes execution and the time when the job is submitted by the user. Thus $s_i = T_i + w_i$, where w_i is the time the job J_i waits before it is executed. Denote the ratio between the system response time and the service time for job J_i by $u_i = s_i/T_i$. Observe that u_i is greater than or equal to one.

Following Naik, Setia and Squillante [16], we use two performance metrics in analyzing the model: the *mean system response time* $\overline{S}$, and the mean ratio of a job's system response time to its service time $\overline{U}$ (we call this *mean user response* for short):

$$\overline{S} = \lim_{n\to\infty} \frac{1}{n}\sum_{i=1}^{n} s_i; \quad \overline{U} = \lim_{n\to\infty} \frac{1}{n}\sum_{i=1}^{n} u_i. \qquad (8)$$

Note that $\overline{S}$ measures the performance from the system's point of view, while $\overline{U}$ measures the performance from the user's point of view [16].

[4] Since we consider a more general architecture than an iPSC/860 hypercube, we assume that the number of processors that a job requests is uniformly distributed. For example, a job that takes 64 processors on a hypercube is assumed to request any number of processors between 32 and 64, with equal probability. Also, we have omitted the very large jobs that request all 128 processors in Feitelson and Nitzberg's data, since these jobs are run at night, when the load is light. Finally, we have not used the *absolute* values for service-times as given in [6]; instead we have chosen values that approximate the ratios between the service-times of different classes.

Let $\overline{E}$ be the mean of the cumulative computation time over all jobs submitted to the system. Then, we define the *system load* ρ as the fraction between the total demand received by the system in one time unit ($\lambda\overline{E}$), and the available computation time per time unit (N, since there are N processors); i.e., $\rho = \lambda\overline{E}/N$.

Class type	*Number of processors*	*Service time*	*Coefficient of variation*	q_{ij}
1	1-16	50	4	0.7
2	16-32	100	2.5	0.2
3	32-64	200	1.8	0.1

Table 2. The workload characteristics.

In the first experiment we compare the microeconomic scheduling policy (*ECON*) with two different variable-partitioning (*VP*) policies ([5], Sec. 3.2.3). A *VP* policy allocates to each job the exact number of processors it requests; the processors are not partitioned into predetermined subsets. The two policies we consider are the following:

- *FCFS*—This is the simplest policy. The jobs are placed in a first-come first-served (FCFS) queue; if there are enough free processors then the first job from the queue is scheduled for execution. If not, the job waits till the requested number of processors becomes free.
- *RES*—In this case, if a sufficient number of processors are not available to run the next job from the queue, the scheduler reserves processors for this job for the earliest time in the future when the required number of processors are available. Further, to make use of the idle processors until that time, the scheduler searches the queue and schedules the earliest jobs whose requests can be satisfied before these processors need to be dedicated to the job with the reservation.

The *FCFS* policy is expected to perform the worst among these policies, since it tends to heavily penalize small jobs when the system load is high. For, suppose the first job in the queue asks for a large number of processors and its request cannot be satisfied. Then, subsequent jobs have to wait, even if there are enough free processors in the system to satisfy their needs. The *RES* policy eliminates this problem; if a large job cannot run immediately, the scheduler searches for subsequent jobs whose requests can be satisfied. Notice that the *RES* policy is a special case of the *ECON* policy in which the income rate of every user is zero (if we assume that the scheduler selects the job that arrives first among jobs that offer the same price).

In the *ECON* policy we assume that that every user has the same income rate equal to 100 dollars/minute. We also assume that a user distributes his income

