Spawn: A Distributed Computational Economy

Carl A. Waldspurger[*] Tad Hogg Bernardo A. Huberman
Jeffrey O. Kephart[†] Scott Stornetta[‡]

Dynamics of Computation Group
Xerox Palo Alto Research Center
Palo Alto, CA 94304 USA

May 1989
Revised: November 19, 1990; October 21, 1991

Abstract

We have designed and implemented an open, market-based computational system called *Spawn*. The *Spawn* system utilizes idle computational resources in a distributed network of heterogeneous computer workstations. It supports both coarse-grain concurrent applications and the remote execution of many independent tasks. Using concurrent Monte-Carlo simulations as prototypical applications, we explore issues of fairness in resource distribution, currency as a form of priority, price equilibria, the dynamics of transients, and scaling to large systems. In addition to serving the practical goal of harnessing idle processor time in a computer network, *Spawn* has proven to be a valuable experimental workbench for studying computational markets and their dynamics.

Index Terms: Concurrent systems, distributed systems, dynamic load sharing, microeconomic algorithms, priority mechanisms, resource allocation, scheduling.

1 Introduction

Recent advances in computer technology have led to a proliferation of powerful networked computers that form large distributed systems. A major difficulty in fully utilizing these systems is managing the complexity of coordinating many tasks on multiple processors.

Simple centralized allocation of tasks becomes increasingly difficult as larger parallel machines and distributed networks are developed. Since data is distributed and rapidly changing, a central controller cannot access all of the information needed to effectively plan detailed behavior. Instead, reliability and rapid

[*]Author's present address: MIT Laboratory for Computer Science, 545 Technology Square, Cambridge MA 02139
[†]Author's present address: IBM T.J. Watson Research Center, Yorktown Heights, NY 10598
[‡]Author's present address: Bellcore, 445 South Street, Morristown, NJ 07960

response to local changes require more localized control over resources. We are thus confronted with the challenge of developing schemes for decentralized resource allocation.

1.1 Motivations

The emergence of large, decentralized systems raises some fundamental questions [13]. Essentially, there is a need for a general theoretic guide to the behavior of large collections of locally-controlled, asynchronous and concurrent processes interacting with an unpredictable environment. In particular, this requires understanding the relation between the overall behavior of the system and that of its constituents, whose decisions are based upon local, imperfect, delayed, and conflicting information. These characteristics, which are also found in social and biological communities, lead us to refer to these collections of interacting processes as *computational ecosystems* [14]. This form of computation has also been termed *open* because it violates the closed-world assumption of traditional computer science [13].

Given this similarity, it is not surprising that there has been some speculation concerning the efficacy and desirability of market mechanisms in distributed computer networks. Since market devices such as auctions and prices facilitate resource management in human societies, one might expect them to be similarly useful in computer networks – a proposal which has been elaborated in some detail [27]. For example, a price mechanism could allow machines with different capabilities to have different values, enabling tasks to flexibly devote their currency to the resources most important for them.

Despite the intuitive appeal of the economic approach to open computational systems, few empirical or quantitative results exist to confirm or deny its suitability. Several differences between human economies and proposed computer systems based on similar principles bring the analogy into question. First, human decision-making is notoriously difficult to quantify. In addition, human beings are extraordinarily diverse in their opinions and methods for making decisions, a fact which can lead to more stability in human economies than in potentially less-diverse computational networks. Finally, decisions made by computers can take place in times which are orders of magnitude faster than those of human decisions. Recent work [14, 17] suggests that the time-scale on which decisions are made has a strong effect on the dynamics of the system. These studies of computational ecosystems also indicate the existence of several other phenomena, such as large-amplitude oscillations and chaotic behavior, that might have detrimental effects on the performance of computer systems. Even if these issues are resolved, there remain practical questions such as the number of tasks needed in order to exhibit meaningful market-like behavior, and how to exploit knowledge of this behavior to efficiently manage resources.

1.2 Spawn Project Overview

In order to understand the behavior of a computational economy, we designed *Spawn*, a market-based computational system that runs in a distributed network of heterogeneous high-performance workstations. By studying a computational economy in actual use, we were able to guide its design and reformulate our notions of appropriate performance criteria in response to our observations.

At a more practical level, since a significant amount of computer equipment sits idle for large fractions of each day, *Spawn* was designed to allow users to tap these otherwise wasted resources. If such an environment is viewed as a large multiprocessor instead of a collection of independent networked workstations, *Spawn* supports both coarse-grain concurrent applications and the simultaneous execution of many unrelated tasks. We have implemented useful *Spawn* applications in both of these categories, primarily in the realms of concurrent Monte-Carlo simulations and remote document formatting.

In this paper, we concentrate on the behavior of large, concurrent Monte-Carlo applications. There are several reasons for this focus. First, allocating resources in a concurrent application raises a number of interesting questions concerning interprocess coordination and management that simply don't exist when executing unrelated tasks. Moreover, although market-like algorithms have been applied to the scheduling of independent tasks [10, 24], there appears to be no previous work on microeconomic approaches for managing resources in concurrent applications. Finally, Monte-Carlo simulations represent a significant class of applications which can make use of large quantities of idle CPU time. Such simulations are frequently used to understand the behavior of systems with many degrees of freedom, such as matter in various states, ecosystems, and economic models. Although these Monte-Carlo simulations are typically performed on expensive supercomputers, they can easily be parallelized, making them a natural candidate for distributed computation. Other potential applications of *Spawn* include remote compilation, and the concurrent computation of graphic frames for computer animation.

As will become apparent, *Spawn* has satisfied our two-pronged research goal: not only has it allowed us to harness the idle-time of a computer network, but it has also proven to be a valuable experimental workbench for studying computational markets and the dynamical behavior of open systems.

1.3 Contributions

The *Spawn* project has resulted in several original contributions, which are summarized below:

- The first implementation of a distributed computational economy. *Spawn* is a working system that serves as a proof of concept for a microeconomic approach toward computational resource management.

- The first application of a microeconomic approach to resource management for concurrent programs. *Spawn* introduces several new concepts for abstract, application-level resource management of cooperative processes in concurrent systems. These include the *Spawn sponsorship hierarchy* mechanism, the use of continuous funding *rates* to simplify funding allocation, and a *conservation of funding* property across process forks.

- The use of monetary funding units as an abstract form of priority in distributed and heterogeneous systems. Experiments with *Spawn* demonstrate that the use of funding as priority can be very effective.

- The use of price information to control adaptive expansion and contraction of process trees in concurrent applications.

- The first examination of the price dynamics of a computational economy. The study of price dynamics is important for understanding the applicability and stability of market mechanisms in computational systems.

1.4 Roadmap

In the next section, we place *Spawn* in context by examining relevant work in a number of related areas. In section three we describe the basic mechanisms underlying the *Spawn* system. Section four presents and analyzes the results of several quantitative experiments in which system parameters were varied in order to assess their influence on its overall behavior. These experiments were supplemented with simulations of *Spawn* that provide insight into its scaling behavior for large networks. In section five we discuss the limitations of *Spawn*, and relate a number of lessons that we have learned from our experience with the system. Finally, we conclude that market-like resource management methods based on a price mechanism can work effectively in a real distributed system, and highlight opportunities for further research.

2 Related Work

Since *Spawn* is a rather unusual experimental system, this section presents an overview of the diverse body of related work. It is important to keep in mind that *Spawn* was not intended to optimally solve any one particular problem. Instead, it is an empirical investigation of the efficacy and desirability of market mechanisms in distributed computer networks.

2.1 Exploiting Idle Time

An early experiment in distributed computation was the *Worms* project [31]. In this system, a *worm* was a computation composed of multiple segments, each executing on a different machine. A worm segment continually searched for idle machines on the network into which it could replicate itself, causing the worm as a whole to "grow". Although the *Worms* project highlighted important directions for future research, it was ultimately rendered impractical. Its primary limitations included a rudimentary control structure and the lack of provisions for protection and sharing on the early Xerox Alto workstations on which it executed. *Spawn* is related to the *Worms* project in that both share the vision of permitting an expanding computation to inhabit idle networked workstations. *Spawn's* mechanisms for resource management and control, however, are more sophisticated and flexible than the simple strategies employed in *Worms*.

Another type of system is exemplified by the *Condor* scheduling system [20], a robust system that identifies idle workstations and schedules background jobs on them. A key concern in this system is achieving some measure of fairness when both heavy and casual users compete for idle time [28]. *Spawn* is similar to *Condor* in its goal of harnessing idle time on networked workstations while simultaneously addressing the issue of fairness. However, *Spawn* differs dramatically from *Condor* in its capabilities, mechanisms, and assumptions. One key aspect of *Spawn* is its support for concurrent applications; an aspect that is absent in systems such as *Condor*. This distinction leads to significantly different assumptions. In systems like *Condor*, the typical scenario is that there are plenty of idle machines, but they are difficult to find. In *Spawn*, we assume that there are users with large concurrent applications capable of utilizing all available CPU time. Thus, the central concern in *Spawn* is the fair allocation of "idle" resources among concurrent applications when there are several applications competing for those resources.

2.2 Computational Resource Management

Improving methods for managing computational resources is a central concern of researchers involved in the study of resource allocation and scheduling. Conventional schedulers for computer systems rely upon centralized global controllers to allocate resources among tasks [4, 30], a technique which is not practical for large distributed systems with a changing environment.

