A Distributed Computational Economy For Utilizing Idle Resources

by

Carl A. Waldspurger

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Computer Science and Engineering
and
Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

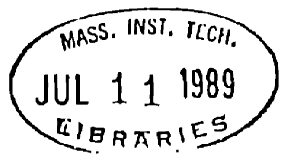
May 1989

Signature of Author_____
Department of Electrical Engineering and Computer Science
May 12, 1989

Certified by_____
Bernardo Huberman
Research Fellow, Xerox Palo Alto Research Center
Company Supervisor

Certified by_____
Carl E. Hewitt
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by_____
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students



ARCHIVES

A Distributed Computational Economy For Utilizing Idle Resources

by

Carl A. Waldspurger

Submitted to the Department of Electrical Engineering and Computer Science
on May 12, 1989, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Computer Science and Engineering
and
Master of Science in Electrical Engineering and Computer Science

Abstract

This thesis presents the design and implementation of 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-grained concurrent applications and the remote execution of many unrelated tasks. Using concurrent Monte-Carlo simulations as prototypical applications, the thesis explores issues of scaling, the fairness of resource distribution, price equilibria, and the dynamics of transients. In addition to serving the practical goal of harnessing processor idle-time in a computer network, *Spawn* has proven to be a valuable experimental workbench for studying the dynamics of computational markets and distributed systems with no global controls.

Company Supervisor: Bernardo Huberman
Title: Research Fellow, Xerox Palo Alto Research Center

Thesis Supervisor: Carl E. Hewitt
Title: Professor of Computer Science and Engineering

Acknowledgments

First, I would like to thank Bernardo Huberman, my thesis supervisor at Xerox PARC. I am grateful to Bernardo for his constant enthusiasm, guidance, and encouragement. Working with Bernardo has taught me many valuable lessons about research, especially the importance of grounding abstract ideas with empirical results.

I am also indebted to the rest of the Dynamics of Computation group at PARC. Tad Hogg, Jeff Kephart, and Scott Stornetta were actively involved in every phase of the *Spawn* project. Their enthusiasm and insight is matched only by their kindness; they were truly a wonderful group of people to work with. Numerous group discussions helped to define the problems and shape the ideas presented in this thesis.

Special thanks to Carl Hewitt, my thesis advisor at MIT. Through coursework, research projects, and discussions, he has helped shape my development and research interests in computer science. His comments on several drafts of this thesis were invaluable.

I thank Dennis Arnon, Rick Beach, and Randy Smith, earlier supervisors during my years as a PARC research intern, for providing me with support, encouragement, and interesting projects as I gradually learned what doing research was all about. I am grateful to them and to everyone else at PARC who has made my time there an incredible experience.

I thank the other VI-A students who made my time in Palo Alto more enjoyable, especially Dave Duis, John Eisenman, Dave Goldstone, Aaron Goodisman, Amy Lim, Jim Rauen, and Luis Rodriguez. Special thanks to Baldo Faieta, whose clever ideas, creative antics, and unusual schedule made him the perfect officemate.

Thanks to all those who have read this thesis and provided helpful comments: Wilson Hsieh, Sanjay Ghemawat, Bob Gruber, and Paul Wang.

I thank my friends at MIT, especially Bob Gruber, Paige Parsons, and Phil Pickens, for making the good times better and the bad times bearable. Their friendship, humor, and company has kept me sane.

I also thank the gang at LCS who put up with me while I wrote this thesis: Wilson Hsieh, Sharon Perl, Boaz Ben-Zvi, Jeff Cohen, Sanjay Ghemawat, Paul Wang, and Debbie Hwang. Their humor, kindness, and help has been much appreciated.

Thanks to Mark Syms, Art Burke, Jeff Clarke, Joe Liebrandt, and Nadine Kowalsky, who have been good friends since high school.

Thanks to Lianna Cleland for her friendship and the many wonderful times that we have spent together. She has made considerable progress in teaching me how to relax and enjoy life more.

I thank my brothers, Roy and Scott, for being themselves.

Finally, I thank my parents for all of their love and support. Without their countless sacrifices and limitless encouragement, none of this would have been possible. They have all of my love and respect.

Contents

List of Figures

Chapter 1

Introduction

Recent advances in computational technology have led to a proliferation of powerful networked computers that form large distributed systems. These systems can be applied to a wide variety of tasks, such as distributed problem-solving, interacting with the physical world, and interpreting real-time multi-sensor data. Such tasks generally require adaptability to unexpected events. They must be capable of coping with imperfect and conflicting information from many sources, and acting before all relevant information is available. Imprecise and inconsistent information can arise not only from hardware limitations but also from computations using probabilistic methods, heuristics, rules with many exceptions, or learning resulting in overgeneralization. Similarly, delays in receiving needed information can be due to the time required to fully interpret signals in addition to physical communication delays.

A major difficulty in fully utilizing such systems is managing the complexity of the required software, especially the need to coordinate many tasks on multiple processors. As larger parallel machines and networks are developed, simple centralized allocation of tasks becomes increasingly ineffective. This is due to the fact that data in the world is distributed and rapidly changing. A central controller cannot access all of the information needed to effectively plan detailed behavior, even if sufficient computational power is available. Instead, reliability and rapid response to local

changes require autonomous, local control of resources. We are thus confronted with the challenge of developing schemes for decentralized resource allocation.

The emergence of large, decentralized systems raises some fundamental questions [Hew85, Hew86]. Basically, there is a need for a general theoretical 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 a 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 this collection of interacting processes as *computational ecosystems.* This form of computation is also called *open* because the closed-world assumption of traditional computer science is no longer valid [Hew85].

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 prices and auctions greatly facilitate resource management in human societies, one might expect them to be similarly useful in computer networks – a proposal which has beer elaborated in some detail [Mil88]. 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.

To date, there have been a number of computational schemes developed with this analogy in mind. Recent work on microeconomic algorithms [Fer88] has determined that they can achieve globally effective allocation of CPU time which is comparable, and in some cases superior, to traditional algorithms. Moreover, such decentralized economic algorithms are often more modular and less complex than conventional load-balancing techniques. An earlier system, named *Enterprise*, also incorporated algorithms motivated by economics. Enterprise [Mal88] is a particularly interesting market-like scheduler that uses a bidding mechanism to allocate independent tasks at

run-time among remote, idle processors. However, Enterprise employed a scheduling priority scheme rather than a price mechanism, making it impossible to assess the efficacy of using a uniform medium of exchange that allows agents to reason independently about their strategies. Also, the absence of prices made it difficult to study the dynamics of the system – in particular, whether or not the system was in equilibrium. Finally, the absence of modern hardware support for multi-processing (e.g., separate per-process address spaces) made the system highly vulnerable to changes, intentional or unintentional, that agents produced in the state of the machines.

Despite the intuitive appeal of the economic approach to open computational systems, few empirical or quantitative theoretical 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. Preferences are often non-transitive or based upon considerations other than those believed to be the relevant performance criteria. 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 [Kep89]. Moreover, decisions made by computers can take place in times which are orders of magnitude faster than those of human decisions. Recent work [Kep88, Hub88, Kep89] 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 open systems. Even if these issues are resolved, there remain the practical questions of how many computational processes are needed in order to exhibit meaningful market-like behavior and how to exploit knowledge of their behavior to efficiently manage them.

In order to understand the behavior of a computational economy, we designed *Spawn*, an open, market-based computational system that runs in a distributed net-

work of heterogeneous high-performance computer workstations. It provided us with a chance to study such a system in actual use, thereby allowing us to guide its design and reformulate our notions of appropriate performance criteria in response to our observations. Having noted that a significant amount of computer equipment sits idle for large fractions of every day, we designed *Spawn* to allow users with large processing requirements to effectively tap these otherwise wasted resources. If we view such an environment as a large multiprocessor instead of a collection of independent networked workstations, *Spawn* supports both coarse-grained concurrent applications (parallel processing) and the remote execution of many unrelated tasks (multi-tasking).

We have implemented useful *Spawn* applications in both of these categories, primarily in the realm of concurrent Monte-Carlo simulations and remote document formatting. Large-scale concurrent Monte-Carlo simulations are a significant class of applications which can make use of idle CPU time. These simulations are frequently used to understand the behavior of systems with many degrees of freedom, such as matter in its 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 does it allow 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.

In the next chapter, we describe the basic mechanisms underlying the implementation of *Spawn*. In chapter 3 we present and discuss the results of a number of quantitative experiments in which the system parameters were varied in order to assess their influence on its overall behavior. These experiments were supplemented by simulations of *Spawn* which provide insight into how its behavior scales for large net-

works. Chapter 4 examines work related to this thesis in greater detail than the brief references made in this section. Finally, we conclude that market-like resource allocation methods based on a price mechanism can work effectively in a real distributed system, and highlight opportunities for further research. Two appendices are provided. Appendix A contains the specifications for the *Spawn* library routines, and Appendix B presents a detailed description of a sample distributed *Spawn* application.

Chapter 2

The Spawn System

2.1 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. This chapter describes the main features of this system.

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 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 can execute an *auction* process to manage the sale of his workstation's computational resources, and a buyer may execute an *application* that manages the purchase and use of computer processing resources.

Spawn was designed to support three kinds of tasks. The first consists of existing applications which cannot be modified for use with *Spawn*. An example of such *black*

box applications is a document formatter. The second class, *homogeneous* tasks, consist of arbitrarily many identical subtasks with limited and uniform communication among the components. Monte-Carlo calculations are a prototypical example. The third, and most general, category consists of *heterogeneous* tasks in which the degree of parallelism and nature of the communication are completely controlled by the application.

2.2 System Processes: Auctions and Resource Managers

The sale of resources is handled by a set of system processes on each machine, as shown in Figure 2-1. An *auction* process controls the sale of an idle workstation's resources, and handles messages from processes wishing to purchase slices of its time. 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 values of parameters set by the seller who initiated it. These parameters include the minimum and maximum allowable time slice lengths that can be sold, and a function which expresses the auction's utility function 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 client processes to terminate.

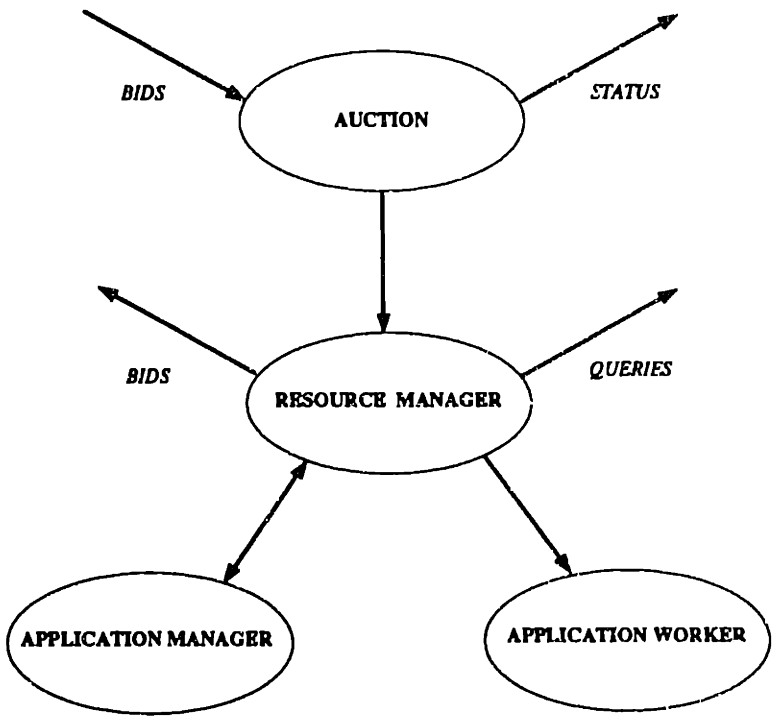


Figure 2-1: Selling Time

The auction process handles bids and status queries. When the auction accepts a bid, it instructs the resource manager to execute the corresponding application manager. The application can then arrange to spawn workers and submanagers via the resource manager.

It is worth noting that our auctions operate as *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 [Fri84]. 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]

Associated with each auction is a *resource manager* process. The resource manager serves as an interface between high-level applications and the rest of the *Spawn* system. In addition, the resource manager is responsible for initiating and monitoring the execution of the application process that purchased the current slice of processor time.