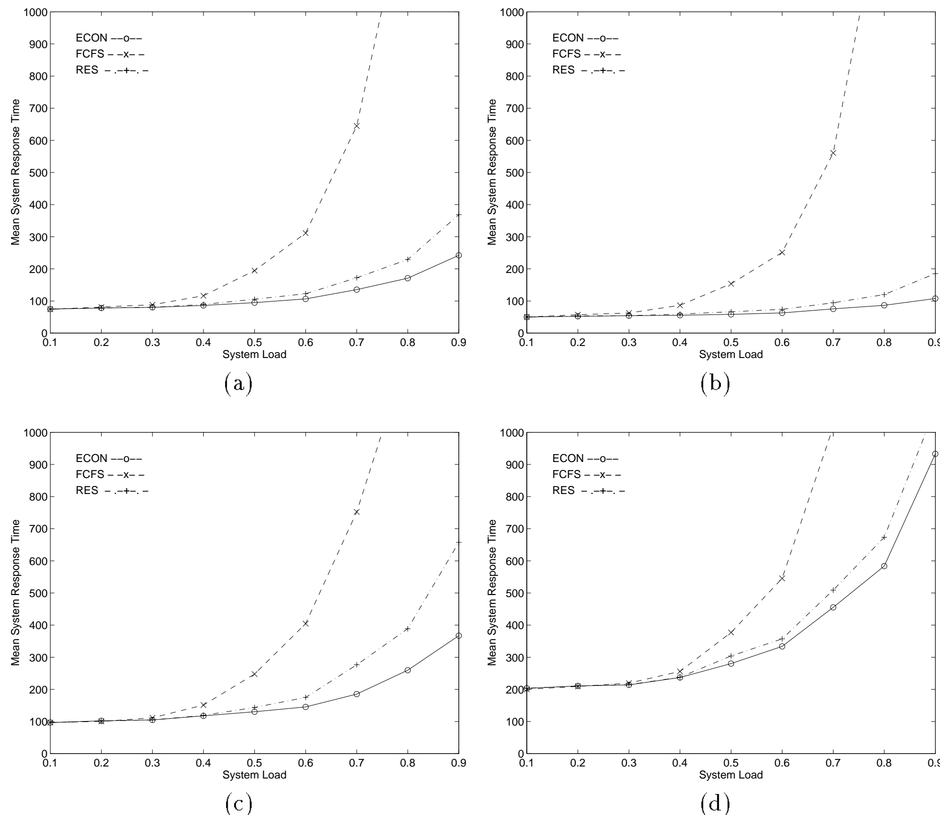


Fig. 3. The mean system response time $\overline{S}$ for: (a) all jobs, (b) jobs in class 1, (c) jobs in class 2, and (d) jobs in class 3.

equally between jobs from different classes, and thus the coefficient associated with each class is equal to 1/3. In each of the following experiments, we generate a system load ρ between 0.1 and 0.9, by suitably varying λ. To attain steady state we run each experiment (for every value of ρ) for 500,000 time-units. [5]

Figure 3(a) shows the mean system response time, $\overline{S}$, for all three policies for values of ρ between 0.1 and 0.9. When $\rho \leq 0.3$, all the policies offer almost the same performance. In this regime, there are few jobs in the system and there are enough processors to satisfy all the incoming requests. Next, for $\rho \geq 0.3$ the mean response time for the *FCFS* policy begins to increase sharply. This is because the large jobs monopolize the resources at the expense of small jobs. Finally, when ρ exceeds 0.4, the *ECON* begins to outperform the *RES* policy.

[5] In the current implementation we have not changed the price of computation over time as described in Equation (2); since we consider only constant workloads (ρ is fixed), we assume that the price is also constant in the steady state.