The problem of scheduling in distributed computer systems has been a topic of active research for many years [35]. Most of the work on distributed scheduling has focused either on deterministic mathematical models or on the formulation of useful heuristics. The mathematical treatments of scheduling, such as those based in graph theory [3, 21], usually make simplifying assumptions that are not viable in real systems. Heuristic approaches, more common in practice, involve the estimation and communication of load information for making distributed scheduling decisions. A variety of heuristic methods have been studied, including random, limited exchanges of information [2], techniques from expert systems and rule-based programming [22], knowledge-based solutions [29], statistical time-series analysis [12], and distributed bidding and market-like negotiation metaphors [32, 24]. This last class of heuristic algorithms, which are most related to *Spawn's* distributed computational economy, are discussed in the following section.

2.3 Computational Markets

Given the similarity between resource allocation problems in distributed computing and economics, it is not surprising that there has been much discussion of applying markets to computation. An early predecessor to computational markets was a manual "futures market" used to allocate blocks of time on a single-user PDP-1 [34]. The most detailed description of possible computational markets has been made by Miller and Drexler [27, 6]. Their work on processor scheduling is most relevant to *Spawn*, but is primarily concerned with the efficient auctioning of processor time within a *single* serial computer system. Nevertheless, their uniprocessor algorithm shares with *Spawn* the characteristic of linearly increasing bids across a series of second-price auctions [6].

Enterprise is a decentralized market-like scheduler for load-sharing in distributed computing environments [24]. *Enterprise*, similar in many respects to the Contract Net protocol [32, 5], is organized around a sequence of *announcement*, *bid*, and *award* actions. In the announcement stage, a *client* broadcasts a request for bids which includes a description of the task to be run, an estimate of required processing time, and a numerical task priority. Idle *contractors* reply with bids containing estimated completion times for the client's announced task. A client collects bids from responding contractors. After a pre-determined amount of time, a client evaluates all of the bids it has received and awards its task to the best bidder (usually the one with the earliest estimated completion time). The *Enterprise* system protocol allows for mutual selection of clients and contractors; contractors can decide which clients to serve based upon task information, and clients can choose among available contractors. In addition to implementing the system on networked workstations, Malone performed a set of simulations to test the effects of various pre-set system parameters on the performance of the system. The results indicated that over a wide range of parameters, pooling tasks in a distributed system using *Enterprise* resulted in significant performance improvements over running the same tasks locally on their home machines.

Like *Worms*, *Enterprise* suffered from the unfortunate protection limitations of early computer workstations. Aside from implementation-related problems, *Enterprise* was also limited by a number of fundamental design decisions. In contrast to *Spawn*, the system had no provisions for market price information. The absence of a price mechanism inhibited the flexibility of the system by constraining the criteria by which contractors and clients could make decisions. For example, clients were incapable of making tradeoffs between fast, expensive contractors and slow, relatively cheap contractors. Moreover, price information would have eliminated the need for complicating the system with artificial "priorities". *Spawn* also differs from *Enterprise* in its support for concurrent applications in a heterogeneous system; *Enterprise* was limited to the execution of independent tasks on compatible workstations.

Ferguson, Yemini, and Nikolaou have examined *microeconomic algorithms* for load balancing in distributed computer systems [10]. Their scheme, like the approach described by Miller and Drexler, involves a competitive market for resources based upon a price mechanism. In their system, *jobs* compete for communication and processing resources by bidding on auctions held by *processors*. Jobs receive an initial allocation of money upon entry to the system, which they must use to purchase processing time, and to pay for communication charges when crossing network links between processors. Processors auction off CPU time and communication bandwidth to jobs, trying only to selfishly maximize their own revenues. Jobs bid on affordable processors based upon a preference relation that is a function of price and service time. When a processor resource becomes idle, it holds an auction to determine which job will next get that resource. In the spirit of Adam Smith's classic "invisible hand" argument from economics, it is reasoned that local optimizations by selfish jobs and processors will lead to globally desirable resource allocation. Ferguson *et. al.* provide and analyze simulation results, leading them to conclude that their competitive economic algorithms achieve a globally effective allocation of resources that is comparable, and in many cases superior, to traditional algorithms. However, it is important to note that these results pertain to the problem of scheduling independent tasks in a network of homogeneous processors. Their sharp focus on the collective performance of a set of independent processes neglected the effects of funding allocation policies. In fact, the initial allocation of money to jobs was performed arbitrarily in their simulations. Although they found that *aggregate* system performance is fairly insensitive to the initial allocation policy, they did not analyze the effects on the performance of individual jobs.

In contrast to the simulations described in [10], *Spawn* was explicitly designed to support concurrent applications in a heterogeneous distributed system. With *Spawn*, we explore the effect of various funding strategies and their impact on performance and fairness of resource distribution.

3 The Spawn System

3.1 Overview

At a very high level of description, *Spawn* is organized as a market economy composed of interacting buyers and sellers. The commodities in this economy are computer processing resources, specifically slices of CPU time on various types of computer workstations in a distributed computational environment.

Buyers are users who wish to purchase time in order to perform some computation. *Sellers* are users who wish to sell unused, otherwise-wasted processing time on their computer workstations. A concrete example of a buyer is a scientist who wants to run a large, concurrent Monte-Carlo simulation. A typical seller is a user who is not actively using his personal workstation. Neither buyers nor sellers need to be physically co-present with their machines in order to participate in the *Spawn* economy. A seller executes an *auction* process to manage the sale of his workstation's processing resources, and a buyer executes an *application* that *bids* for time on nearby auctions and manages its use of computer processing resources. In the *Spawn* economy, monetary *funds* encapsulate resource rights, and *price* equates the supply and demand of processing resources.

3.2 System Processes

The sale of processing resources is handled by a set of system processes that executes on each machine involved in the *Spawn* economy.

3.2.1 Auctions

An *auction* process controls the sale of an idle workstation's resources, and handles messages from application tasks that want to purchase slices of its time. Each workstation only executes a single application task per time slice. Thus, a task that has purchased a time slice on a machine is guaranteed exclusive access to that machine for the duration of the time slice.

An auction continuously accepts bids on the *next available* slice of time; i.e., a block of time beginning after the termination of the slice purchased by the currently executing application. A *bid* consists of a length of time, a quantity of funds, and a brief task description.

An auction follows a bid-processing strategy defined by the seller who initiated it. Auction strategies are parameterized by the minimum and maximum allowable time slice lengths that can be sold, and a function that expresses the auction's strategy in terms of slice length, current bids, and other market values. For example, depending on market conditions, an auction process may decide to give discounts or charge premiums for purchases of long time slices. A simple strategy, used throughout this paper, is a linear function relating cost to time slice length. In practice, a bounded range of allowable time slice lengths is used to ensure that processes execute long enough to amortize start-up overhead, but not so long that a user returning to his "idle" workstation would be inconvenienced while waiting for a client process to terminate.

The auctions employed by *Spawn* are *sealed-bid, second-price* auctions. "Sealed" means that bidding agents cannot access information about other agents' bids, and "second-price" indicates that the amount paid by the winning agent is the amount offered by the next-highest competitive bidder.[1] This type of auction provides an incentive for agents to bid the amount that a time slice is actually worth to them, and has proved very effective in human markets. Empirical studies and theoretical analyses of auctions can be found in the economics literature [9, 11]. A sealed-bid, second-price auction leads to market prices similar to those which would be created by a familiar first-price auction where agents continually try to marginally outbid the current high bidder.[2] In our system, an auction does not commit to a bidder until the last possible moment; it accepts the highest bidder's task at the price set by the second-highest bidder when the current time slice is about to end.[3]

3.2.2 Resource Managers

A *resource manager* process is associated with each auction. The resource manager is responsible for initiating and monitoring the execution of the application task that purchased the current slice of processor time. If an application consumes more resources than it has purchased, the resource manager forcibly terminates it. To avoid terminating an application process before it has completed (e.g., due to an inaccurate processing time estimate), an application is given a *right of first refusal* before the next time slice is sold. This means that the currently-executing application is allowed to continue its execution as long as it can pay the going market price for *extension* time slices. This capability is provided because terminated processes are not allowed to be restarted and cannot migrate.[4] It is thus the application's burden to ensure that important computations are well-funded, or to cope with failure due to aborted computations.

The resource manager also serves as an interface between high-level applications and the rest of the *Spawn* system. We feel that a high-level application should not be encumbered with decision-making concerning the

[1] More specifically, suppose that agents A and B are the current highest and second-highest bidders, respectively, on a particular auction. If agent C then submits a bid greater than that of A, it will become the new highest bidder. The second-highest bid will be recorded as that of A if C and A are allowed to compete; otherwise, the second-highest bid will be remain that of B. The precise distinction between "competitive" and "friendly" agents is explained in the next section.

[2] We have experimented with first-price "English" auctions and found that they yield approximately the same results as sealed-bid second-price auctions, but incur a much higher computational cost due to the increased communication overhead.

[3] Note that if there is *no* second-highest bidder, the price is zero; i.e., in the absence of competition, resources are free.

[4] These constraints are imposed by the programming environment which we used to implement *Spawn*. A discussion of related implementation issues can be found in section 3.5.

low-level market mechanisms that locate, schedule, and purchase the resources necessary for its execution. At the same time, however, it should be possible for an application to exert some control over the general allocation of funds to allow for interesting strategies. The *Spawn* architecture provides a uniform mechanism with these capabilities.