A high-level user application should not be encumbered with decision-making concerning the 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 its funds to allow for more sophisticated strategies. The *Spawn* architecture provides a uniform mechanism with these capabilities. The resource manager process encapsulates low-

[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 Section 2.3.

[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.

level details about auctions and bidding procedures, and communicates with a set of nearby auction processes. Each resource manager maintains a list of "neighboring" auctions which supply information about their current selling prices. This list is easily modified, and has allowed us to experiment with a variety of 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.

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. That is, the currently-executing application is allowed to continue its execution as long as it can pay the going market price for extensions. 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 and to cope with failure due to aborted computations.

2.3 Application Processes: Managers and Workers

Applications are divided into manager and worker modules. An *application worker* is the primary computational task for which resources have been allocated. It is essentially a black box application, except that it may communicate to its manager. Each application worker has a corresponding manager process. An *application manager* coordinates the execution of some task in a distributed application. It arranges, via communication with the resource manager interface, to spawn child workers and sub-

[4]These constraints are imposed by the programming environment which we used to implement *Spawn*. A detailed discussion of related implementation issues can be found in Section 2.5.

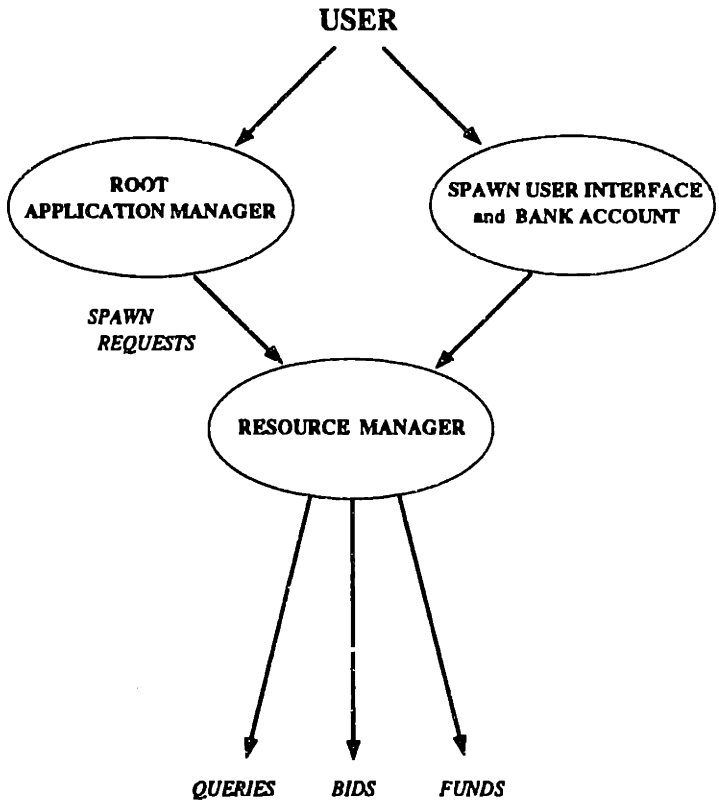


Figure 2-2: Buying Time

The user invokes a root application manager which registers itself with *Spawn*. The application can be controlled by interacting with the root or directly by issuing commands to the Spawn system's user interface. The resource manager interacts with the rest of the system to provide the required services.

managers 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 a 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 amount of 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 which does not directly interact with the *Spawn* system. Its corresponding application manager is a simple process which captures any output generated by the worker and sends it back to the root for display on the user's personal workstation.

A more typical application worker performs intensive computation and periodically reports partial results or error conditions to its immediate manager. This manager then combines and processes incoming partial results and sends aggregate results up to the next higher level in the management tree. If partial results can be combined at each level of management, such progress 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 2-3.

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, a list of task arguments, and a network directory pathname. The use of abstract task names provides a level of indirection which 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 maps particular workstation configurations onto the binary executable files which 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 computational 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 a spawn-worker message, as well as several parameters that the resource manager needs in order to find resources for the task's execution. These

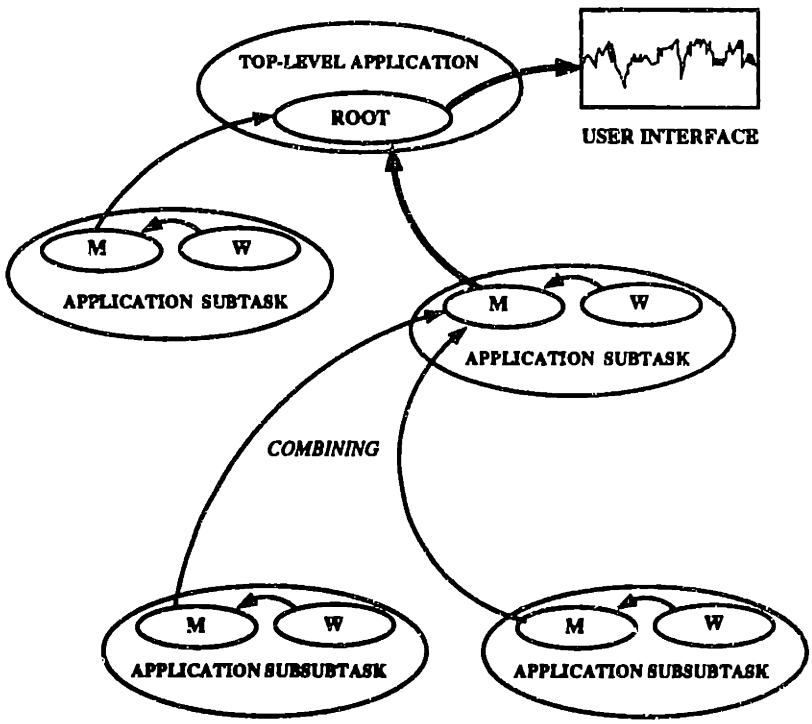


Figure 2-3: Application Reports

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.

include a specification of the relative importance of service time versus price, and the estimated processing time for the task (normalized to a machine with an application-specific rating of unity). 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 processes 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 that they contain to deliver aggregate reports to higher management. Of course this feature need not be used; for example, black box applications never send messages to their managers.

2.4 Sponsors and Funding

The concept of *sponsored* computations was pioneered by researchers who were concerned with the development of linguistic constructs for the allocation of resources in actor programming systems [The83, Man87]. *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 percentage of funding allocated to each child. A simple manager strategy would be to allocate equal amounts of funding to each child; more sophisticated strategies may take the relative progress achieved by each child into account. In this way, *Spawn* provides basic support for the use of heuristics similar to the "interestingness" metric employed by Eurisko [Len83] for guiding exploratory computations.

Relative allocation of funds provides a clean high-level framework for sponsorship. 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 by itself, a manager can adjust the widths of its output pipes and the size of its reservoir. The top-level manager for a computation controls the total amount of funding which is pumped into the root of the sponsorship hierarchy. Since funding cannot 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 2-4.

For example, consider a homogeneous task that can spawn a large number of similar subtasks. A simple, effective manager strategy would be to fund each child equally. Since 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 those children which are most productive or cost-effective.

This view of sponsorship networks greatly simplifies many issues related to funding by removing low-level details such as absolute funding amounts and the need for complex funding request and evaluation protocols. Nevertheless, we recognize that some sophisticated managers may be able to make better decisions when provided access to such low-level knowledge. Our current system allows managers to make explicit requests for detailed information to the local resource manager when necessary.

At the highest level, the allocation of funds to users is negotiated by human system administrators. Some users are able to earn the funds they spend by selling unused time on their personal workstations. Special provisions are needed for users with processing requirements that far exceed their earning potentials. The integration of the *Spawn* economy with human organizations raises many interesting issues.

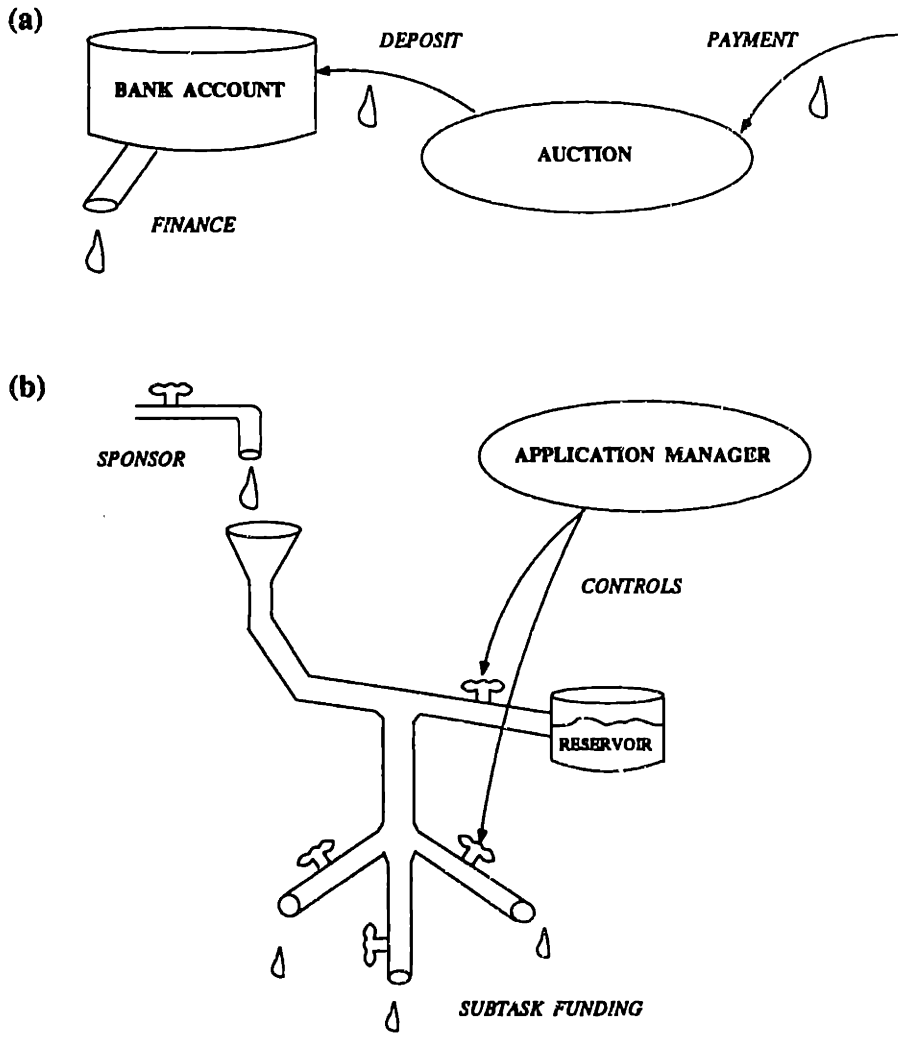


Figure 2-4: Flow of Funds

(a) The auction at a seller node receives payments and deposits them in its bank account. The seller can finance its own applications through this account.
(b) An application's sponsor continually adds funds whose distribution is controlled by the application manager.

2.5 Implementation

Spawn runs on a network of heterogeneous Unix workstations. Such networks are becoming increasingly popular in both corporate and academic computing environments. 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* [Mal88] and *Worms* [Sho82]. A network of Unix workstations provides a minimal substrate upon which *Spawn* can be successfully implemented. Ideally, we would prefer to take advantage of a sophisticated distributed operating system and a programming language designed for open systems [Tan85, Ras88, Kah88, Lis88, Man87]. Unfortunately, these tools are neither generally nor uniformly available on the existing machines and networks which we use; many are still research prototypes. Note that 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).

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 mechanism for dynamic re-routing of messages and a facility for the delegation of responsibility. Every workstation that participates in the *Spawn* economy executes a *router* process that maintains a cache of defunct processes and their "forwarding addresses". Message sends which result in errors may use the router to deliver messages to a forwarding address. 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.

There are two major 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 much better solution would be for *Spawn* to utilize a substrate for reliable distributed computation that provides robust recovery from failure by using atomic transactions [Lis83, Par83]. 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. The secure use of money in computational economies is an important research topic that we have not addressed [Mil87, Kah89].

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). It is in experimental use on numerous Sun workstations at Xerox PARC.

Chapter 3

Experiments

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 an asynchronous Monte-Carlo simulation which we ran concurrently in a network of Sun workstations.