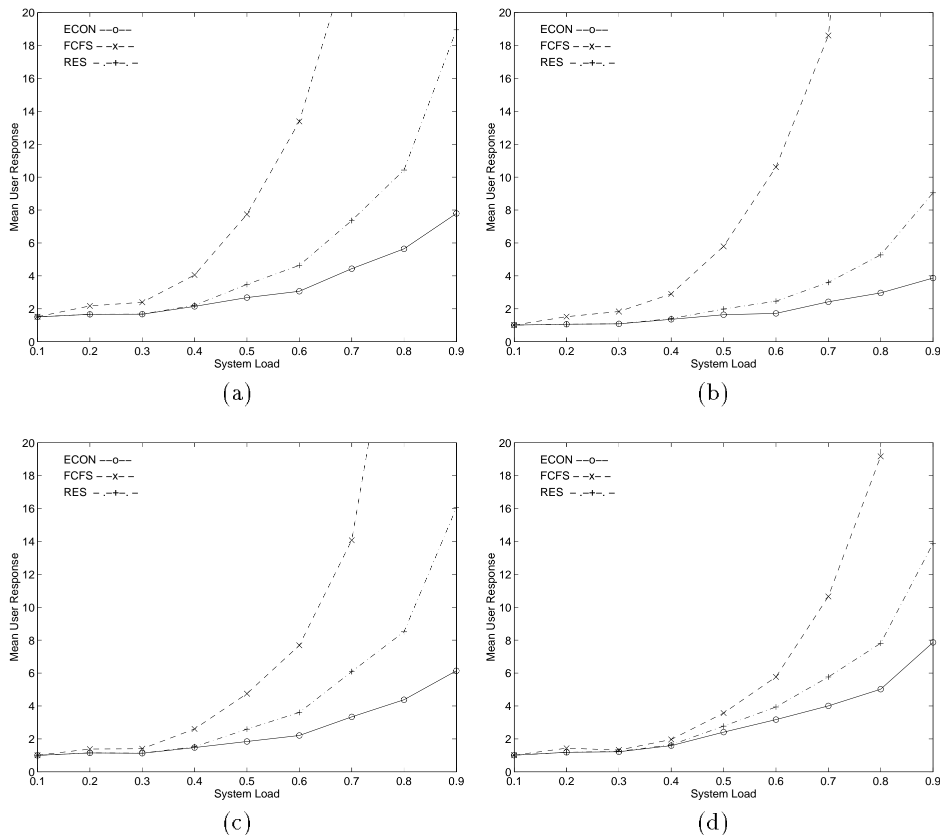


Fig. 4. The mean user response $\overline{U}$ for: (a) all jobs, (b) jobs in class 1, (c) jobs in class 2, and (d) jobs in class 3.

The improvement in $\overline{S}$ obtained with $ECON$ over RES is significant: When $\rho = 0.9$, $\overline{S}$ decreases by more than 34%. Figures 3(b), 3(c) and 3(d) compare the system response times for each class of jobs. As expected, the biggest gain is for small and medium jobs (classes 1 and 2). This is because the $ECON$ policy asks a job to pay not only for the computation time it needs, but also for the wasted time. This favors smaller jobs, since we expect that the larger the number of processors a job requests, the greater is the wasted time for which the job has to pay. Next, Figures 4(a), 4(b), 4(c) and 4(d) show the mean user response ($\overline{U}$) for the three policies; first, for all jobs combined, and next, for each class of jobs. The behavior of the mean user response as a function of the arrival rate, and as a function of the job class, is quite similar to the behavior of the system response time; the advantage of the $ECON$ policy relative to the other policies is even greater.

In the next experiment we study how the user income rate influences the

user performances. For this experiment we consider three different income rates for the first user, 50, 100 and 200 dollars/minute, while the income rates for all other users remain unchanged at 100 dollars/minute. Let $\overline{W}(R_i)$ denote the mean user waiting time for user i when his income rate is R_i. Figure 5 shows that the waiting time for the first user is inversely proportional to his income rate, when the mean job arrival rate is sufficiently large. For instance, when $\rho = 0.9$ and $R_1 = 50$ dollars/minute, the mean user waiting time is 186% of the value when $R_1 = 100$ dollars/minute, while for $R_1 = 200$ dollars/minute it is 55% of this value. To see why this happens, consider the case when $R_1 = 200$ dollars/minute. Since the first user receives twice as much income as the others, he can transfer money to his jobs roughly twice as fast. Therefore the price per minute offered by his jobs increases proportionally faster, and consequently the mean waiting time of these jobs reduces by approximately a half.

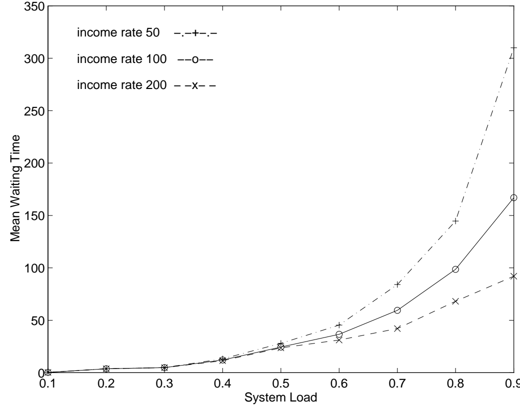


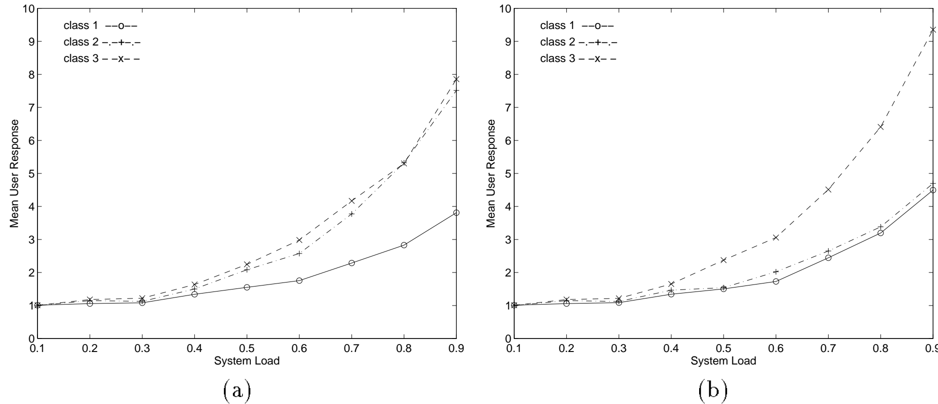
Fig. 5. The mean waiting time for user 1 for three income rates 50, 100 and 200 dollars/minute. All other users have an income rate of 100 dollars/minute.