Application processes, which will be described in more detail in the next section, simply provide high-level funding information for their subtasks. The resource manager encapsulates details about auctions and bidding procedures, and communicates with a set of nearby auctions. Each resource manager maintains a list of "neighboring auctions" which supply price and availability information. This list is easily modified, and has allowed us to experiment with a variety of logical interconnection topologies. Normally each machine is only connected to a small number of other machines, demanding highly decentralized decision-making with no global state or controls.

3.3 Application Processes

Applications are divided into manager and worker modules. An *application worker* is the primary computational task for which resources are allocated. Each worker has a corresponding manager process to which it may communicate by sending messages. An *application manager* coordinates the execution of some set of tasks in a distributed application. It arranges, via communication with the resource manager, to spawn child workers and submanagers responsible for various subtasks. The application manager thus contains the interface of the application to the *Spawn* system. A special *root application manager* resides on the top-level user's personal workstation and serves as the user-interface for a distributed computation.

The simplest *Spawn* applications are black-box applications. A root application manager requests the execution of a single remote task and provides some fraction of its available funding to pay for it. When the local resource manager has won a bid on an affordable auction, the remote task (consisting of both an application manager and an application worker) is run. The application worker is a black-box task (such as a document formatter) that does not directly interact with the *Spawn* system. Its corresponding application manager is a simple process that captures any output generated by the worker and sends it back to the root for display on the user's personal workstation.

More interesting applications may consist of a tree of tasks executing concurrently. In such applications, a typical worker performs intensive computation and periodically reports partial results to its immediate manager. This manager combines and processes incoming partial results and sends the aggregate results up to the next level in the management hierarchy. If partial results can be combined at each level of management, such reports can be efficiently combined in a concurrent, decentralized manner. Managers of decomposable tasks may also choose to *spawn* additional children to manage subtasks. This process is shown in Figure 1.

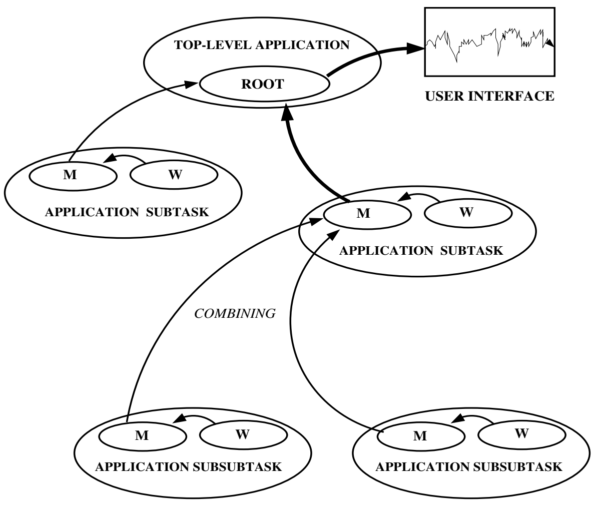

Figure 1: Application Process Hierarchy
Workers (W) report to their local managers (M), who in turn make reports to the next higher level of management. Upper management combines data into aggregate reports. Finally, the root manager presents results to the user.

To create a local worker process, an application manager sends a `spawn-worker` message to its local resource manager. This message contains an abstract task name and a list of task arguments. The abstract task name provides a level of indirection that facilitates the execution of tasks in a heterogeneous computing environment. A task name refers to a set of mappings, specified in a *task file*, which map particular workstation configurations onto network pathnames for the executable files that implement the named task for that configuration. A task file also defines application-specific ratings for each possible configuration. This information enables applications to specify the relative efficiency of executing a task on different hardware configurations, and is used by the resource manager to find the best match between a task and the resources for sale in the current market.

To create a remote manager for processing a subtask, an application manager sends a `spawn-manager` message to the resource manager. This message contains the information carried by the `spawn-worker` message, as well as several parameters that the resource manager needs in order to find resources for the task's execution. These include the estimated processing time for the task (normalized to a machine with an application-specific rating of unity), and a specification of the relative importance of time versus price, similar to the metric used in [10]. An application manager may avoid the introduction of unwanted competition (e.g., between two cooperating subtasks) by specifying user and group identifications that should be considered "friendly" when bidding on time for new subtasks. Such tasks are not considered to be competitors by auction processes.

Application tasks can easily send application-specific (i.e., opaque to the *Spawn* system) messages to their parent managers. Managers receiving messages from children can combine the information they contain to deliver aggregate reports to higher management. Of course this feature need not be used; for example, black box tasks never send messages to their managers.

3.4 Sponsors and Funding

A key issue that *Spawn* confronted was the development of a general mechanism for funding distributed computations. This problem is very different from that of scheduling independent tasks, since our primary concern is instead with the fair execution of concurrent, tree-structured computations.

The concept of *sponsored computations* was developed by researchers investigating linguistic constructs for the allocation of resources in actor programming systems [36, 1]. The most recent actor language, *Acore* [25], employs sponsors to control the rate at which concurrent threads of computation proceed. In Acore, every transaction (i.e., message-send and reply) in an actor computation must be sponsored. When a transaction is run, the runtime system requests an allocation of *ticks* from its sponsor. A tick is the basic unit of computational resource, and represents the resources required to perform a single message-send. A sponsor may either grant a number of ticks, or deny further funding, in which case the transaction's thread is aborted. In practice, Acore sponsors have primarily been used to deny funding to unwanted computations. More sophisticated sponsor strategies are difficult to reason about, since ticks are not sensitive to dynamic resource utilization information, and always represent the same quantity of resources.

Spawn extends and refines the notion of sponsorship in the context of a market-based computational economy. In *Spawn*, managers serve as funding sponsors for their children, dynamically controlling the relative fraction of funding allocated to each child task. Relative allocation of funds provides a clean high-level framework for sponsorship by removing low-level details, such as absolute funding amounts and the need for complex funding request and evaluation protocols. A simple manager strategy allocates equal amounts of funding to each child; more sophisticated strategies may take the relative workloads, progress, or cost-effectiveness of each child into account. In this way, *Spawn* provides basic support for the use of heuristics similar to the "interestingness" metric employed by *Eurisko* [18] for guiding exploratory computations.

The *Spawn* sponsorship hierarchy can be visualized as a tree of pipes, in which funds flow from the root node of an application to its managers. Each manager corresponds to a branch point in the tree consisting of a single input pipe, a reservoir, and a variable-width output pipe for each of its children. To control the relative amount of funding flowing to individual children and the amount of funding retained for its own future use, a manager can adjust the widths of its output pipes and the size of its reservoir. Although

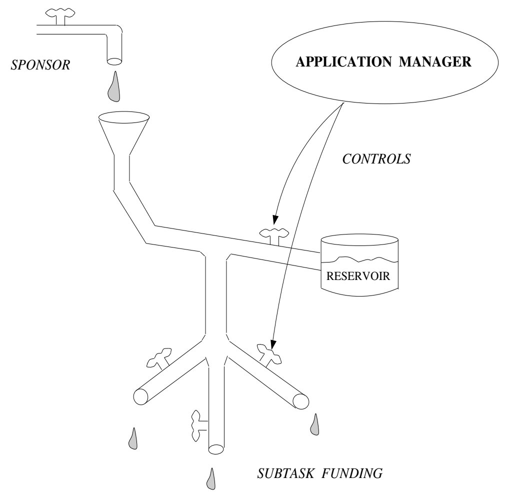

Figure 2: Spawn Sponsorship Hierarchy
An application's sponsor continually gives funds to its child tasks. The relative funding rates among child tasks are controlled by the application manager.

managers are free to allocate their incoming funding in any manner, they can neither create nor destroy funds. This *conservation of funding* property holds at all branch points in the application process tree.

The top-level manager for a computation controls the total amount of funding that is pumped into the root of the sponsorship hierarchy. Since funding cannot really be continuous, the root node delivers it in discrete drops, which split into finer drops at branch points. The flow of funds is illustrated in Figure 2.

For example, consider a homogeneous task that can spawn a large number of similar subtasks. A simple, effective manager strategy is to fund each child equally. Because the overall application can be viewed as a tree of managers with a branching factor greater than one, it is clear that funds introduced by the root manager will be subdivided and ultimately delivered to the leaves of the management tree. Since each leaf is actively bidding to spawn additional tasks, the distributed computation will be able to expand to more machines when prices are low and will be forced to shrink back to fewer machines when the market is not as favorable. A more intelligent manager could decide to heavily fund the most cost-effective or productive children.

At the highest level, the allocation of funding to users is negotiated by human system administrators. Some users may be able to earn the funds they spend by selling unused time on their personal workstations. Users with processing requirements that far exceed their earning potentials can be granted funding *income rates* that reflect their relative importance or need for resources.

3.5 Implementation

Spawn runs on a network of heterogeneous Unix workstations. The protection mechanisms offered by modern workstations (e.g., separate, per-process memory address spaces and other support for multi-processing) obviate many technical difficulties that plagued prior implementations of related systems such as *Enterprise* [24] and *Worms* [31]. A network of Unix workstations provides a minimal substrate upon which *Spawn* can be successfully implemented. Ideally, we would have preferred to take advantage of a sophisticated distributed operating system and a programming language designed for open systems [35, 15, 19, 25]. Unfortunately, these tools were neither generally nor uniformly available on the existing machines and networks that we used; many are still research prototypes. *Spawn* is written in the C programming language and utilizes the Sun RPC and NFS protocols for networked computing. *Spawn* has been successfully tested on networks containing Unix workstations manufactured by Sun Microsystems (Sun 3, Sun 4) and Digital Equipment Corporation (VAX family).