The Monte-Carlo algorithm was chosen because it is paradigmatic of simulation techniques requiring great amounts of CPU time and is readily converted into a parallel algorithm. Because of the limited need for communication among processes, it is easy to create a collection of processes that can replicate themselves into an arbitrary number of machines. Monte-Carlo is a probabilistic algorithm which computes average properties of systems [Sob75]. 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. Moreover, 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 used in a number of applications, including the numerical evaluation of integrals, determining the behavior of complicated states of matter, and exploring interactions in ecosystems and economics.

Monte-Carlo is an ideal algorithm for our purposes, as it easily decomposes into an arbitrary number of subtasks, each performing a number of independent trials. Thus, it allows a simple tradeoff between longer computations using a few machines versus many short ones employing numerous processors. In order to avoid unnecessary price competition we made the processes involved in a single Monte-Carlo task friendly; i.e., no price competition occurs between them.

In this chapter we first study the utilization of idle machines, and determine the fairness of resource distribution. We then investigate the behavior of price equilibria and their fluctuations. Finally, we study the appearance of price differentials in heterogeneous systems. These studies are conducted by first considering small, easy to understand, fully-connected networks, and then examining larger, more sparsely connected ones. This is important in order to establish the scaling properties of these systems to very large sizes. This last point is underscored by simulations of *Spawn's* pricing behavior for large configurations.

3.1 Use of Idle Machines

The first experiment measured the efficiency with which idle machines are used in *Spawn*. To do this we introduced one Monte-Carlo task into a network of otherwise idle machines. Each application manager tried to spawn two submanagers and did not ask for extensions. As this task executed, it spawned processes in all of the available machines. We then measured the number of trials in the Monte-Carlo task executed per second as a function of the number of such machines in the network. Figure 3-1 shows the resultant speeds compared to those which could have ideally been attained by running independent Monte-Carlo tasks on the same set of machines. These latter speeds were computed by measuring the speed on individual Sun4/110's and Sun4/260's (11,000 and 14,700 trials/sec., respectively). Because of the negligible communication overhead in Monte Carlo, we observe a linear relationship; the slope

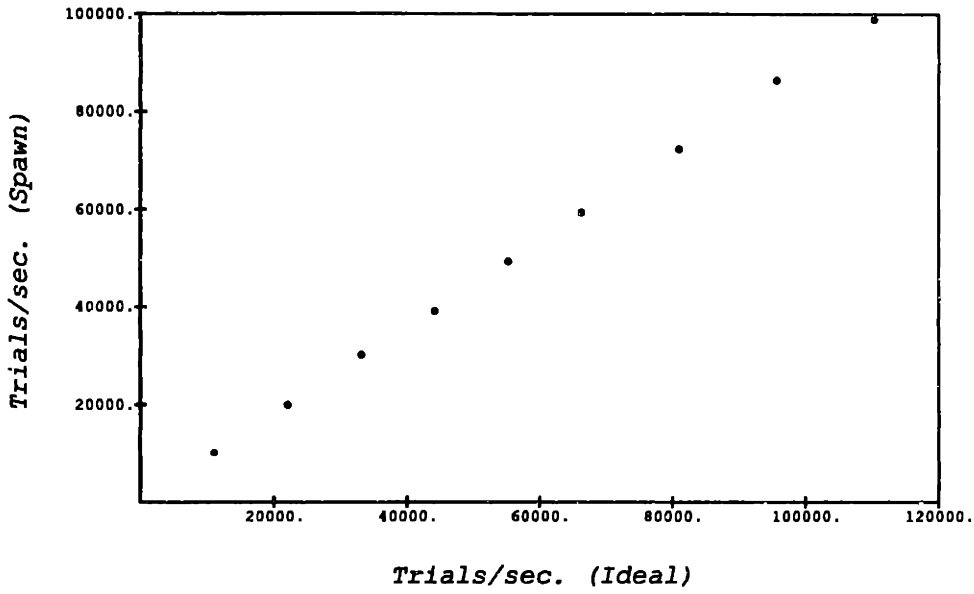


Figure 3-1: Spawn Performance

Number of Monte-Carlo trials per second calculated by *Spawn* vs. total number of trials/sec. that could be performed concurrently on independent machines for 1,2,...,9 machines. The time slices were all 60 sec. The best linear fit indicates that *Spawn* operates with 89.7% efficiency (i.e., 10.3% overhead) for this particular task and granularity size. When the time slices are doubled to 120 sec., the overhead drops to 7.6%.

of approximately 0.9 essentially characterizes the efficiency of the *Spawn* system. The efficiency can be increased by several percent by increasing the length of the timeslice and eliminating runtime diagnostics. Thus, *Spawn* is effectively able to run decomposable, homogeneous tasks on otherwise idle machines.

3.2 Fairness of Resource Distribution

In order to test the distribution of resources when there are competing tasks executing in *Spawn*, we measured the resource utilization with multiple versions of the Monte-Carlo task used above, given various initial funding ratios. All of the agents used a "money is no object" strategy: they bid on the auction that was slated to begin its next timeslice the earliest. The results for a number of representative examples are

Full Connectivity	
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

Local Connectivity	
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 3-2: Fairness of Resource Allocation

Fairness of *Spawn* for various funding ratios among tasks. 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 agents 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 agents accepted extensions and funded children based on cost and performance.

summarized in Figure 3-2.

As we can see from the table, 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. We thus expect a proper response in more complex situations; for example, where tasks are continually entering and leaving the system or where tasks are funded dynamically based upon partial results. An example of the latter case is an application which provides tasks with most of their funds near the beginning of a computation to get a rough estimate, then pays less for further accuracy, as can be appropriate in Monte-Carlo calculations. For these runs we have observed efficiencies between 80% and 90%. The higher figure is obtained when there is no competition; i.e., when there is only one root task in the system.

3.3 Prices

Since *Spawn* uses currency and prices as its basic mechanism for resource allocation, we are interested in how close it comes to behaving like a market; i.e., whether a

meaningful market price for processor time can 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 number of buyers and sellers, and to variations in their strategies? In this section, we experimentally examine the existence of equilibrium prices, transients, and fluctuations. We also investigate the effects of heterogeneous processing elements on prices in *Spawn's* computational economy.

3.3.1 Equilibrium

The simplest case that we consider involves a fully-connected network of homogeneous machines (i.e., each machine is equally valued by the tasks). Each agent placed a bid on the auction which was slated to finish first, and bid elsewhere only upon receiving a rejection notice. Figure 3-3 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 fluctuations were relatively 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 in Figure 3-3 (the standard deviation measured from $t = 400$ to $t = 1200$) were approximately 13%. These can be attributed to the small number of processes participating in the experiment. A closer analysis revealed that, within each of the six auctions, the fluctuations are in the vicinity of 25%.

In addition, we examined the behavior when the network was less densely connected, a characteristic of large networks. For these experiments, the application managers allocated some of their funding for extensions, enabling them to persist for several time slices. This allowed the management hierarchy to become deeper, and facilitated the spread of tasks to distant nodes in the network.

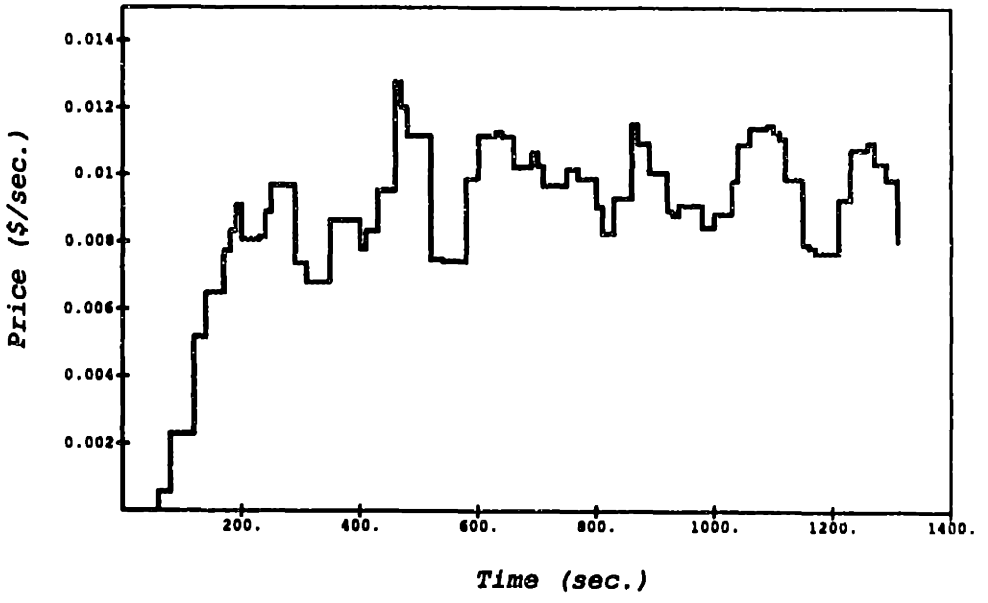


Figure 3-3: Price Equilibrium: Full Connectivity

Average price as a function of time in a network with six fully-connected machines. Funding is supplied every 10 seconds to three tasks A, B, and C in the amounts \$0.30, \$0.20, and \$0.10, respectively, and the length of each time-slice is 60 seconds. Thus 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.

When machines had few connections to one another, employing the simple funding strategy used earlier (equal funding to each child) led to a significant difference in prices among the machines. This was due to the limited ability of the highly funded roots to provide funds to distant submanagers. 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 less expensive machines are bid up by the smart managers looking for a bargain.

To address the problem of price differentials, a more sophisticated funding strategy was developed for the applications. Instead of funding each child equally, those children running on cheaper machines were given the funding. This strategy eliminated the price difference; the resulting behavior is shown in Figure 3-4. The ability to balance prices merely by changing an application's funding strategy illustrates the flexibility of the market mechanism.

3.3.2 Transients

In addition to obtaining meaningful prices, the system should adapt to changes in load. 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 of the Monte-Carlo tasks described above, waited for the establishment of equilibrium, and then added a third task. Figure 3-5 shows the resulting average price as a function of time. Before the addition of the third task, the observed equilibrium price is close to 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 the processors. Note that this transient time is the same as when starting from an initially idle network (compare with Figure 3-3).

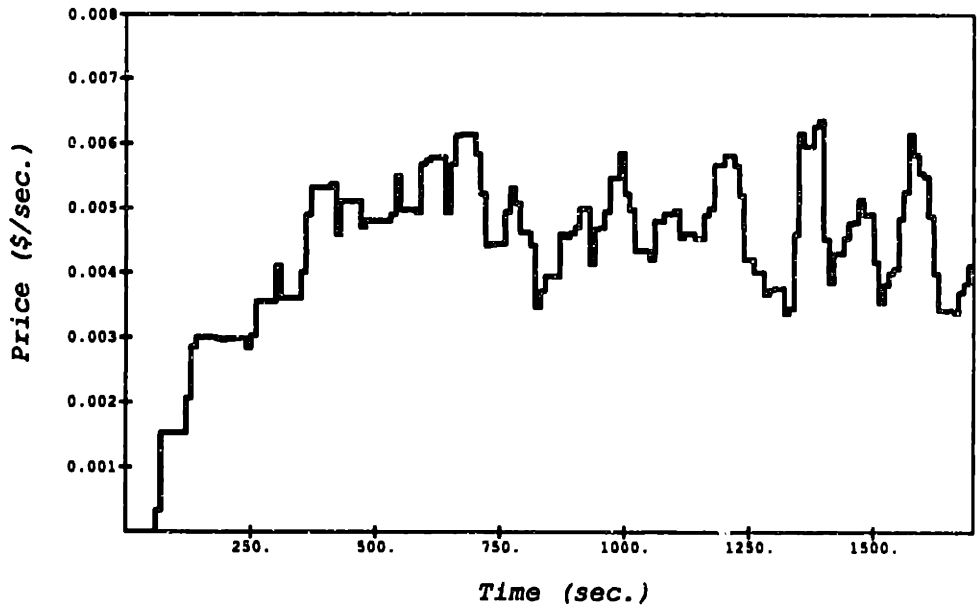


Figure 3-4: Price Equilibrium: Local Connectivity

Average price as a function of time in a network with twelve locally-connected machines configured as a torus. Funding is supplied every 10 seconds to three tasks A, B, and C in the amounts \$0.30, \$0.20, and \$0.10, respectively, and the length of each time-slice is 60 seconds. Thus the total rate at which money enters the system is \$0.06/sec. These tasks were started on nonadjacent nodes. Measuring the average price between $t = 400$ and $t = 1600$ sec., we find that the average price = \$0.00487/sec, with standard deviation \$0.0007/sec.