In the last experiment we evaluate an adaptive strategy that controls the relative user response for each class. More specifically, let $\overline{U}_1$, $\overline{U}_2$, $\overline{U}_3$ be the mean user responses for jobs in class 1, class 2 and class 3, respectively. Our goal is to enforce certain ratios between the mean user responses for each class of jobs, i.e. $\overline{U}_1 : \overline{U}_2 : \overline{U}_3 = a_1 : a_2 : a_3$, where a_1, a_2 and a_3 are predefined constants. In other words, we would like each class to satisfy

$$\frac{\overline{U}_i}{\overline{U}_1 + \overline{U}_2 + \overline{U}_3} = \frac{a_i}{a_1 + a_2 + a_3}; \quad \text{for } 1 \leq i \leq 3.$$

To achieve this objective, the user periodically adjusts the coefficients associated with every class (see Section 2.4) according to the following equations:

$$\alpha_i^k = \alpha_i^{k-1} \frac{\overline{U}_i^{k-1}}{\overline{U}_1^{k-1} + \overline{U}_2^{k-1} + \overline{U}_3^{k-1}} \cdot \frac{a_1 + a_2 + a_3}{a_i},$$



(a) (b)

Fig. 6. The measured mean user responses for each class: $\overline{U}_1, \overline{U}_2, \overline{U}_3$. The desired ratios are: (a) $\overline{U}_1 : \overline{U}_2 : \overline{U}_3 = 1 : 2 : 2$, and (b) $\overline{U}_1 : \overline{U}_2 : \overline{U}_3 = 1 : 1 : 2$.

where $\overline{U}_i^k$ represents the mean user response for jobs belonging to class i at the kth iteration. Obviously, we have $\overline{U}_i = \lim_{k\to\infty} \overline{U}_i^k$. Notice that whenever $\overline{U}_i^k$ is larger than expected, i.e., $\overline{U}_i^k/(\overline{U}_1^k + \overline{U}_2^k + \overline{U}_3^k) > a_i/(a_1 + a_2 + a_3)$, then α_i^k increases and therefore the jobs in class i will receive a larger share of the user income. Conversely, if $\overline{U}_i^k$ is smaller than expected, then the user decreases the share of the income allocated to jobs in class i. We update the coefficients every $2,000$ time-units in our experiment.

Figure 6(a) shows the mean user response for each class of jobs when $a_1 = 1$ and $a_2 = a_3 = 2$. Again, when the system load is low there is not much the algorithm can do, since there are few jobs in the system and the resources are plentiful. On the other hand, adaptive control becomes increasingly efficient when the system load increases. For example, when $\rho = 0.9$, the measured mean user response ratios are $U_1 : U_2 : U_3 = 1 : 1.97 : 2.06$, which is close to the prescribed ratios $1 : 2 : 2$. Finally, Figure 6(b) show the mean user responses for a different set of ratios: $a_1 = a_2 = 1$ and $a_3 = 2$. In this case, when $\rho = 0.9$, the measured ratios $U_1 : U_2 : U_3 = 1 : 1.04 : 2.08$ are again close to the prescribed ratios.

5 Conclusions and Future Work

We have applied the microeconomic paradigm to schedule computation-bounded jobs on parallel systems. Our simulation results show that the microeconomic scheduler compares favorably with other variable partitioning policies both in terms of system and user performances. Additionally, the scheduler guarantees an adequate level of *fairness* in allocating resources among the users. Finally, by using a simple adaptive mechanism that adjusts the rate at which money is

transferred from the user savings account to a job expense account, the scheduler *controls* the relative job performances.

Many open problems remain.

We are currently extending the model to schedule jobs that specify a minimum and a maximum number of processors, and which can be allocated a number of processors within this interval at load-time. (We intend also to consider jobs that can *dynamically* change the number of processors *during* execution). The idea is to study the trade-off between the number of processors a job requests and the price it has to pay. Notice that if a job J_{ik} reduces the number of processors it requests, then the price it pays decreases for two reasons: First, the wasted time a job pays for decreases with fewer processors. Second, the cumulative computation time (E_{ik}) decreases if the job's speedup is sub-linear. Moreover, when requesting fewer processors, the waiting time is also likely to decrease. Therefore it would be possible for a job to obtain a better response time using fewer processors and paying less (if the decrease in the waiting time offsets the increase in the service time T_{ik})!