At the computational level, all *Spawn* processes communicate via an asynchronous message-passing protocol. Since processes may be short-lived (by design or due to lack of funds), *Spawn* provides a facility for dynamic re-routing of messages and a mechanism for the delegation of responsibility. Before terminating, an application manager may delegate responsibility for its children to any other manager in the sponsorship hierarchy; typically it will delegate to its own immediate sponsor. Additional implementation details can be found in [37].

Our computing environment imposed several limitations on the scope of the *Spawn* project. Since *Spawn* executes as an ordinary user-mode Unix application, process creation is an expensive operation, and processes are unable to migrate between machines. This limits us to very coarse-grained processes. Because we were not able to use a programming language designed for robust concurrent and distributed computation, we decided to limit the scope of our applications to those that could be easily parallelized and expressed using extensions to existing serial languages (such as C and FORTRAN).

There are two additional shortcomings in the present *Spawn* implementation. First, *Spawn* does not provide applications with robust recovery in the event of failure. User computations can be aborted due to machine failure or insufficient funding; it is currently the responsibility of applications to recover from such failures. A better solution would be for *Spawn* to utilize a substrate for reliable distributed computation that provides robust recovery from failure by using atomic transactions [19]. Second, *Spawn* is not secure. Although reasonable safeguards and checks have been included, no attempt has been made to protect the *Spawn* economy from malicious users intent upon forging currency or deliberately cheating agents [26, 16].

4 Experiments

4.1 Overview

In order to evaluate the *Spawn* system, we have conducted a number of experiments which served to quantify its ability to make use of idle machines, distribute resources among competing tasks, and respond to a changing environment. These experiments were based on asynchronous Monte-Carlo applications executing concurrently in a distributed network.

Monte-Carlo is a probabilistic algorithm that computes average properties of systems [33]. It consists of a large number of independent trials of a system, each in a different configuration. The desired average is then computed over the ensemble of these independent trials. Since errors in the computed averages are inversely proportional to the square root of the number of trials, accurate results require a large number of trials. This technique has been successfully employed in wide spectrum of applications, including the numerical evaluation of integrals, determining the behavior of complicated states of matter, and exploring interactions in ecosystems and economics.

The Monte-Carlo algorithm was ideal for our purposes. It is paradigmatic of simulation techniques requiring great amounts of CPU time, and it easily decomposes into an arbitrary number of subtasks, each performing a number of independent trials. By experimenting with simple Monte-Carlo applications, we were able to meaningfully explore microeconomic approaches for managing concurrent applications without making our experiments too complex and cumbersome to analyze.

The data presented in this section is derived from two separate sources: a working implementation of *Spawn* in a network of Sun workstations, and a detailed *Spawn* simulator based on the NEST simulation engine for distributed systems [7]. Simulation results are primarily used to establish the scaling properties of the system. Each experiment is labelled with the relevant data source and conditions under which the experiment was conducted. Typical logical interconnection topologies used in the experiments include small, fully-connected systems, and larger systems in which each node is locally connected to four others in a regular grid to form a torus.

We first study the utilization of idle machines and overhead associated with the *Spawn*. We then examine the use of funding as a priority mechanism, and investigate the resulting fairness of resource distribution. Finally, we explore the temporal and spatial dynamics of prices and funding, examining fluctuations, equilibria, and transients.

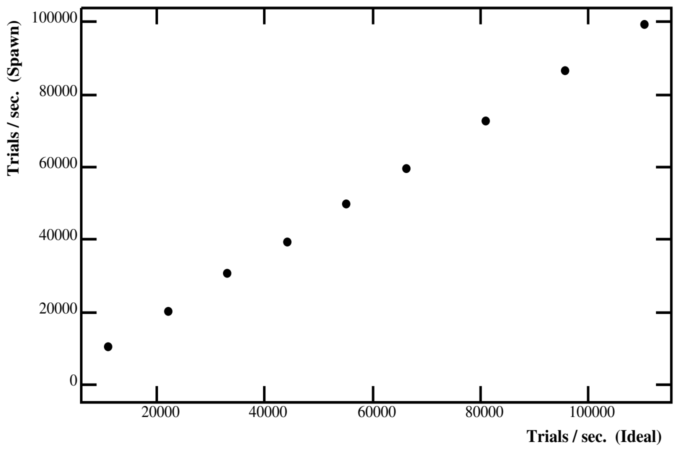

Figure 3: Basic Spawn Efficiency

Number of Monte-Carlo trials/sec. calculated by *Spawn* vs. ideal number of trials/sec. that could be performed concurrently on independent machines, for 1 to 9 machines. The time slices were all 60 seconds. The best linear fit indicates that *Spawn* operates with 89.7% efficiency (i.e. 10.3% overhead) for this particular task and granularity. When the time slices are doubled to 120 seconds, the overhead drops to 7.6%.

4.2 Use of Idle Machines

Our first experiment measured the efficiency with which idle machines are used in *Spawn*. To do this, we introduced one concurrent Monte-Carlo task into a network of otherwise idle machines. As this task executed, it quickly spawned subtasks into all of the available machines. Each application manager tried to spawn two submanagers and did not ask for extensions.

We then measured the number of Monte-Carlo trials executed per second as a function of the number of machines in the network. Figure 3 shows the resultant speeds compared to those which could have been ideally obtained by running independent serial Monte-Carlo tasks (containing no *Spawn*-related code) on the same set of machines.

Because of the negligible communication overhead in the Monte-Carlo application, we observe a linear relationship; the slope of approximately 0.9 essentially characterizes the efficiency of the *Spawn* system. Further experiments indicated that the bulk of the measured overhead could be attributed to extensive logging of runtime diagnostics, and the use of a single file-server from which executable files were loaded. Despite these influences and the need for optimizations, we were satisfied that an overhead of approximately 10% was adequate for our experimental purposes.

6 Nodes, Fully Connected	
Funding Ratio	*Time Allocation*
1:1	1.04:1
2:1	1.85:1
10:1	12.36:1
3:2:1	2.79:2.00:1

12 Node Torus	
Funding Ratio	*Time Allocation*
1:1:1	1.01:1.00:1
2:1	2.92:1
3:2:1	3.50:2.37:1

Figure 4: Fairness of Resource Allocation

Fairness of *Spawn* for various funding ratios among applications. All experimental runs were performed on Sun4/110's using 60 sec. time slices. The runs lasted between 20 and 30 minutes. The first set compares the allocated time to the funding ratio for six fully-connected machines. In these runs, the tasks declined extensions and children were funded equally. The second set is for twelve machines each connected to four others in a regular grid to form a torus. In this case the tasks accepted extensions and funded their two most cost-effective children.

4.3 Fairness of Resource Distribution

Conventional operating systems often employ a simple notion of *priority* in scheduling processes. A process with high priority is given absolute precedence over a process with lower priority, and the use of a processor is granted to the highest priority process that tries to obtain it. The ability to express priorities for processes affords programmers some measure of control over the management of computational resources. However, the conventional conception of priority in sequential systems does not scale well to concurrent and distributed systems. In fact, the very notion of priority is not well-defined for distributed systems, especially if these systems are composed of heterogeneous machines.

In *Spawn*, monetary funds abstractly encapsulate relative resource rights, and are analogous to priority. Funding units are abstract since they are completely independent of machine details. They are also relative, since the amount of a resource to which a task with a given amount of funding is entitled varies dynamically in proportion to the contention for that resource. This property permits concurrent *Spawn* applications to adaptively expand into more machines when prices are low, and forces them to contract into fewer machines when prices are high. Through our experiments, we have found that the use of funding as priority is highly effective, even in decentralized systems.

In order to test the distribution of resources when there are competing tasks executing in *Spawn*, we measured the resource utilization with multiple competing versions of the concurrent Monte-Carlo application, given various top-level funding ratios. Each application spawned a tree of subtasks, and each subtask simply bid all of its available funding on the auction with the earliest available next time-slice.

The results for a number of representative examples executed on the actual *Spawn* system are summarized in Figure 4. Simulation results for larger systems are tabulated in Figure 5. The *funding ratio* columns specify the relative top-level funding rates *given* to applications. Similarly, the *time allocation* columns indicate the relative amount of processing time *obtained* by each application. A *fair* resource distribution is one in which

	Time Allocation		
Funding Ratio	*16 Node Torus*	*36 Node Torus*	*64 Node Torus*
1:1:1	1.15:1.06:1	1.06:1.04:1	1.08:1.04:1
3:2:1	2.89:1.84:1	2.89:1.75:1	2.95:1.77:1
10:5:1	9.46:4.22:1	12.11:4.68:1	8.18:3.34:1

Figure 5: Fairness In Larger Systems
Fairness of simulated *Spawn* system for various funding ratios among applications. All experimental runs were simulated using 15 sec. time slices. The simulated run lengths were approximately 40 minutes. In all simulations, each machine was connected to four others in a regular grid to form a torus. Tasks accepted extensions and funded their two most cost-effective children.