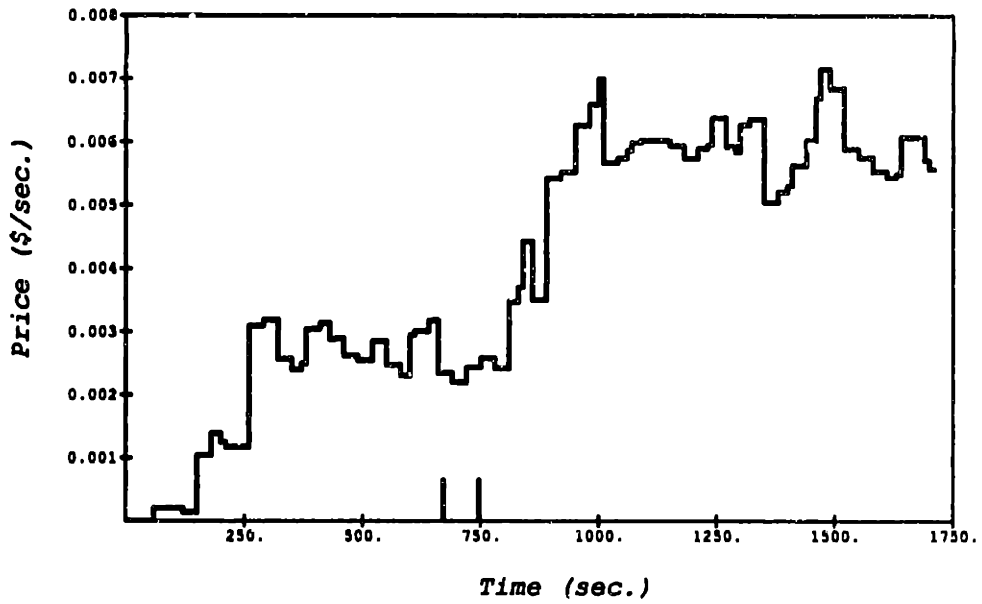


Figure 3-5: Market Price: Adaptation

Average price as a function of time in a network with six fully-connected machines. Initially, two tasks, each of which received \$0.10 every 10 seconds, competed with one another. The equilibrium price quickly reached a level of about \$0.0033/sec, around which it fluctuated. At t = 671 sec. (marked by first line segment on time axis), a third task, which received \$0.20 every 10 seconds, was added to the system. It made its first successful bid 75 sec. later (second line segment). After this point, the price quickly rose to a new level, fluctuating about a value of \$0.0067/sec. after a transient lasting just a few auction cycles.

The transient time could be reduced even more if new tasks started with enough funding to match the current market price instead of waiting for sufficient funds to accumulate. At any rate, the fact that an equilibrium market price is readily established, even with relatively few agents, makes such strategies possible.

3.3.3 High priority tasks

Systems that schedule tasks according to fixed priorities enable high-priority tasks to run right away. *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 tasks were running; figures 3-6 and 3-7 illustrate the system's response. The experimental market initially consisted of two tasks continually funded at rates of \$0.01/sec. and \$0.02/sec. After the initial transients subsided, a new task was injected into the market with a funding rate of \$0.07/sec. and a total allocation of \$50.40.

As the figures show, the high-priority task rapidly took over most of the available resources. This confirms the responsiveness of the system to sudden changes in demand. This is not a surprising result, given the previous results on fairness of resource allocation and the short observed transients.

3.3.4 Price differentials in heterogeneous systems

When the machines in the network are not homogeneous, price differentials develop between machines, reflecting their relative values. Figure 3-8 shows the prices in a system with 9 auctions: 3 running on Sun 4/260's and 6 on Sun 4/110's. The agents hold a Sun 4/260 to be 1.4 times as valuable as a Sun 4/110. 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 close to 1.4: \$0.0031/sec. for the 110's and \$0.0054/sec. for the 260's.

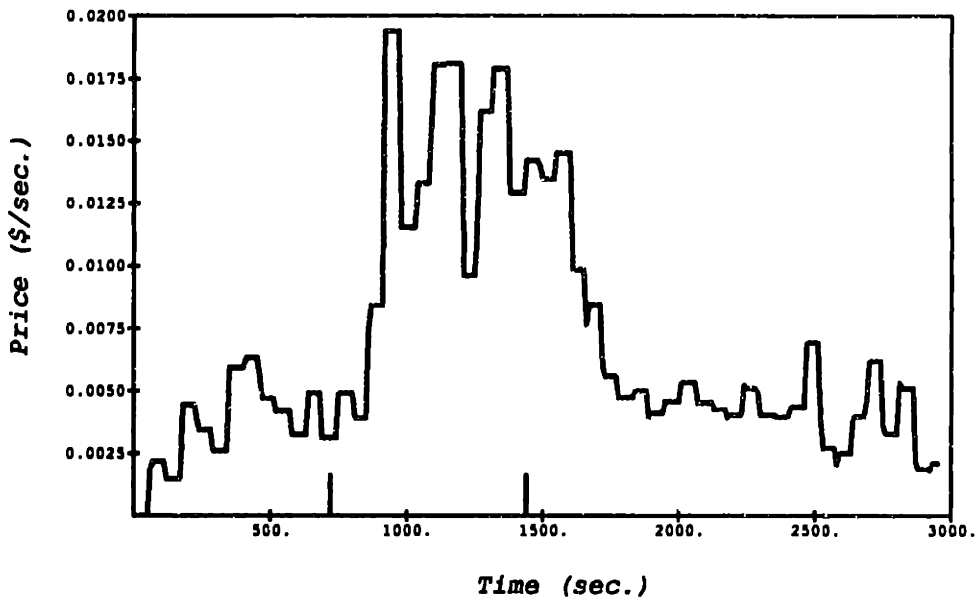


Figure 3-6: Market Response to a High Priority Task

Average price as a function of time when a high-priority task is introduced into a system with low-priority jobs in it. The first vertical line segment on the time axis denotes the introduction of the high-priority task, and the second one the termination of its funding.

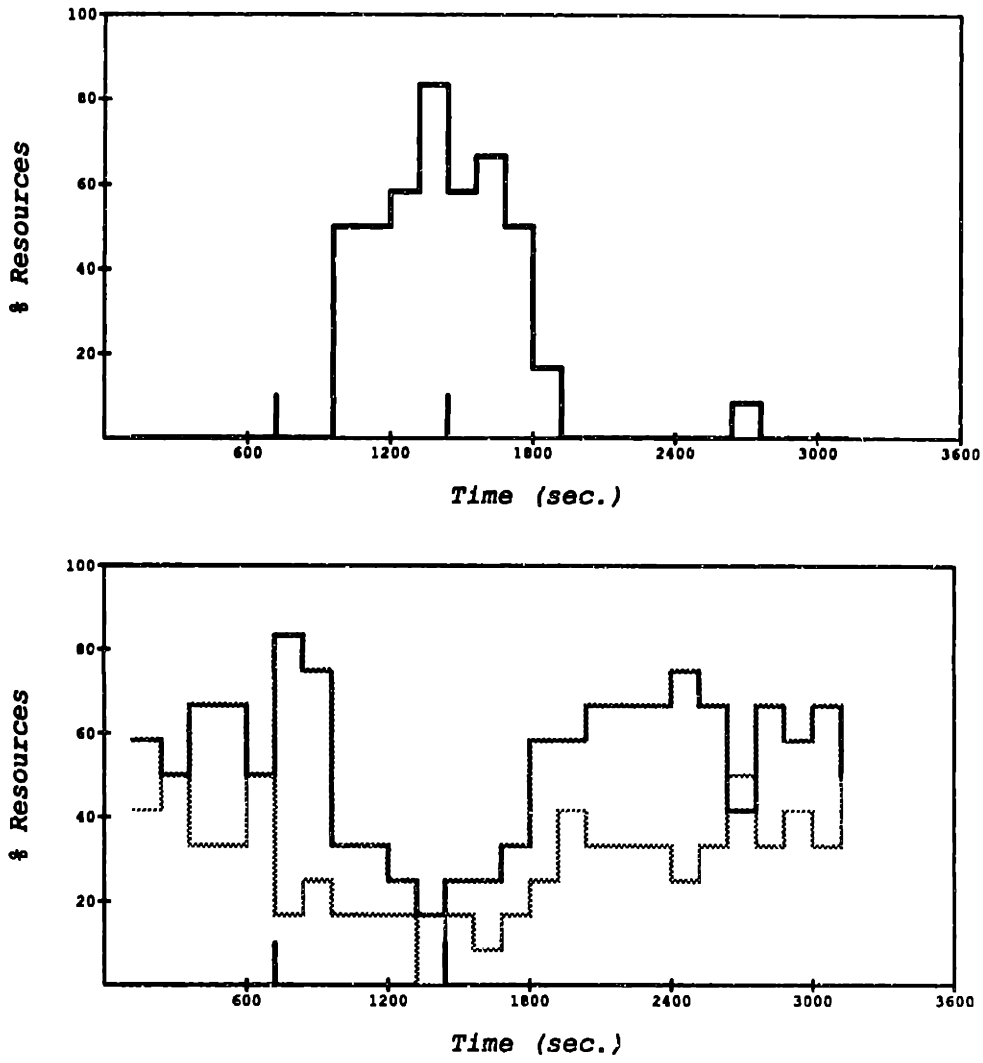


Figure 3-7: High-Priority Fairness

Resource usage for the tasks. Each figure plots the fraction of available resources used vs. time. The upper figure corresponds to the high-priority task while the lower one corresponds to the two low-priority tasks.

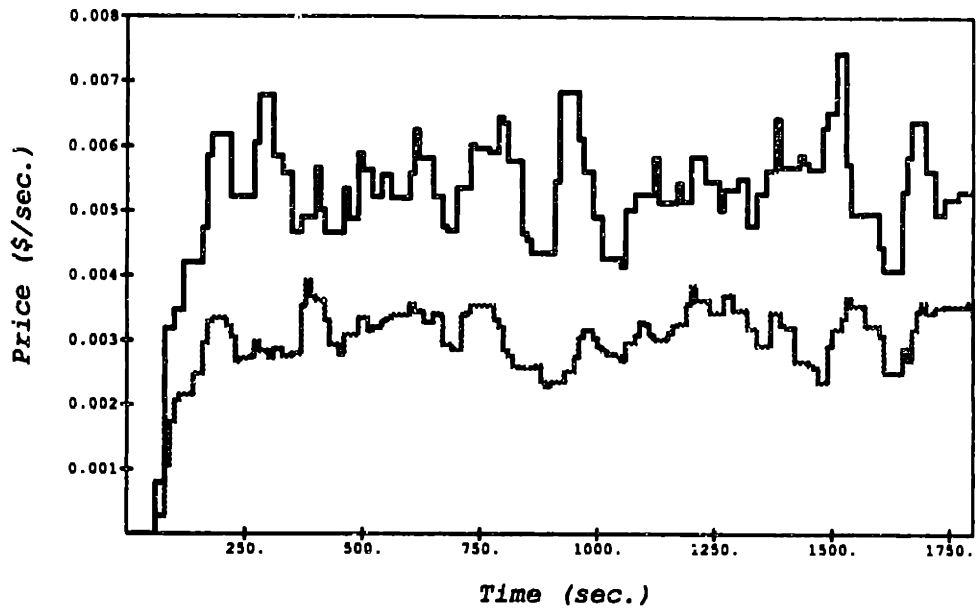


Figure 3-8: Price Differential in a Heterogeneous System

Price differentials in an inhomogeneous system. The price averaged over 3 Sun 4/260's is in black; the price averaged over 6 Sun 4/110's is in gray. The time slices were 60 sec. Each of four roots were funded with $0.10 every 10 sec.

3.3.5 Scaling to larger systems

The experiments performed above exhibit noticeable price fluctuations, unlike human markets. Since the human economy consists of a very large number of agents, it is of interest to see the extent to which the fluctuations diminish as the system size increases. Because addressing this question would require systems with more tasks and machines than are available to us, we instead simulate the key features of the *Spawn* system: its funding and bidding.