A second area for future work is to extend the model to other system resources such as memory and I/O bandwidth. One difficulty here is correlating the allocation of the various resources. For example, when a job buys computation time it has also to buy enough memory; otherwise instead of computing, it has to wait for the memory pages to be swapped in and out.

Third, it will be interesting to explore other policies for transferring funds from a user's savings account to a job expense account. It might be worth considering variable user income rates. The idea would be to allocate a share of the system resources to every user and then to dynamically adjust the income rate in order to ensure that every user receives his share. Here, the trade-off is between increasing algorithm overhead and increasing accuracy of control.

We believe that the microeconomic paradigm may serve as a unifying theme for multiprocessor scheduling. We have seen that the variable partitioning scheme with job reservations is a special case of the microeconomic scheduler (when the income rates are zero). We hope to show in future work that other scheduling policies might also be obtained by suitably choosing the parameters in the microeconomic paradigm.

References

1. J. Blazewicz, M. Dror and J. Weglarz, "Mathematical Programming Formulations for Machine Scheduling: A Survey", *European Journal of Operational Research*, No. 51, 1991, pp. 283-300.
2. D. R. Cheriton and K. Harty, "A Market Approach to Operating System Memory Allocation", URL page: http://www-dsg.stanford.edu/Publications.html, Stanford University.
3. E. G. Coffman, M. R. Garey, D. S. Johnson, R. E. Tarjan, "Performance Bounds for Level-Oriented Two-Dimensional Packing Algorithms", *SIAM Journal of Computing*, Vol. 9, No. 4, November 1980, pp. 808-826.

4. K. E. Drexler and M. S. Miller, "Incentive Engineering for Computational Resource Management", in [11], pp. 231-266.
5. D. G. Feitelson, "A Survey of Scheduling in Multiprogrammed Parallel Systems", *Research Report RC 19790*, IBM T.J. Watson Research Center, 1994.
6. D. G. Feitelson and B. Nitzberg, "Job Characteristics of a Production Parallel Scientific Workload on the NASA Ames iPSC/860", D. G. Feitelson and L. Rudolph (eds.), *Lecture Notes in Computer Science*, Vol. 949, Springer-Verlag, 1995.
7. D. Ferguson, Y. Yemini and C. Nikolau, "Microeconomic Algorithms for Load Balancing in Distributed Systems", *Proc. of the 8th International Conference on Distributed Computer Systems*, IEEE, 1988, pp. 491-499.
8. M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, San-Francisco, 1979.
9. R. L. Graham, "Bounds on Multiprocessing Timing Anomalies", *SIAM Journal of Applied Mathematics*, Vol. 17, No. 2, March 1969, pp. 416-428.
10. J. L. Hellerstein, "Achieving Service Rate Objectives with Decay Usage Scheduling," *IEEE Transactions on Software Engineering*, Vol. 19, No. 8, August 1993, pp. 813-825.
11. B. Huberman (ed.), *The Ecology of Computation*, North-Holland, 1988.
12. J. Kay and P. Lauder, "A Fair Share Scheduler", *Communication of the ACM*, Vol. 31, No. 1, January 1988, pp. 44-45.
13. S. Majumdar, D. L. Eager, and R. B. Bunt, "Scheduling in Multiprogrammed Parallel Systems", *Proceedings of the 1988 ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, pp. 104–113.
14. T. W. Malone, R. E. Fikes, K. R. Grant and M. T. Howard, "Enterprise: A Market-Like Task Scheduler for Distributed Computing Environments", in [11], pp. 177-205.
15. M. S. Miller and K. E. Drexler, "Markets and Computation: Agoric Open Systems", in [11], pp. 133-176.
16. V. K. Naik, S. K. Setia and M. S. Squillante, "Performance Analysis of Job Scheduling in Parallel Supercomputing Environments", *Research Report RC 19138*, IBM T.J. Watson Research Center, 1993.
17. M. G. Norman and P. Thanisch, "Models of Machines and Computation for Mapping in Multicomputers", *ACM Computing Surveys*, Vol. 25, No. 3, September 1993, pp. 263-302.
18. K. S. Trivedi, *Probability and Statistics with Reliability, Queuing and Computer Science Applications*, Prentice-Hall, 1982.
19. C. A. Waldspurger, T. Hogg, B. A. Huberman, J. O. Kephart and W. S. Stornetta, "Spawn: A Distributed Computational Economy", *IEEE Transactions on Software Engineering*, Vol. 18, No. 2, February 1992, pp. 103-117.