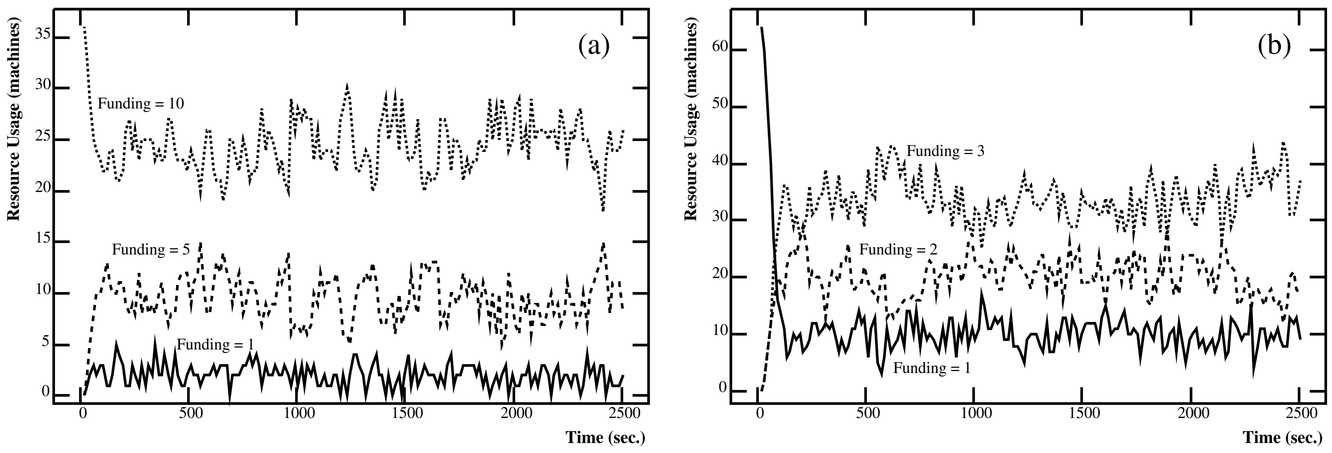

Figure 6: Continuous Fairness
Resource usage for three concurrent tasks in two simulated systems. The number of machines inhabited by each of the three tasks is plotted as a function of time. (a) Tasks were funded with a funding ratio of 10:5:1 in a 36 node system configured as a torus. The actual resource usage was observed to be 12.11:4.68:1, averaged over the entire run. (b) Tasks were funded with a funding ratio of 3:2:1 in a 64 node system configured as a torus. The actual resource usage was observed to be 2.95:1.77:1, averaged over the entire run.

each application is able to obtain a share of system resources that is close to its share of total system funding. As we can see from the tables, the auction mechanism allocates time in a manner that is reasonably close to the funding ratio in all runs. Moreover, this fairness of allocation is usually observed throughout the entire run. Figure 6 presents representative simulation runs which demonstrate that a reasonable degree of fairness is continuously maintained, even in large systems. We thus expect a proper response in more complex situations; for example, when tasks are continuously entering and leaving the system, or when tasks are funded dynamically based upon partial results. An example of this latter case is an application which provides tasks with most of their funds near the beginning of a computation to obtain a rough estimate, then pays less for further accuracy, as can be appropriate in Monte-Carlo calculations.

4.4 Market Dynamics

Since *Spawn* uses currency as its basic mechanism for resource allocation, we are interested in how close it comes to behaving like a market. Can a meaningful market price for processor time be established and sustained, even when relatively few machines and tasks are involved? If such a price can be established, does it respond in a reasonable and timely way to changes in the numbers of buyers and sellers? In this section, we investigate the existence of equilibrium prices, temporal and spatial fluctuations, and transients. We also examine the effects of heterogeneous machines on prices in *Spawn's* computational economy.

4.4.1 Equilibrium

The simplest case that we consider involves a fully-connected network of homogeneous machines (i.e., each machine is valued equally by the tasks). Each task placed a bid on the auction which was slated to finish first, and bid elsewhere only upon receiving a rejection notice. Each manager tried to spawn two submanagers, which were funded equally. Figure 7a shows how the price, averaged over all machines, changed as a function of time. As can be seen, a reasonable equilibrium was reached in the sense that the temporal fluctuations were small compared to the average. In equilibrium, the total rate at which currency enters the system (here \$0.06/sec.) ought to equal the rate at which the auctions collect revenue. Since there are six auctions, we expect the average price to be \$0.01/sec., which matches the measured average. The fluctuations in the average price (the standard deviation measured from $t = 400$ to $t = 1200$) were approximately 13%. A closer analysis also revealed that, within each of the six auctions, the fluctuations are in the vicinity of 25%.

Next we examined the behavior when the network was less densely connected, a characteristic of larger networks. When machines had few connections to one another, employing the simple equal-funding strategy used in the fully-connected case led to a significant difference in prices among the machines. Such price differences resulted when relatively wealthy tasks (close to the root managers of their sponsorship hierarchies) were spawned near one another, and their relatively poorer subtasks (deeper in the sponsorship hierarchies) also happened to be spawned near one another. Where there is a persistent price difference across machines, clever managers should be able to take advantage of the situation by spending their money on the less expensive ones. If several clever managers compete against one another, we would then expect the price differentials to disappear as the prices on the less expensive machines are bid up by the smart managers looking for a bargain.

To address the problem of price differentials, a slightly more sophisticated funding strategy was employed. Instead of funding each child equally, only those children running on the cheapest few machines were given the funding. This strategy eliminated the price difference; the resulting behavior is shown in Figures 7b

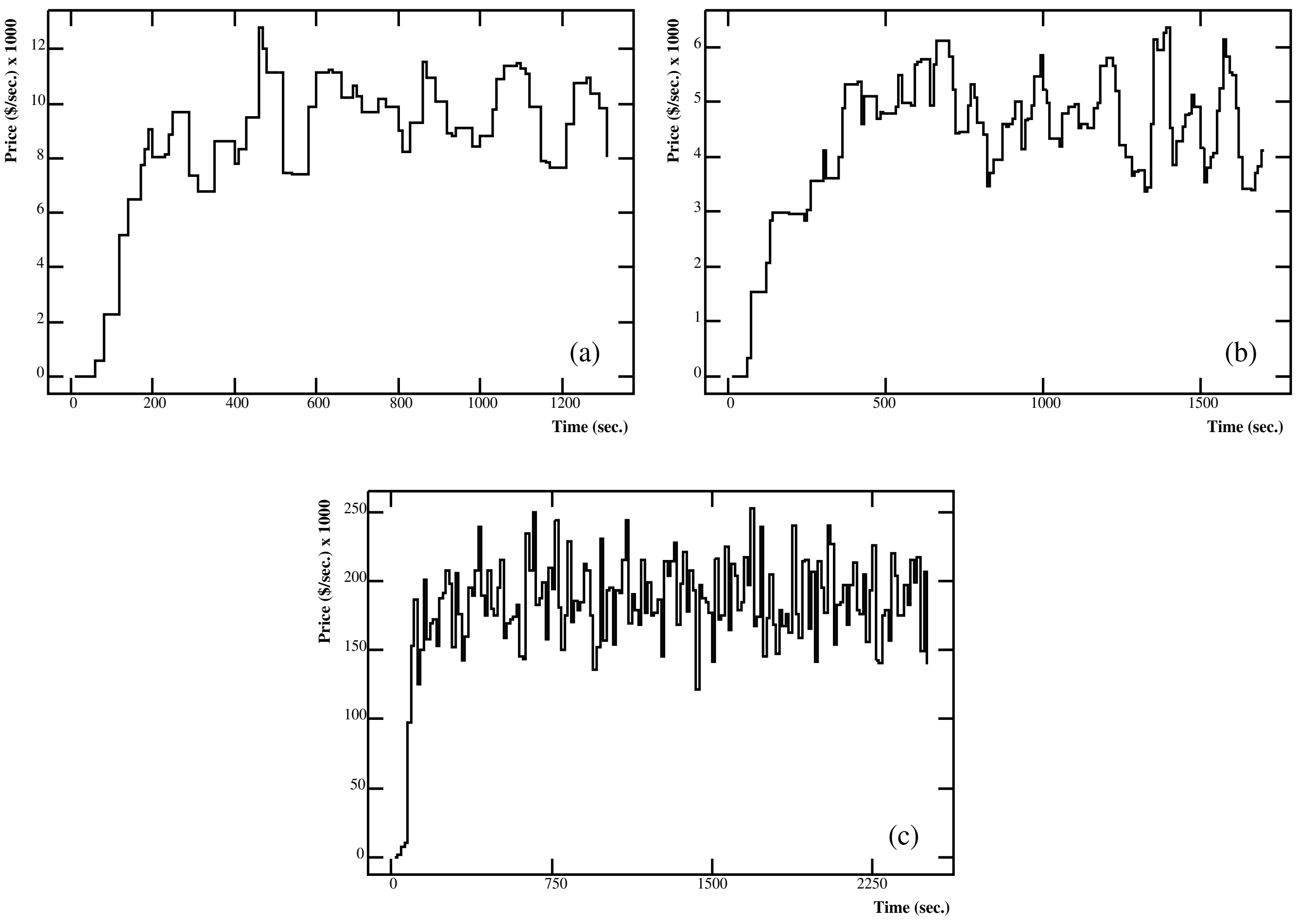

Figure 7: Price Equilibrium

Average price as a function of time. In each experiment, there are three concurrent tasks, with a funding ratio of 3:2:1.