The simulation follows the evolution of the agents' bank accounts and the auction prices, given various initial conditions. The basic iterated loop of the simulation consists of calculating the time of the next event – either the beginning of an auction timeslice or the arrival of a funding droplet to one of the agents – and updating the agents' bank accounts accordingly. If the event happens to be the beginning of an auction's timeslice, the price at which it was bought is recorded. The evolution of

the simulated system is completely determined by the initial conditions.

The simulation differs somewhat from the actual implementation of *Spawn* in that the tree of agents funded by a given root is replaced by a group of agents, each of which receives an identical amount of funding per funding period. Variation in funding among agents descended from the same root, which arises primarily from the distribution in the number of management levels above an agent, is completely ignored. None of the complicated issues of parentage and delegation of responsibility for orphans to other manager processes are dealt with in the simulation.

As in *Spawn*, agents which are members of the same group do not compete with one another. The number of agents in each group is assumed to be fixed by the funding ratio, the number of auctions, and the branching ratio used by the managers when they try to spawn the next generation of agents. For example, in the run in Figure 3-2 (page 29) with funding ratio 3:2:1 and 6 fully-connected auctions, the branching ratio was 2. Therefore, at any given moment there were about 12 agents bidding for the next timeslice. Since the funding ratio is 3:2:1, the simulation would set the number of agents in each group to be 6, 4, and 2, respectively.

The simulations show that the magnitude of the fluctuations relative to the average price is approximately proportional to $1/N$, where N is the number of agents in the system. We have verified this dependence for many different configurations: multiple auctions, different sizes of groups, etc. Thus, even if initial experiments on small networks display large price fluctuations, we expect the price equilibria to be reasonably well-defined in medium-sized networks. In addition, differences in the average prices of various auctions diminish as the network grows in size.

The simulations also establish that the price fluctuations are similarly insensitive to how much competition there is in the system, provided that there is more than one competitor. This was determined by running the simulator with a large number of agents (60) which were arranged in anywhere from one to twenty groups and noting that, for more than one group, there was little variation and no discernible trend in

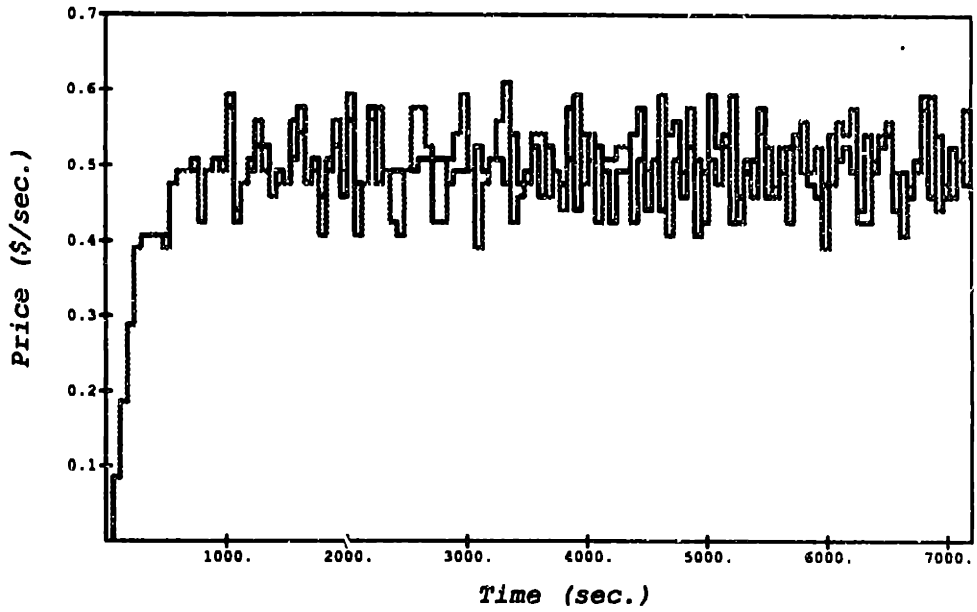


Figure 3-9: Sensitivity to Initial Conditions

Price as a function of time in the simulated *Spawn* system. The prices which result from starting the roots 0.5 sec. apart are represented by the black curve; those which result from starting the roots 0.6 sec. apart are represented by the gray curve. Although the initial behaviors are very similar, they diverge noticeably near $t = 1000$. The complete lack of correlation is particularly evident between $t = 2000$ and $t = 3000$.

the price fluctuations.

Although the fluctuations decrease with N, their irregular nature may seem somewhat mysterious at first because the simulation is completely devoid of any random inputs. Further investigation reveals that the price fluctuations are extremely sensitive to the initial conditions. Figure 3-9 illustrates the fact that two sets of starting times for the roots which are extremely close eventually lead to completely divergent behavior. This sensitivity to initial conditions is a characteristic of deterministic chaos [Par87]. The chaotic nature of *Spawn* arises from the nonlinear updating of the agents' bank accounts – the selection of the highest bidder (i.e., the one with the highest bank account) and the subtraction of the second highest bid from the highest bidder's bank account.

As a consequence of the chaotic nature of *Spawn*, no two experiments will ever

yield exactly the same result. Nevertheless, by carefully measuring the starting times of roots in an actual Spawn experiment on a single auction, we have been able to verify that the simulation can reproduce the actual prices for about 25 auction timeslices before diverging beyond recognition.

Further simulations suggest that, in addition to affecting the the exact history of prices, the initial conditions can also significantly influence the measured average fluctuations. This is due to the fact that time-averages of chaotic series are inherently unreliable even if they are taken over very long periods of time. In either case, if we are interested in measuring price fluctuations accurately, it is necessary to run experiments for long periods of time to ensure adequate statistics.

Despite the fact that the fluctuations are chaotic, this should not adversely affect the performance of *Spawn*, since their magnitude is proportional to $1/N$. Furthermore, our fairness results (observed for the actual system) show that *Spawn* performs well even for small cases. This is corroborated by measurements of fairness in larger, simulated systems.

Chapter 4

Related Work

4.1 Exploiting Idle Time

An early experiment in distributed computation was the *Worms* project conducted by Shoch and Hupp at Xerox PARC [Sho82]. A *worm* in their system 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, thereby causing the worm as a whole to "grow" as it expanded into additional networked computers. The simplest worm program was the existential worm: a computation which sought only to survive and grow. Other worm computations included novel, but not particularly useful, applications such as a distributed alarm clock and a roving "cartoon of the day" billboard system. Perhaps the most interesting application used worms to generate computer animation sequences by parcelling the graphic computations by frames to remote machines.

One shortcoming of the Worms project is that it relied upon the Xerox Alto workstation. The Alto was designed for a single user, and lacks provisions for protection and sharing (such as multiple memory address spaces) that can be found in more modern workstations [Tha82]. An idle machine had to be rebooted to execute a worm segment, and bug-ridden worms could (and did) crash numerous machines [Sho82].

A more substantial problem was the rudimentary control structure employed, which presented difficulties in managing stable, well-behaved worms. These serious problems, while highlighting important directions for future research, ultimately rendered the Worms project impractical. *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 much more sophisticated than the simple strategies employed in Worms.

4.2 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 [Cof73, Pet85].

The problem of scheduling in distributed computer systems has been a topic of active research for many years; a good survey can be found in [Tan85]. In the area of distributed processor scheduling, most of the work 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 [Cho82, Lo84], 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 in order to make distributed scheduling decisions. A variety of heuristic methods have been studied. These include random, limited exchanges of information [Bar85], techniques from expert systems and rule-based programming [Lo87], knowledge-based solutions [Pas88], statistical time-series analysis [Hai88], and distributed bidding and negotiation metaphors [Smi80, Mal83]. It is this last class of heuristic algorithms, based upon bidding and market-like negotiations, which are most related to *Spawn's* distributed computational economy. Research involving market-

based mechanisms is considered in greater detail in the following section.

4.3 Computational Markets

A predecessor to automatic market-based strategies for managing computational resources was Sutherland's "futures market" in computer time [Sut68]. Users engaged in a continuous auction which allowed the price of computer time on a single PDP-1 to fluctuate with demand. The system was run manually; users posted bids on a physical scheduling board, which reflected the current state of the auction. Users were assigned different amounts of "currency" by system administrators; the amounts depended on the importance of their projects. This currency was then used to reserve blocks of time: a user would indicate a block of time and the amount he was willing to pay for it. Higher bids could pre-empt lower ones. Since the price of time fluctuated according to demand, users were presented with a choice between short periods of expensive prime time and longer blocks during less-desirable off-peak hours. Users with higher priority were issued more currency than those with lower priority, who they could consequently outbid. Sutherland reported that his system worked more efficiently than any of the many other schemes which he had tried.

Malone's *Enterprise* system is a decentralized market-like scheduler for load-sharing in distributed computing environments [Mal83, Mal87, Mal88]. Enterprise, which is similar in many respects to the Contract Net protocol of Smith and Davis [Smi80, Smi81, Dav83], 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 which is 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 an implementation of his system on Xerox D-machine workstations, Malone performed a set of simulations to test the effects of various preset system parameters on the performance of the system. He found that pooling tasks in a distributed system using Enterprise resulted in significant performance improvements over running the same tasks locally in their home machines. His simulations indicated that such improvements are robust over a wide range of system parameters, including a variety of message-delay times and substantial errors in processing time estimates.

Like Worms, Enterprise suffered from the unfortunate protection limitations of computer workstations in which all processes share a single memory address space. Moreover, since Enterprise was implemented using the Interlisp-D environment, a client could arbitrarily mutate a contractor's global Lisp environment, resulting in many difficulties. For example, a common problem was that client document formatting processes often required the loading of bulky font information, which used up substantial amounts of memory in the contractor workstation. The possibility of undesirable side-effects to one's environment was a disincentive for use, and Enterprise failed to remain in regular operation.

Aside from implementation-related problems, Enterprise was also limited by a number of fundamental design decisions. First, the system had no provisions for true 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. Price information would also obviate the need for complicating the system with artificial "priorities" such as those employed by Enterprise. Another limitation was a lack of support for operation

in a truly heterogeneous distributed system (i.e., one populated by workstations with different CPU types).

Recently, Miller and Drexler published a set of papers on the subject of *agoric* systems [Mil88, Dre88]. They define an agoric system (from the Greek *agora*: a meeting and market place) as "a software system using market mechanisms, based on foundations that provide for the *encapsulation* and *communication* of *information*, *access*, and *resources* among *objects*." Miller and Drexler envision a world in which large, evolved agoric systems with valuable emergent properties are inextricably linked to larger human economic markets. Their work consists of a wealth of interesting but untested conjectures about the desirability, behavior, and efficiency of computational markets. The agoric systems which they describe are populated with *agents* following *business strategies*, which may have both pre-defined and adaptive components. Given a set of initial seed strategies, it is their hope that agoric systems will grow and evolve through a combination of human design and adaptive agent behavior. In [Dre88], auction-based processor scheduling, rent-based storage management, and agent occupations such as managers, sponsors, and consultants are discussed. Due to the complexity of their proposed mechanisms as well as several open problems, their ideas remain unimplemented. Their work on processor scheduling is most relevant to this thesis, but is primarily concerned with the efficient auctioning of processor time within a *single* serial computer system. They are thus concerned with problems such as the apparent paradox that agents who wish to purchase time must obtain time to place their bids. These issues have only limited relevance to the *Spawn* system. Nevertheless, their uniprocessor algorithm shares with *Spawn* the features of linearly increasing bids across a series of second-price auctions [Dre88]. The agoric systems approach as outlined by Miller and Drexler provides a plethora of interesting topics for future research.

A very recent publication by Ferguson, Yemini, and Nikolaou describes what they term *microeconomic algorithms* for load balancing in distributed computer systems

[Fer88]. Their scheme, like the agoric approach described by Miller and Drexler, involves a competitive market for resources based upon a pricing mechanism. In their system, *jobs* compete for communication and processing resources by bidding on auctions held by *processors*. Jobs are agents which have a task to perform, and may enter the system at any processor. They 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 communications bandwidth to jobs, trying only to selfishly maximize their own revenues. Information about current prices is available to jobs via *bulletin boards*, which are maintained by each processor; bulletin boards contain pricing "advertisements" from neighboring processors. A job schedules itself on a processor through the following sequence of events: it queries the local bulletin board, computes a *budget set* of affordable processors, chooses an element of that set based upon a preference relation that is a function of price and service time, and then places a bid on that processor's auction. A processor is responsible for auctioning its idle resources and updating its prices on local bulletin boards. When a processor resource becomes idle, the processor holds an auction to determine which job will next get that resource. Different auction models, including *sealed bid auctions* and *Dutch auctions* were used in various instantiations of their system. Based upon Adam Smith's classic "invisible hand" argument from economics, it is reasoned that selfish local optimizations by 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 CPU and communication resources that is comparable, and in many cases superior, to traditional algorithms.