(a) Six fully-connected machines. Time slices are 60 seconds; funding is supplied every 10 seconds. Tasks declined extensions and funded children equally. The total rate at which money enters the system is \$0.06/sec. Measuring the average price between $t = 400$ and $t = 1200$ sec., we find that the average price = \$0.0098/sec., with standard deviation \$0.0013/sec.

(b) Twelve locally-connected machines configured as a torus. Time slices are 60 seconds; funding is supplied every 10 seconds. Tasks accepted extensions and funded their two most cost-effective children. The total rate at which money enters the system is \$0.06/sec. Measuring the average price between $t = 400$ and $t = 1600$ sec., we find that the average price = \$0.00487/sec., with a standard deviation of \$0.0007/sec.

(c) Simulated 64 node system configured as a torus. Time slices are 15 seconds; funding is supplied every 5 seconds. Tasks accepted extensions and funded their two most cost-effective children. The total rate at which money enters the system is \$12/sec. Measuring the average price between $t = 150$ and $t = 2250$ sec., we find that the average price = \$0.1879/sec., with a standard deviation of \$0.0266/sec.

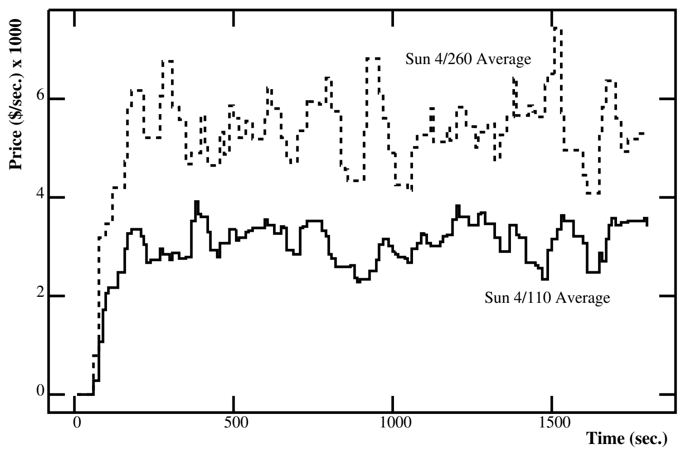

Figure 8: Price Differential in a Heterogeneous System
Price differentials in a nine node inhomogeneous system configured as a torus. The length of each time slice is 60 sec. Each of four roots is funded with \$0.10 every 10 seconds. The solid line shows the average price for six Sun4/110's; the dashed line indicates the price averaged over three Sun4/260's. The observed price ratio for the different types of machines is close to their relative worth to the applications.

and 7c. The ability to balance prices merely by changing an application's funding strategy illustrates the flexibility of the market mechanism. However, it can be argued that a more elegant solution would be to use nonlocal communications to initially distribute tasks more uniformly than was possible with nearest-neighbor connections in a logical torus topology. This issue will be discussed in section 5.

4.4.2 Price Differentials in Heterogeneous Systems

When the machines in the network are not homogeneous, price differentials develop between machines that reflect their relative values to applications. Figure 8 shows the prices in a system with 9 locally-connected auctions configured as a torus: 3 running on Sun4/260's and 6 on Sun4/110's. The applications hold a Sun4/260 to be 1.4 times a valuable as a Sun4/110. The factor of 1.4 is based entirely on the relative speed of the machines for the given application. Once the average prices reach equilibrium, they differ by a factor within 20% of 1.4: \$0.0032/sec. for the 110's and \$0.0053/sec. for the 260's.

4.4.3 Transients

In addition to displaying meaningful prices, the system should adapt to changes in demand. In order to determine how long it takes the average price to stabilize after a new task is added to the system, we first introduced two concurrent Monte-Carlo tasks, waited for the establishment of an equilibrium, and then injected a third task into the system.

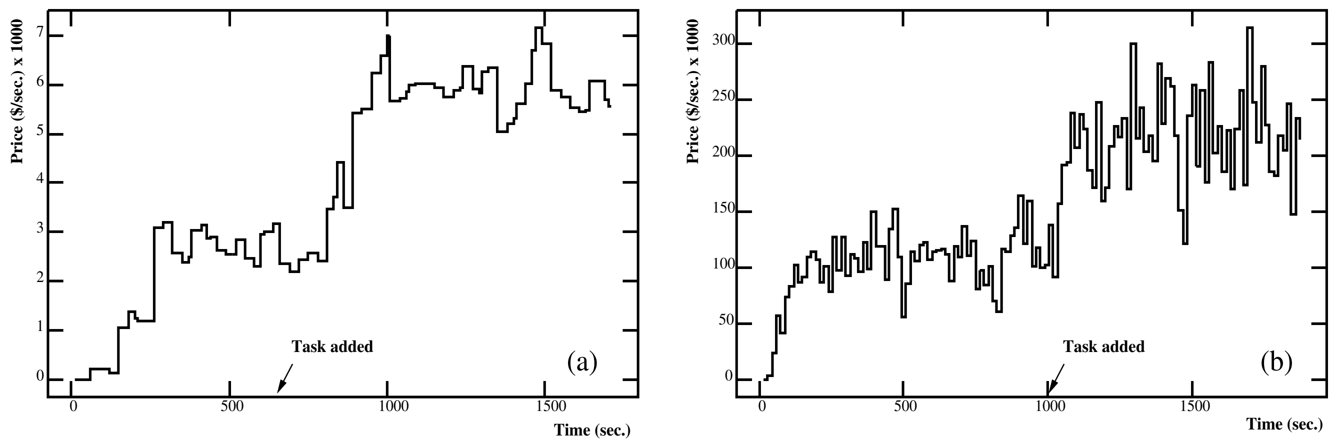

Figure 9: Market Price Adaptation

Average price as a function of time. In each experiment, two tasks with equal funding initially compete with one another, reaching an equilibrium. Later, at the point indicated on the time axis, a third task with twice the funding of each original task is added to the system.

(a) Six fully-connected machines. Time slices are 60 seconds; funding is supplied every 10 seconds. An initial equilibrium price of \$0.0033/sec. is reached, and the new task is added at t = 671 sec. It made its first successful bid 75 seconds later. After this point, the price quickly rose to a new equilibrium value of \$0.0067/sec.

(b) Simulated 36 node system configured as a torus. Time slices are 15 seconds; funding is supplied every 5 seconds. An initial equilibrium price of \$0.1101/sec. is reached, and the new task is added at t = 1000 sec. It made its first successful bid 8 seconds later. After this point, the price quickly rose to a new equilibrium value of \$0.2196/sec.

Figure 9a shows the resulting average price as a function of time in a fully-connected six node system. Before the addition of the third task, the observed equilibrium price is close the the theoretical value (i.e., the rate at which currency enters the system). After introducing the additional task, the average price adjusts within a few auction cycles to the new equilibrium value, a sign of the adaptability of the system when there are multiple tasks competing for processing resources. Note that this transient time is the same as when starting from an initially idle network (compare with Figure 7a).

A similar experiment was repeated for a larger, locally-connected system in a simulation consisting of 36 nodes configured as a torus. Figure 9b shows the resulting average price as a function of time.

4.4.4 High Priority Tasks

Systems that schedule tasks according to fixed priorities enable high-priority tasks to run immediately. *Spawn's* market mechanisms can also give some tasks high priority, where higher funding corresponds to a higher priority. To demonstrate this, we introduced a high-priority task into a system in which low-priority

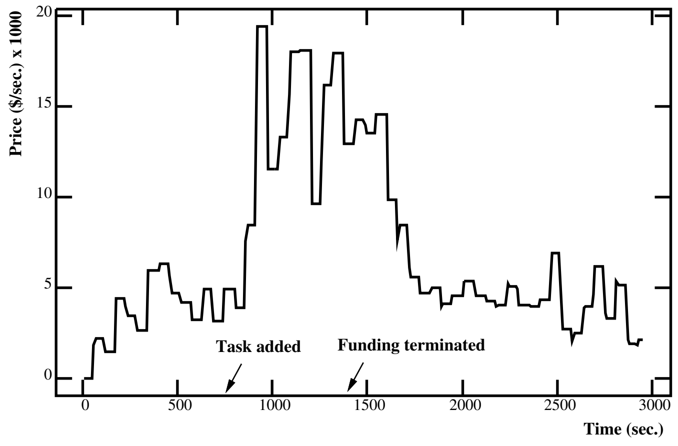

Figure 10: Market Response to a High Priority Task
Average price as a function of time when a high priority task is injected into a 6 node, fully-connected system with low-priority jobs. The introduction and termination of the high priority task's funding are indicated on the time axis.

tasks were running; Figure 10 illustrates the system's response. The experimental market initially consisted of two tasks continuously funded at rates of \$0.01/sec. and \$0.02/sec. After the initial transients subsided, a new task was injected at $t = 700$ into the market with the substantially higher funding rate of \$0.07/sec. and a fixed total allocation of \$50.40.

The high priority task made its first successful bid at $t = 855$, roughly two auction time-slices after of its creation. It then rapidly took over most of the available resources. From this time until $t = 1600$, shortly after its funding terminated at $t = 1400$, the high priority task captured approximately 60% of the total system resources. This is in close agreement with the "fair" value of 70%, since the three competing tasks were funded in a 7:2:1 ratio over that time interval. This confirms the responsiveness of the system to sudden changes in demand. Given the previous results on fairness of resource allocation and the fairly short observed transients, this is not a surprising result.

4.4.5 Auction Revenue Distribution

Our final set of *Spawn* experiments examined the *spatial* distribution of prices across all of the machines in a homogeneous system. Since the income earned by an auction captures the price history for a node, we use the distribution of auction revenue as a measure of spatial equilibrium.

Let $\mathcal{R}$ denote the set of cumulative auction revenues. Figure 11 plots the ratio of the standard deviation of $\mathcal{R}$ to its average as a function of time. This illustrates the relative magnitude of fluctuations in auction revenue over time.

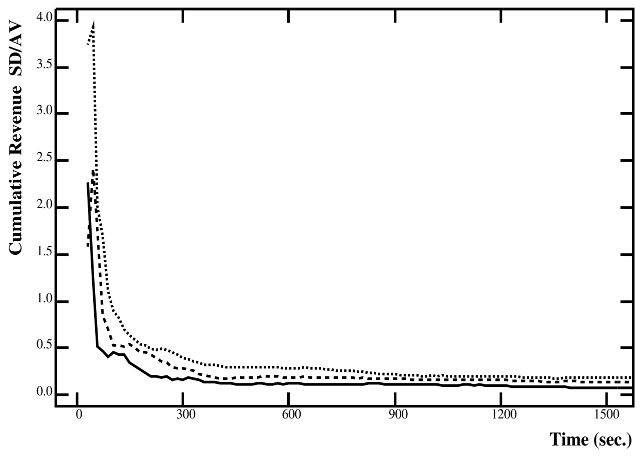

Figure 11: Relative Imbalance in Auction Revenues
Relative fluctuations in cumulative auction revenue distributions as a function of time. The vertical axis measures the ratio of the standard deviation to the average of cumulative revenues across all auctions. The solid line presents the data for a simulated 16 node system configured as a torus. The dashed line is for a simulated 36 node torus, and the dotted line plots the simulation results for a 64 node torus. In all of the simulations, time slice lengths were 15 seconds, and three concurrent tasks were executed with a 3:2:1 funding ratio.

The auction revenues are initially very poorly balanced. However, the balance quickly improves over time until the magnitude of the fluctuations in revenue are reasonably small when compared with the average revenue. Further experiments indicate that this effect remains essentially the same when different times are chosen as the reference point from which auction revenues are accumulated. This indicates that although prices may be poorly balanced at any particular instant, the cumulative balance rapidly improves over time.

The time necessary to achieve a reasonable balance increases with network size. For a simulated 16 node system configured as a torus, the fluctuations (as measured by the standard deviation) drop to approximately 13% of the average within 25 auction time slices. A 36 node system requires 30 auction time slices to decrease fluctuations to 18% of the average, and a 64 node system needs 50 time slices for fluctuations of approximately 25%.

5 Limitations and Lessons

Experience with *Spawn* has taught us a number of valuable lessons about the design and implementation of computational markets. In this section, we present some of these lessons and discuss related limitations in the *Spawn* system.

5.1 Scaling to Larger Networks

A desirable characteristic of any distributed system is the ability to scale well to large networks. The experiments and simulation results presented in section 4 indicate that with respect to fairness of resource distribution and temporal price dynamics, *Spawn* scales gracefully to large systems. However, in terms of spatial price dynamics, the scaling results for *Spawn* are less favorable. This is indicated by Figure 11, and is also suggested by the need to employ an improved manager strategy to avoid price differentials, as described in section 4.4.

We believe that the underlying cause for the observed spatial price fluctuations and differentials is the limited communication of information permitted in *Spawn* systems configured as a torus with only nearest-neighbor connections. Nearest-neighbor connections were also used in [10], but since all reported experiments were performed on a small, nine node simulated system, these problems were not encountered. The constraint of nearest-neighbor communication limits the rate at which tasks can discover and exploit favorable situations that exist in distant regions of large networks.

Thus, for improved performance in computational markets, scalable propagation of information is needed. Several effective methods exist that could be used in future systems. One technique is to communicate with a small number of nodes chosen at random [2]. Another promising option is to propagate information using a hierarchy of information flow rates [23].

5.2 Auctions and Bidding

As explained in section 3, *Spawn* uses a sealed-bid, second-price auction mechanism to determine the price for a time slice of a processing resource. This type of auction is very effective in human markets, and has been proposed as a resource allocation mechanism in work on computational markets [6, 10]. Nevertheless, the dynamics of such auctions have not been analyzed carefully.

In this section we examine the dynamical behavior of a single *iterated* sealed-bid, second-price auction. In particular, we look at the behavior of such an auction when the bidders increase their bids linearly over time. This situation is characteristic of the *escalator algorithm* proposed for single-processor scheduling in

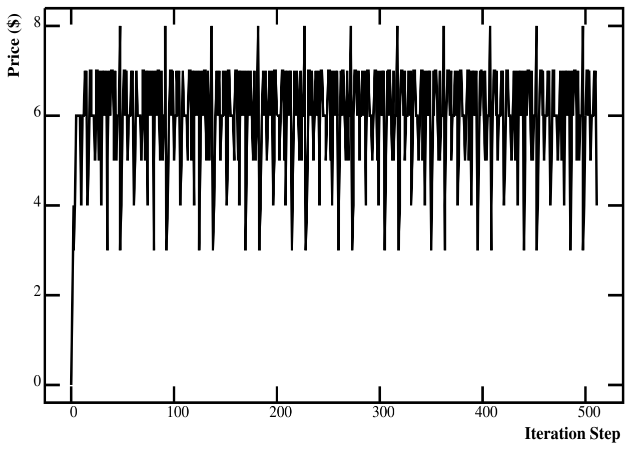

Figure 12: Price for a Single Iterated Auction
Price as a function of time (iteration step) for a single iterated sealed-bid, second-price auction. Three bidding tasks were given funding rates of \$1, \$2, and \$3 per auction.

[6], and the state of an individual auction in a *Spawn* economy receiving bids from tasks with linear funding rates.

Figure 12 plots the selling price for a series of iterated sealed-bid second-price auctions when there are three bidders with funding rates of \$1, \$2, and \$3 per auction. The oscillatory price behavior and volatility are rather striking. Over a sufficiently long time interval, the average price is \$6 per auction, with a standard deviation of \$1.14. However, the price continues to oscillate between a low of \$3 and a high of \$8 per auction, with most prices falling in the \$5 to \$7 range. Further experiments indicate that the price fluctuations were even sensitive to the order in which ties were broken – causing the standard deviation of long runs to vary between \$0.82 and \$1.14 per auction.

Another metric that warrants investigation is fairness. Figure 13 shows the fraction of auctions won by each of the three bidders under the conditions described above. Given enough time, resource allocations converge near their expected "fair" values. In this case, we expect a ratio of 3:2:1, and we observe an allocation of 3.43:2.00:1.00. However, over a shorter interval there is considerable volatility in the observed ratios.

In addition to volatility, auctions exhibit another undesirable property in computational markets. Bidding requires all interested tasks to explicitly communicate with a resource before any decisions are made; the resource must then inform all bidders of the acceptance or rejection of their offers. The overhead imposed may be negligible in a distributed system such as *Spawn*, but becomes increasingly important as the granularity of tasks decreases. One possibility is to use a more centralized market instead of separate markets on each machine. This would help buyers and sellers locate one another, yet no single agent would be required to

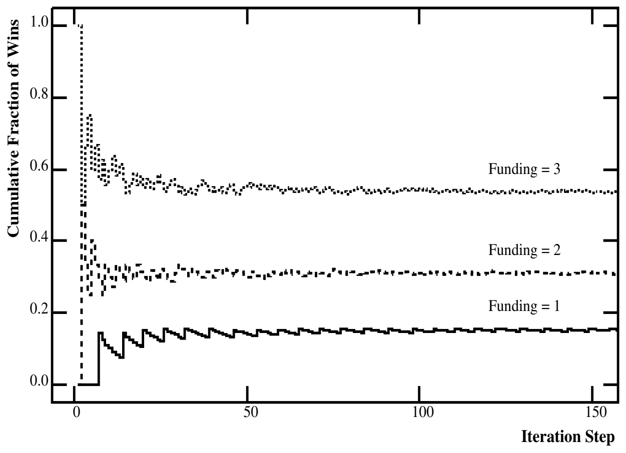

Figure 13: Fairness for a Single Iterated Auction

Cumulative fraction of auctions won by each of three tasks with a funding ratio of 3:2:1 in a single iterated auction. The actual allocation achieved was 3.43:2:1, averaged over the entire run.

coordinate information regarding the preferences and capabilities of all agents, as would be required by a truly centralized controller.

These observations have prompted us to explore less volatile, low-overhead mechanisms for determining resource prices. A promising approach was conceived after examining a number of graphs similar to Figures 12 and 13. The basic insight is that the average price in an iterated second-price auction converges to the sum of the funding rates of all bidders. Instead of relying upon an iterated auction, a resource can be *directly* allocated (e.g. time multiplexed at a very fine granularity) among the tasks competing to use it, in proportion to their funding rates. This directly achieves fairness while maintaining a stable price for the resource that is proportional to demand. This type of direct mechanism is at the core of a new market-like system for allocating resources to fine-grained tasks in multiprocessors [38].