These results are very encouraging; through simulation techniques Ferguson *et. al.* have demonstrated the viability of market-based mechanisms for resource management in distributed systems. However, it is important to note that they were primarily

concerned with scheduling unrelated tasks in a network of homogeneous processors [Fer88]. Their sharp focus on the collective performance of a set of independent processes caused them to neglect 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 which it had on the performance of individual jobs.

In contrast to the simulations described in [Fer88], *Spawn* was explicitly designed to support large concurrent applications that execute in a heterogeneous distributed system. The experiments and analysis we present in this thesis examine the effects of various funding allocation strategies. In particular, we explore the impact of funding policies on effective dynamic priorities and the fairness of resource distribution for a set of competing, concurrent applications. The dynamic behavior of prices for a variety of system configurations, including those with heterogeneous processors, is also examined. We believe that *Spawn* is the first market-based system which has considered the issue of funding concurrent applications. One of the fundamental innovations of the *Spawn* system is its sponsorship mechanism, which manages a concurrent computation through the relative allocation of continuously-flowing funds in its tree of active processes.

4.4 Sponsorship 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 single tasks, since the primary concern is with the execution of concurrent, tree-structured computations.

Spawn's sponsorship mechanism was influenced by programming language constructs in the actor languages developed by the Message-Passing Semantics Group at

MIT. The actor model of computation is explained in [Agh86]. The idea of *sponsored* computations was first proposed for managing resources in the actor language *Act2* [The83]; the role of sponsors has evolved in successive actor languages. The most recent actor language, *Acore* [Man87], 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 has refined the actor notion of sponsorship in the context of a computational economy. Two weaknesses with the actor approach were solved in *Spawn*. First, in Acore, sponsors are forced to make critical decisions at specified points; they must respond quickly to requests for additional ticks. Second, sponsors are also required to make decisions involving absolute amounts of funding. In *Spawn*, we found it desirable to avoid these constraints. Our solution is based on the idea of a *continuous* flow of funds through the tree-structured sponsorship hierarchy. Instead of requiring subtasks to explicitly request absolute amounts of funding from their sponsors, we allow sponsors to dynamically adjust the *relative* outgoing allocations of funds to their children.[1] This solution avoids critical decision points because sponsors can adjust relative funding allocations at any time. Relative allocations are easier for

[1]Note that it is possible to explicitly program relative funding allocation given a mechanism for distributing absolute amounts of funding. In this way, the actor sponsorship mechanisms can be made to behave like the *Spawn* sponsorship hierarchy. However, this approach has not been used in practice. By providing relative allocation as a primitive operation, *Spawn* encourages the sponsorship approach outlined above.

sponsors to reason about, and reduce the allocation problem to one of adjusting a few "weights"; this facilitates simple, adaptive funding algorithms. Finally, the size of a concurrent application's tree of active processes dynamically adjusts to market conditions, as explained in Section 2.4.

4.5 Computational Dynamics

The dynamical behavior of open systems is the focus of active research by the Dynamics of Computation Group at Xerox PARC. Their work analyzes the dynamics of *computational ecologies*, so named due to their resemblance to social and biological organizations.

Using analytic tools from statistical physics and nonlinear game dynamics, Huberman and Hogg discovered that open systems composed of even very simple processes can exhibit a rich variety of dynamical behavior, including regimes characterized by fixed points, oscillations, and chaos [Hub88]. Computer simulations were developed which allowed the observation of resource allocation dynamics in computational ecosystems. These simulations enabled Kephart, Hogg, and Huberman to verify their theoretical predications concerning the appearance of complicated dynamical behavior [Kep89].

Spawn has served as an experimental workbench for studying the behavior of a real computational ecology. The ability to use *Spawn* to perform quantitative experiments in computational dynamics has facilitated the development of a synergistic relationship between theory and implementation. This relationship has advanced both our abstract models and our experimental system.

Chapter 5

Conclusions

5.1 Summary

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, dynamic load balancing, the management of concurrent computations, and the utilization of otherwise-wasted computational resources. As the experiments show, the system can successfully handle these problems without resorting to global controls. Furthermore, distributed computations competing for resources can be efficiently managed with acceptable overhead. This allows the use of existing networks for certain classes of large, easily-parallelized computations that are commonly run on supercomputers. This is the case with Monte-Carlo, which has emerged as a favorite simulation tool in the physical sciences.

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 very small number of agents (of order 10) can produce an identifiable market, since fluctuations were not able to obscure the stable equilibria in prices. Moreover, *Spawn* has opened the door to a number of experiments that can

test proposed methods for dealing with key characteristics of distributed computation, such as imperfect knowledge and delayed information.

5.2 Directions for Future Research

One research area that warrants considerable attention is the smooth integration of *Spawn's* ideas with a robust language for parallel and distributed computation. As an experimental research tool, *Spawn* is somewhat fragile and awkward to use. Simple, expressive linguistic mechanisms would facilitate more rapid advancement by researchers concerned with market-based computational resource management. We are currently evaluating the suitability of available platforms for distributed and concurrent computation, upon which we can implement an improved version of our software.

Another area of interest is the specification and implementation of more sophisticated funding strategies for controlling concurrent computations. This thesis has demonstrated the effectiveness of a small number of simple funding strategies, but 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" [Ell85] 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 is an open question.

An additional topic for further work is the determination of suitable granularities for market-based resource management. *Spawn* operates in the context of very coarse-grained distributed computations. Attempting to use market mechanisms to solve other problems in distributed computing (such as reclaiming wasted resources consumed by orphan processes [Nel81] and performing intelligent object migration)

would provide insight into the general applicability of such techniques. Much would also be learned by attempting to address the resource management problems of fine-grain parallel computers. For example, the market mechanisms explored in this thesis could possibly be adapted to perform scheduling and load-balancing on parallel message-passing machines [Dal89].

A final area for further study is diversity in computational economies. Although we have focused on the purchase of processor time only, 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 enabling tasks to 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 that studied here, in which computational results obtained by agents can themselves become marketable goods of potential use to other agents [Mil88]. 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.

Appendix A

Spawn Library Routines

A.1 Overview

This appendix contains the specifications for the application programmer's interface to the *Spawn* system. The interface specifications are given for a simple stack-based Algol-like language. Specifications are informally documented in the style presented in [Lis86]. These routines provide a *Spawn* application programmer with facilities for interaction with the *Spawn* system processes.

An application worker uses the routines provided in the **WorkerLib** interface in order to communicate with its manager. An application manager makes use of the routines contained in the **ManagerLib** interface to interact with distributed application processes and the *Spawn* system processes.

A.2 WorkerLib

- *Spawn Worker Library Routine Specifications*

WorkerLib_Initialize: **procedure**() **returns**(success: boolean)

requires: Must be called prior to calling any other WorkerLib routines.

effects: Initializes the Spawn Worker interface. Returns true if successful, false otherwise.

WorkerLib_SelfTaskId: **procedure**() **returns**(taskId: integer)

requires: WorkerLib interface has been initialized.

effects: Returns the unique task identifier for the worker.

WorkerLib_MessageToSponsor: **procedure**(buffer: array of character, length: integer) **returns**(success: boolean)

requires: WorkerLib interface has been initialized.

effects: Delivers the opaque application-specific message encoded by buffer[1..length] to the worker's sponsor. Returns true if successful, false otherwise.

A.3 ManagerLib

• *Spawn Manager Library Routine Specifications*

ManagerLib_Initialize: **procedure**() **returns**(success: boolean)

requires: Must be called prior to calling any other ManagerLib routines.

effects: Initializes the Spawn Manager interface. Returns true if successful, false otherwise.

ManagerLib_SelfTaskId: **procedure**() **returns**(taskId: integer)

requires: ManagerLib interface has been initialized.

effects: Returns the unique task identifier for the manager.

ManagerLib_SelfUserId: **procedure**() **returns**(user: string)

requires: ManagerLib interface has been initialized.

effects: Returns the user identification string for the manager.

ManagerLib_SelfGroupId: **procedure**() **returns**(group: string)

requires: ManagerLib interface has been initialized.

effects: Returns the group identification string for the manager.

ManagerLib_MessageToSponsor: **procedure**(buffer: array of character, length: integer) **returns**(success: boolean)

requires: ManagerLib interface has been initialized.

effects: Delivers the opaque application-specific message encoded by buffer[1..length] to the manager's sponsor. Returns true if successful, false otherwise.

ManagerLib_MessageToSubmanager: **procedure**(taskId: integer, buffer: array of character, length: integer) **returns**(success: boolean)

requires: ManagerLib interface has been initialized.

effects: Delivers the opaque application-specific message encoded by buffer[1..length] to the submanager specified by taskId. Returns true if successful, false otherwise.

ManagerLib_DeclineExtensions: **procedure**() **returns**(success: boolean)

requires: ManagerLib interface has been initialized.

effects: Informs the local resource manager not to purchase extensions when they are offered. Returns true if successful, false otherwise.

ManagerLib_SetFundingAllowance: **procedure**(taskId: integer,
allowance: real) **returns**(success: boolean)

requires: ManagerLib interface has been initialized.

effects: Sets the relative funding allocation of the submanager named by taskId to allowance. Returns true if successful, false otherwise.

ManagerLib_SetSelfAllowance: **procedure**(allowance: real) **returns**(success: boolean)

requires: ManagerLib interface has been initialized.

effects: Sets the relative funding allocation of the manager's own reservoir to allowance. Returns true if successful, false otherwise.

ManagerLib_ReleaseReservoir: **procedure**(fraction: real) **returns**(success: boolean)

requires: ManagerLib interface has been initialized, $0.0 \leq$ fraction ≤ 1.0

effects: Distributes a portion of the funds accumulated in the manager's reservoir to its children (according to their current relative allowances). Returns true if successful, false otherwise.

ManagerLib_AbortWorker: **procedure**(taskId: integer) **returns**(success: boolean)

requires: ManagerLib interface has been initialized.

effects: Terminates the execution of the worker named by taskId. Returns true if successful, false otherwise.

ManagerLib_AbortManager: **procedure**(taskId: integer) **returns**(success: boolean)

requires: ManagerLib interface has been initialized.

effects: Terminates the execution of the submanager named by taskId. Any unused funds are returned prior to its termination. Returns true if successful, false otherwise.

ManagerLib_RepossessManager: **procedure**(taskId: integer) **returns**(success: boolean)

requires: ManagerLib interface has been initialized.

effects: Repossesses unused funds held by the submanager named by taskId. Returns true if successful, false otherwise.

ManagerLib_SpawnWorker: **procedure**(taskName: string,
taskArgs: array of string, nTaskArgs: integer,
workDirPath: string) **returns**(success: boolean)

requires: ManagerLib interface has been initialized.

effects: Requests that a worker process of type taskName be created locally. Returns true if successful, false otherwise.

The abstract taskName is resolved via the mappings contained in the caller's task file which map the abstract name and the local machine type into a concrete network path for the appropriate executable file.

The array taskArgs contains nTaskArgs command line arguments with which to invoke the executable file.

The network path workDirPath specifies the directory in which the executable should be invoked; i.e. it is the "current working directory" for the invocation.

An APPLICATION_WORKER_BIRTH message is delivered to the caller when the worker process becomes active.
An APPLICATION_WORKER_DEATH message is sent when the worker process terminates.

ManagerLib_SpawnComplexWorker: **procedure**(useTaskMap: boolean,
taskName: string, taskArgs: array of string,
nTaskArgs: integer, workDirPath: string,
outputHost: string, outputPort: integer) **returns**(success: boolean)

requires: ManagerLib interface has been initialized.

effects: Detailed interface for creating a local worker process. Returns true if successful, false otherwise. Works like the simpler SpawnWorker(), with the following differences:

- If useTaskMap is false, the name specified by taskName is not resolved via the mappings contained in the caller's task file. Instead, taskName is used as the concrete network path name for the appropriate executable file.