6 Conclusions

6.1 Summary of Present Work

We have described the architecture, implementation, and testing of *Spawn*, a distributed computational system that shares many properties with human markets and auctions. *Spawn* addresses the problems of resource contention, fair dynamic load sharing, resource management for concurrent computations, and the notion of priority in distributed systems. As the experiments show, the system can successfully handle these problems without resorting to global controls.

As a practical system, *Spawn* efficiently harnesses otherwise-wasted idle time in a distributed network of heterogeneous workstations. Thus, distributed computations competing for resources can be efficiently managed with acceptable overhead. This allows certain classes of large, easily-parallelized computations that are commonly run on supercomputers to execute on existing, underutilized networks.

In addition to its successful performance, *Spawn* has enabled many quantitative experiments that probe the dynamics of real computational markets. In particular, we have found that a small number of agents can produce an identifiable market, since fluctuations were not able to obscure the stable equilibria in prices. Moreover, we have observed that monetary funding can be used as an effective form of priority in distributed systems containing heterogeneous nodes. Experiments have also demonstrated that price information can be used to adaptively control the expansion and contraction of process trees in concurrent applications. Finally, we have examined problems with auction mechanisms in computational economies, and propose the use of a simple sharing rule as an alternative.

6.2 Directions for Future Research

As an experimental research tool, *Spawn* is somewhat fragile and awkward to use. One research area that warrants considerable attention is the smooth integration of *Spawn's* mechanisms with a robust language for parallel and distributed computation. Simple, expressive linguistic mechanisms would facilitate more rapid advancement by researchers concerned with market-based computational resource management. The *priority flow* framework is directed at fulfilling this need [38]. It facilitates the expression of abstract resource management schemes in parallel systems, using a low-overhead, fine-grained market-like substrate. The *priority flow* framework also uses a simple sharing rule in place of auctions to determine prices, and incorporates a scalable, hierarchical technique for propagating information.

Another area of interest is the specification and implementation of more sophisticated funding strategies for controlling concurrent computations. Although we have demonstrated the effectiveness of a small number of simple funding strategies, the general topic has been largely unexplored. It may be possible to develop automated tools to assist in the development of novel funding algorithms. For example, software similar to a "trace scheduler" [8] capable of monitoring dynamic resource usage patterns in a concurrent computation may prove valuable for improving or evolving funding strategies for a given application. The applicability and interaction of techniques such as static program analysis and adaptive algorithms in the context of market-based resource allocation remains an open question.

A final area for further study is diversity in computational economies. Although we have focused on the purchase of processor time, we should point out that the price mechanism can allow machines with different capabilities (floating point hardware, large disc space, direct access to special databases or proprietary algorithms, etc.) to have different values. Thus, tasks can flexibly devote their currency to the resources most important to them. Such a scenario would bring *Spawn* into greater correspondence with a real economy, in which there is a multitude of different goods. By the same token, the market mechanism supports a deeper symmetry than studied here, in which computational results obtained by agents can themselves become marketable goods of potential use to other agents [27]. In this way, one can envision a more cooperative collection of processes, which, despite their different goals and characteristics, can contribute to each other's performance.

References

[1] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, 1986.

[2] A. Barak and A. Shiloh. A distributed load-balancing policy for a multicomputer. *Software Practice and Experience*, pages 901–913, September 1985.

[3] T. C. K. Chow and J. A. Abraham. Load balancing in distributed systems. *IEEE Transactions on Software Engineering*, pages 401–412, July 1982.

[4] Edward G. Coffman and Peter J. Denning. *Operating Systems Theory.* Prentice Hall, Englewood Cliffs, NJ, 1973.

[5] Randall Davis and Reid G. Smith. Negotiation as a metaphor for distributed problem solving. *Artificial Intelligence*, 20:63–109, 1983.

[6] K. Eric Drexler and Mark S. Miller. Incentive engineering for computational resource management. In B. A. Huberman, editor, *The Ecology of Computation*, pages 231–266. North-Holland, Amsterdam, 1988.

[7] Alex Dupuy. Network Simulation Tested (nest) User Manual. Technical Report CS-NEST, Columbia University, Computer Science Department, 1986.

[8] J. R. Ellis. *Bulldog: A Compiler for VLIW Architectures.* PhD thesis, Yale University, February 1985. YALEU/DCS/RR-364.

[9] Richard Engelbrecht-Wiggans, Martin Shubik, and Robert M. Stark. *Auctions, Bidding, and Contracting: Uses and Theory.* New York University Press, New York, NY, 1983.

[10] Donald Ferguson, Yechiam Yemini, and Christos Nikolaou. Microeconomic algorithms for load balancing in distributed computer systems. In *International Conference on Distributed Computer Systems*, pages 491–499. IEEE, 1988.

[11] Daniel Friedman. On the efficiency of experimental double auction markets. *American Economic Review*, 24(1):60–72, March 1984.

[12] Max Hailperin. Load balancing for massively-parallel soft-real-time systems. Knowledge Systems Laboratory Report KSL-88-62, Dept. of Computer Science, Stanford University, August 1988.

[13] Carl Hewitt. The challenge of open systems. *Byte*, 10:223–242, April 1985.

[14] Bernardo A. Huberman and Tad Hogg. The behavior of computational ecologies. In B. A. Huberman, editor, *The Ecology of Computation*, pages 77–115. North-Holland, Amsterdam, 1988.

[15] Kenneth M. Kahn and Mark S. Miller. Language design and open systems. In Bernardo A. Huberman, editor, *The Ecology of Computation*, pages 291–313. North-Holland, Amsterdam, 1988.

[16] Kenneth M. Kahn and Vijay A. Saraswat. Money as a concurrent logic program. Technical report, Xerox PARC, 1989.

[17] Jeffrey O. Kephart, Tad Hogg, and Bernardo A. Huberman. Dynamics of computational ecosystems. *Physical Review A*, 40:404–421, 1989.

[18] Douglas B. Lenat. The role of heuristics in learning by discovery: Three case studies. In R. S. Michalski et. al., editor, *Machine Learning: An Artificial Intelligence Approach*, pages 243–306. Tioga, Palo Alto, CA, 1983.

[19] Barbara Liskov and Robert Scheifler. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, July 1983.

[20] Michael J. Litzkow, Miron Levy, and Matt W. Mutka. Condor – A hunter of idle workstations. In *International Conference on Distributed Computer Systems*, pages 104–111. IEEE, 1988.

[21] Virginia M. Lo. Heuristic algorithms for task assignment in distributed systems. In *International Conference on Distributed Computing Systems*, pages 30–39. IEEE, 1984.

[22] Virginia M. Lo and David Chen. Intelligent scheduling in distributed computing systems. Technical Report CIS-86-14, Dept. of Computer and Information Science, University of Oregon, April 1987.

[23] E. Lumer and B.A. Huberman. Dynamics of resource allocation in distributed systems. Technical report, Xerox Palo Alto Research Center, March 1990.

[24] T. W. Malone, R. E. Fikes, K. R. Grant, and M. T. Howard. Enterprise: A market-like task scheduler for distributed computing environments. In B. A. Huberman, editor, *The Ecology of Computation*, pages 177–205. North-Holland, Amsterdam, 1988.

[25] Carl R. Manning. Acore: An actor core language. MPSG Apiary Design Note 7, MIT AI Laboratory, August 1987.

[26] Mark S. Miller, Daniel G. Bobrow, Eric Dean Tribble, and Jacob Levy. Logical secrets. In Ehud Shapiro, editor, *Concurrent Prolog: Collected Papers.* MIT Press, Cambridge, MA, 1987.

[27] Mark S. Miller and K. Eric Drexler. Markets and computation: Agoric open systems. In B. A. Huberman, editor, *The Ecology of Computation*, pages 133–176. North-Holland, Amsterdam, 1988.

[28] Matt W. Mutka and Miron Levy. Scheduling remote processing capacity in a workstation-processor bank network. In *International Conference on Distributed Computer Systems*, pages 2–9. IEEE, 1987.

[29] Joseph Carlo Pasquale. *Intelligent Decentralized Control in Large Distributed Computer Systems.* PhD thesis, University of California, Berkeley, April 1988.

[30] James L. Peterson and Abraham Silberschatz. *Operating Systems Concepts.* Addison-Wesley, Reading, MA, 2nd edition, 1985.

[31] John F. Shoch and Jon A. Hupp. The "Worm" programs – Early experience with a distributed computation. *Communications of the ACM*, 25(3):172–180, March 1982.

[32] Reid G. Smith. The Contract Net protocol: High-level communication and control in a distributed problem solver. *IEEE Transactions on Computers*, C-29(12), December 1980.

[33] I. M. Sobol. *The Monte Carlo Method.* Mir Publishers, Moscow, 1975.

[34] I. E. Sutherland. A futures market in computer time. *Communications of the ACM*, 11(6):449–451, June 1968.

[35] Andrew S. Tanenbaum and Robert Van Renesse. Distributed operating systems. *ACM Computing Surveys*, 17(4):419–470, December 1985.

[36] Daniel G. Theriault. Issues in the design and implementation of Act 2. Technical report, MIT AI Laboratory AI-TR-728, 1983.

[37] Carl A. Waldspurger. A distributed computational economy for utilizing idle resources. Master's thesis, Massachusetts Institute of Technology, May 1989.

[38] Carl A. Waldspurger. Priority Flow: A framework for abstract, adaptive resource management. MIT LCS Parallel Software Group, Internal Memo (unpublished), May 1990.