- Worker output is captured and sent to the port specified by outputPort on the machine specified by outputHost. If outputHost is nil, worker output is simply discarded. Note that is is very useful for capturing the results of "black-box" applications.

ManagerLib_SpawnManager: **procedure**(taskName: string,
taskArgs: array of string, nTaskArgs: integer,
workDirPath: string, subTaskMapPath: string,
essential: boolean,
workEstimate, workExtensionEstimate: integer,
warnTime: integer, allowance: real,
timeWeight, priceWeight: real,
taskGroup: string, intraGroupCompete: boolean,
taskUser: string, intraUserCompete: boolean) **returns**(success: boolean)

requires: ManagerLib interface has been initialized.

effects: Requests that a new submanager process of type taskName be spawned on a remote machine. Returns true if successful, false otherwise.

The abstract taskName is resolved via the mappings contained in the caller's task file which map the abstract name and remote machine type into a concrete network path name for the appropriate executable file.

The array taskArgs contains nTaskArgs command line arguments with which to invoke the executable file.

The network path workDirPath specifies the directory in which the executable should be invoked; i.e. it is the "current working directory" for the invocation. The network path subTaskMapPath specifies the location of the task file to use in resolving task names for spawn requests made by the new submanager.

The value of the "essential" flag indicates whether or not the new submanager requested is essential – i.e. it must be executed even if no time can be purchased during the caller's lifetime. If essential is set and the new task has not been scheduled when the caller terminates, responsibility for it is delegated to the caller's sponsor.

The workEstimate and workExtensionEstimate parameters specify the estimated processing times required for the new submanager (in units of seconds for a machine with a task-specific work-rating of unity). The warnTime argument indicates the absolute amount of warning time (in seconds) that should be provided to the new submanager if it becomes necessary to forcibly terminate it. A Unix SIGTERM signal is delivered to the submanager warnTime seconds prior to the delivery of a fatal SIGKILL signal.

The allowance given to the new submanager specifies its relative allocation of incoming funding. The total allowances for all submanagers and the caller's reservoir is normalized to sum to unity.

The timeWeight and priceWeight parameters allow the crude specification of a utility function used by the resource manager in placing bids on the caller's behalf. These parameters are based on identical parameters defined by Ferguson and his colleagues at Columbia. In the thesis, we *always* set timeWeight = 1.0 and priceWeight = 0.0, demonstrating a constant preferance for the machine with the soonest completion time, despite its price.

The taskUser and taskGroup arguments specify the user and group names for the new submanager's identity. If intraUserCompete is set, two tasks with the same user name are allowed to compete. If intra-

GroupCompete is set, then two tasks with the same group name are allowed to compete.

An APPLICATION_MANAGER_BIDDING message is delivered to the caller when the resource manager begins to bid for processing time on its behalf. An APPLICATION_MANAGER_ACTIVE message is delivered to the caller when the resource manager is successful in purchasing processing time for the new task. An APPLICATION_DETACH or APPLICATION_ABORT message is sent if the submanager process terminates.

ManagerLib_SelfPathInfo: **procedure**(workDirPath: string, taskMapPath: string)

requires: ManagerLib interface has been initialized.
modifies: workDirPath, taskMapPath
effects: Sets workDirPath to the network path for the current working directory. Sets taskMapPath to the network path for the task file used to resolve abstract task names into concrete executable files. Note that these paths were specified by the manager's sponsor before the manager's creation.

ManagerLib_SelfPurchasePrice: **procedure**(time, funds: real)

requires: ManagerLib interface has been initialized.
modifies: time, funds
effects: Sets time to the length of the time slice (in seconds) which was purchased for the manager's execution by its sponsor. Sets funds to the amount of money paid by the sponsor for that time slice.

ManagerLib_RequestGrantTotal: **procedure**() **returns**(success: boolean)

requires: ManagerLib interface has been initialized.
effects: Requests the total amount of money received by the manager from its sponsor. The total is computed and delivered in an APPLICATION_GRANT_TOTAL message. Returns true if successful, false otherwise.

ManagerLib_RegisterHandler: **procedure**(messageCode: integer,
handler: procedure) **returns**(success: boolean)

requires: ManagerLib interface has been initialized.
effects: Defines handler to be the application procedure which is called to handle a pending message of type messageCode. The handler is invoked when ManagerLib_HandleMessage() is called to dispatch to the appropriate message handler. Returns true if successful, false otherwise.
The valid messageCodes are:

APPLICATION_MESSAGE,
APPLICATION_ABORT,
APPLICATION_ATTACH,
APPLICATION_DETACH,
APPLICATION_GRANT_TOTAL,
APPLICATION_EXTEND,
APPLICATION_MANAGER_BIDDING,
APPLICATION_MANAGER_ACTIVE,
APPLICATION_WORKER_BIRTH,
APPLICATION_WORKER_DEATH

ManagerLib_PendingMessage: **procedure**(timeOut: integer) **returns**(success: boolean)

requires: ManagerLib interface has been initialized.

effects: Returns true if any incoming messages are available. If no messages are waiting, blocks (without busy-waiting) until:
(1) a message is available
(2) an interrupt (e.g. signal from OS) occurs, or
(3) timeout occurs after timeOut milliseconds
If timeOut is nil, the call will never time out. Returns false if a timeout or interrupt occurs.

ManagerLib_HandleMessage: **procedure**() **returns**(success: boolean)

requires: ManagerLib interface has been initialized.

effects: If no incoming messages are available, returns false. Otherwise, causes a dispatch to the appropriate handler procedure specified in an earlier call to ManagerLib_RegisterHandler(). If no appropriate handler was previously defined, the message is ignored. Returns true if a message was handled.

- *Special Root Manager Routine Specifications*

ManagerLib_RootInitialize: **procedure**(userName, groupName, taskMapPath: string) **returns**(success: boolean)

requires: Must be called prior to calling any other ManagerLib routines.

effects: Initializes the Spawn Manager interface for a root manager. The strings userName and groupName specify user and group identifications, respectively, for the root manager. The network path taskMapPath specifies the location of the task file used to resolve abstract task names into concrete executable files. Returns true if successful, false otherwise.

ManagerLib_RootAbort: **procedure**() **returns**(success: boolean)

requires: ManagerLib interface has been initialized for root manager.

effects: Starts a recursive abort of root manager and all submanager processes. Any unused funds are returned to the top-level user's bank account prior to the application's termination. Returns true if successful, false otherwise.

ManagerLib_RootSetFunding: **procedure**(reserve, dropSize, period: real) **returns**(success: boolean)

requires: ManagerLib interface has been initialized for root manager.

effects: Sets the top-level funding for the root manager to be $dropSize every period seconds up to a total of $reserve. The money is withdrawn from the root's local bank account. Overrides all previous calls to this routine. Returns true if successful, false otherwise.

ManagerLib_RootDirectFunding: **procedure**(amount: real) **returns**(success: boolean)

requires: ManagerLib interface has been initialized for root manager.

effects: Delivers $amount as a single funding drop to the root manager. Note that this is in addition to any periodic funding drops specified via ManagerLib_RootSetFunding(). The money is withdrawn from the root's local bank account. Returns true if successful, false otherwise.

Appendix B

A Spawn Application

B.1 Overview

This appendix presents a detailed look at a sample *Spawn* application, a concurrent Monte-Carlo simulation. The application is written in pseudo-code with Algol-like syntax and standard constructs from block-structured languages such as C and Pascal (e.g., `begin-end` blocks, `while` and `for` loops, and `if-then-else` conditionals). All parameter-passing is call-by-reference. For simplicity, we assume the existence of a primitive `HashTable` abstraction which internally uses dynamic memory allocation for the insertion of new items.

The documentation style for procedural interfaces is identical to the format used in Appendix A. Other comments are italicized and preceded by bullets (•).

B.2 Worker Module

- *Spawn Application Worker Module*
- *A Worker for Distributed Monte-Carlo Computations.*

- *Types*

```
MonteCarloState: record
                   • Application-dependent state.
                 end
```

- *Global State*

```
TerminateFlag: boolean
```

Interrupt: **procedure**()
modifies: TerminateFlag (global)
effects: Sets global termination flag in response to an asynchronous termination signal from the operating system.

```
begin
  • set global termination flag
  TerminateFlag := true
end
```

Terminate: **procedure**()
modifies: everything
effects: Cleans up and exits process.

```
begin
  • Close any open files, etc., and call Unix exit().
end
```

- *Specification only.*

ResetProgress: **procedure**(progress: MonteCarloState)
modifies: progress
effects: Initializes (clears) progress.

Initialize: **procedure**(newProgress: MonteCarloState)
modifies: newProgress
effects: Performs all necessary initialization. Clears newProgress.

```
begin
  TerminateFlag := false
  ResetProgress(newProgress)

  • Perform any other necessary initializations. For example,
    Initialize random number generator, using current time,
    process id to form seed.
end
```

• *Specification only.*

EncodeProgress: **procedure**(incrementalProgress: MonteCarloState
buffer: array of character, length: integer)

requires: buffer is large enough to hold encoded progress

modifies: buffer, length

effects: Uses Sun's external data representation (XDR) routines to encode the information in incrementalProgress in a machine-independent manner. Sets length to the size of the encoded data, and places the encoded results into buffer[1..length].

ReportProgress: **procedure**(incrementalProgress: MonteCarloState)

effects: Encodes incrementalProgress as an application message and delivers it to our manager.

```
buffer: array of character
length: integer

begin
  EncodeProgress(incrementalProgress, buffer, length)
  WorkerLib_MessageToSponsor(buffer, length)
  ResetProgress(incrementalProgress)
end
```

• *Main Program*

```
begin

  newProgress: MonteCarloState

  • Register Interrupt() procedure as handler for Unix SIGTERM signal.

  • Initialize
  WorkerLib_Initialize()
```

```
Initialize(newProgress)

• Main Loop

while true do
  • Compute a small number of monte-carlo trials.

  • If terminating, make final progress report and die.
  if TerminateFlag then
    ReportProgress(newProgress)
    Terminate()
  endif

  • Check if it's time to report progress.
    An application may decide, for example, to report progress
    every N trials (using a simple counter), or
    every T seconds (using a real-time alarm signal).
    if ( • time to report progress ) then
      ReportProgress(newProgress)
    endif

  endwhile

end
```

B.3 Manager Module

- *"Simple" Spawn Application Manager Module*
- *A Manager for Distributed Monte-Carlo Computations.*

- *This manager implements a very simple-minded funding strategy: Always maintain* MC_BRANCH *submanagers. If a funded submanager detaches or aborts, spawn another to take its place. This manager does not use relative progress or cost metrics in allocating funds.*

- *Types*

```
MonteCarloState: record
                   • Application-dependent state.
                 end
```

- *Constants*

```
MC_ALLOWANCE = 1.0
MC_BRANCH = 2
```

- *Global State*

```
NSubManagers, NFundedSubmanagers: integer
TaskFilePath, WorkDirPath: string
MgrArgs: array of string
NMgrArgs: integer
TerminateFlag: boolean
NReports: integer
Progress: MonteCarloState
```

Interrupt: **procedure**()

modifies: TerminateFlag (global)

effects: Sets global termination flag in response to an asynchronous termination signal from the operating system.

```
begin
  • set global termination flag
  TerminateFlag := true
end
```

Terminate: **procedure**()

modifies: everything

effects: Cleans up and exits process.

```
begin
  • Close any open files, etc., and call Unix exit().
end
```

• *Specification only.*
ResetProgress: **procedure**(progress: MonteCarloState)
modifies: progress
effects: Initializes (clears) progress.

Initialize: **procedure**(newProgress: MonteCarloState)
modifies: newProgress
effects: Performs all necessary initialization. Clears newProgress.

```
begin
  TerminateFlag := false
  ResetProgress(newProgress)

  NFundedSubmanagers := 0
  NSubmanagers := 0
  NReports := 0

  • Perform any other necessary initializations.
end
```

• *Specification only.*
EncodeProgress: **procedure**(incrementalProgress: MonteCarloState,
buffer: array of character, length: integer)
requires: buffer is large enough to hold encoded progress
modifies: buffer, length
effects: Uses Sun's external data representation (XDR) routines to encode the information in incrementalProgress in a machine-independent manner. Sets length to the size of the encoded data, and places the encoded results into buffer[1..length].

• *Specification only.*
DecodeProgress: **procedure**(incrementalProgress: MonteCarloState,
buffer: array of character, length: integer)
requires: buffer is large enough to hold encoded progress
modifies: incrementalProgress
effects: Uses Sun's external data representation (XDR) routines to decode the machine-independent progress information encoded by buffer[1..length], placing the decoded results in incrementalProgress.

ReportProgress: **procedure**(incrementalProgress: MonteCarloState)

effects: Encodes incrementalProgress as an application message and delivers it to our manager.

```
buffer: array of character
length: integer

begin
  EncodeProgress(incrementalProgress, buffer, length)
  ManagerLib_MessageToSponsor(buffer, length)
end
```

SpawnWorker: **procedure**()

effects: Spawns a local Monte-Carlo worker process.

```
begin
  ManagerLib_SpawnWorker(
    • "mc-worker" is an abstract task name which is mapped by
      the task file into the appropriate binary file for the
      machine where it is executed.
    "mc-worker",

    • This spawn request does not specify any command line
      arguments with which to invoke the "mc-worker" process.
    nil,
    0,

    • The "mc-worker" process is to be executed with WorkDirPath
      as its current working directory.
    WorkDirPath
    )
end
```

SpawnManager: **procedure**()

effects: Requests that a remote Monte-Carlo submanager process be spawned.

```
begin
  • Note: These arguments may be known statically as constants
    or can be computed dynamically as a function of command-line
    arguments and incoming messages.

  ManagerLib_SpawnManager(
    • "mc-manager" is an abstract task name which is mapped by
      the task file into the appropriate binary file for the
```

machine where it is executed.
`"mc-manager",`

- *There are NMgrArgs command-line arguments to "mc-manager", contained in the MgrArgs array. In this case we are simply passing our own command-line arguments to our submangers.*

`MgrArgs,`
`NMgrArgs,`

- *The "mc-manager" process is to be executed with WorkDirPath as its current working directory.*

`WorkDirPath,`

- *The abstract-to-concrete mappings for task names to executables to use in resolving task names for deeper spawn requests.*

`TaskFilePath,`

- *The requested submanager is not essential, so don't delegate responsibility for this task when we terminate if it is not yet active.*

`FALSE,`

- *WorkEstimate and WorkExtensionEstimate specify the estimated processing times for "mc-manager" (in seconds for a machine with a task-specific work-rating of unity).*

`WorkEstimate,`
`WorkExtensionEstimate,`

- *WarnTime sets the absolute amount of warning time (in seconds) that should be provided to "mc-manager" if it becomes necessary to forcibly terminate it.*

`WarnTime,`

- MC_ALLOWANCE *specifies the relative funding allowance for the new "mc-manager" task. Since each submanager is spawned with the same allowance, funding is divided evenly among them.*

`MC_ALLOWANCE,`

- *Setting the time weight to 1.0 and the price weight to 0.0 specifies a preference for scheduling "mc-manager" on*

```
        the machine with the soonest completion time, no matter
        what the price.
      1.0,
      0.0,

      • Finally, we specify that tasks owned by the same user may
        compete, but that tasks in the same group should not compete.
      ManagerLib_SelfGroupId(),
      FALSE,
      ManagerLib_SelfUserId(),
      TRUE
      )

    NSubmanagers := NSubmanagers + 1
    NFundedSubmanagers := NSubmanagers + 1
  end
```

• *Spawn Message Handlers*

HandleAttachMessage: **procedure**(taskId: integer, taskName: string)

modifies: NSubmanagers (global)

effects: Processes an APPLICATION_ATTACH message. Adjusts the global count of active submanagers.

```
  begin
    NSubmanagers := NSubmanagers + 1
  end
```

HandleDetachMessage: **procedure**(taskId: integer, allowance: real)

modifies: NSubmanagers, NFundedSubmanagers (globals)

effects: Processes an APPLICATION_DETACH or APPLICATION_ABORT message. Adjusts the global count of active submanagers. If a funded submanager has been lost (allowance > 0), a replacement submanager is spawned.

```
  begin
    NSubmanagers := NSubmanagers - 1
    if allowance > 0.0 then
      NFundedSubmanagers := NFundedSubmanagers - 1
    endif

    • If we lost a funded submanager, replace it with a new one.
    if NFundedSubmanagers < MC_BRANCH then
      SpawnManager()
```

```
    endif
  end
```

HandleProgressMessage: **procedure**(taskId: integer,
buffer: array of character, length: integer)

modifies: Progress (global)

effects: Processes an APPLICATION_MESSAGE message. Combines incremental progress encoded by buffer with aggregate Progress.

```
  incrementalProgress: MonteCarloState
  begin
    DecodeProgress(incrementalProgress, buffer, length)
    CombineProgress(Progress, incrementalProgress)
    NReports := NReports + 1

    • Report progress to higher management after NSubmanagers reports.
    if (NReports mod NSubmanagers) = 0 then
      ReportProgress(Progress)
      ResetProgress(Progress)
    endif
  end
```

• *Main Program*

```
begin
```

• *Register Interrupt() procedure as handler for Unix SIGTERM signal.*

• *Initialize*

```
ManagerLib_Initialize()
Initialize(Progress)
```

• *Set NMgrArgs, MgrArgs using information in command-line arguments. May also set various other state variables using this information.*

• *Register message handlers for dispatch on incoming messages.*

```
ManagerLib_RegisterHandler(APPLICATION_MESSAGE,HandleProgressMessage)
ManagerLib_RegisterHandler(APPLICATION_ATTACH, HandleAttachMessage)
ManagerLib_RegisterHandler(APPLICATION_DETACH, HandleDetachMessage)
ManagerLib_RegisterHandler(APPLICATION_ABORT,  HandleDetachMessage)
```

• *Allocate nothing for self preservation, giving all incoming funding to children.*

```
    ManagerLib_SetSelfAllowance(0.0)

    • Explicitly decline the purchase of extensions.
    ManagerLib_DeclineExtensions()

    • Get inherited path data from our own task description.
    ManagerLib_SelfPathInfo(WorkDirPath, TaskFilePath)

    • Spawn a local worker.
    SpawnWorker()

    • Request that MC_BRANCH remote managers be spawned.
    for i:integer := 1 to MC_BRANCH do
      SpawnManager()
    endfor

    • Process incoming messages.
    while true do

      • Block until message or interrupt is pending.
      if ManagerLib_PendingMessage(nil) then
        • Dispatch to appropriate registered handler.
        ManagerLib_HandleMessage()
      endif

      • If terminating, make final progress report and die.
      if TerminateFlag then
        ReportProgress(Progress)
        Terminate()
      endif

    endwhile

  end
```

B.4 A Smarter Manager Module

- *"Smart" Spawn Application Manager Module*
- *A Manager for Distributed Monte-Carlo Computations.*

- *This manager implements a more sophisticated funding strategy: Maintain at least* MC_BRANCH *submanagers, but only fund the least expensive* MC_BRANCH *submanagers at any given time. If at any time there are fewer than* MC_BRANCH *submanagers, spawn more. This manager uses cost metrics in allocating funds. A more sophisticated application manager could also take relative progress into account when allocating funds.*

- *Types*

```
Child: record
         taskId: integer
         unitPrice: real
       end

MonteCarloState: record
                   • Application-dependent state.
                   unitPrice: real
                 end
```

- *Constants*

```
MC_ALLOWANCE = 1.0
MC_BRANCH = 2
```

- *Global State*

```
TaskFilePath, WorkDirPath: string
MgrArgs: array of string
NMgrArgs: integer
TerminateFlag: boolean
NReports: integer
Progress: MonteCarloState

Children: HashTable of Child
PendingChildren: integer

FundedChildren: array of Child
NFunded: integer
```

Interrupt: **procedure**()

modifies: TerminateFlag (global)

effects: Sets global termination flag in response to an asynchronous termination signal from the operating system.

```
begin
  • set global termination flag
  TerminateFlag := true
end
```

Terminate: **procedure**()

modifies: everything

effects: Cleans up and exits process.

```
begin
  • Close any open files, etc., and call Unix exit().
end
```

• *Child Management Operations*

• *Specification only.*

CheapestChildren: **procedure**(children: HashTable of Child, n: integer, cheapest: array of Child, nCheapest: integer)

requires: cheapest has $\geq$ n elements.

modifies: cheapest, nCheapest

effects: Sets nCheapest = min(n, number of elements in children). Finds the nCheapest elements of children with the least expensive price per unit time. Sets the first nCheapest elements of cheapest to these children.

FundChildren: **procedure**(old, new: array of Child, nOld, nNew: integer)

effects: Changes the relative allocation of funding to children. Completely halts new funding to the nOld children in old. Funds each of the nNew children in new equally.

```
begin
  • A straight-forward implementation. Note that this could be
    optimized by avoiding any changes to the elements of old ∩ new.

  for i:integer := 1 to nOld do
    ManagerLib_SetFundingAllowance(old[i].taskId, 0.0)
  endfor

  for i:integer := 1 to nNew do
```

```
        ManagerLib_SetFundingAllowance(new[i].taskId, MC_ALLOWANCE)
      endfor
    end
```

AllocateFunding: **procedure**(children: HashTable of Child)

modifies: FundedChildren, NFunded (globals)

effects: Changes the relative funding allocations to fund the MC_BRANCH submanagers which are currently the least expensive. If fewer than MC_BRANCH submangers exist, spawns more.

```
  begin
    bestChildren: array of Child
    nBest: integer
    nSpawn: integer

    CheapestChildren(children, MC_BRANCH, bestChildren, nBest)
    FundChildren(FundedChildren, NFunded, bestChildren, nBest)

    • Update information about currently funded children.
    FundedChildren := bestChildren
    NFunded := nBest

    • If too few children, spawn some more.
    nSpawn := MC_BRANCH - nBest - PendingChildren
    if (nSpawn > 0)
      for i:integer := 1 to nSpawn do
        SpawnManager()
      endfor
    endif
  end
```

• *Specification only.*

ResetProgress: **procedure**(progress: MonteCarloState)

modifies: progress

effects: Initializes (clears) progress.

Initialize: **procedure**(newProgress: MonteCarloState)

modifies: newProgress

effects: Performs all necessary initialization. Clears newProgress.

```
  begin
    TerminateFlag := false
    ResetProgress(newProgress)
```

```
    HashTable_Initialize(Children)
    HashTable_Initialize(FundedChildren)
    PendingChildren := 0
    NFunded := 0
    NReports := 0

    • Perform any other necessary initializations.
  end
```

• *Specification only.*

EncodeProgress: **procedure**(incrementalProgress: MonteCarloState,
buffer: array of character, length: integer)

requires: buffer is large enough to hold encoded progress

modifies: buffer, length

effects: Uses Sun's external data representation (XDR) routines to encode the information in incrementalProgress in a machine-independent manner. Sets length to the size of the encoded data, and places the encoded results into buffer[1..length].

• *Specification only.*

DecodeProgress: **procedure**(incrementalProgress: MonteCarloState,
buffer: array of character, length: integer)

requires: buffer is large enough to hold encoded progress

modifies: incrementalProgress

effects: Uses Sun's external data representation (XDR) routines to decode the machine-independent progress information encoded by buffer[1..length], placing the decoded results in incrementalProgress.

ReportProgress: **procedure**(incrementalProgress: MonteCarloState)

effects: Encodes incrementalProgress as an application message and delivers it to our manager.

```
  buffer: array of character
  length: integer

  begin
    EncodeProgress(incrementalProgress, buffer, length)
    ManagerLib_MessageToSponsor(buffer, length)
  end
```

SpawnWorker: **procedure**()

effects: Spawns a local Monte-Carlo worker process.

```
begin
  ManagerLib_SpawnWorker(
    • "mc-worker" is an abstract task name which is mapped by
      the task file into the appropriate binary file for the
      machine where it is executed.
    "mc-worker",

    • This spawn request does not specify any command line
      arguments with which to invoke the "mc-worker" process.
    nil,
    0,

    • The "mc-worker" process is to be executed with WorkDirPath
      as its current working directory.
    WorkDirPath
    )
end
```

SpawnManager: **procedure**()

effects: Requests that a remote Monte-Carlo submanager process be spawned.

```
begin
  • Note: These arguments may be known statically as constants
    or can be computed dynamically as a function of command-line
    arguments and incoming messages.

  ManagerLib_SpawnManager(
    • "mc-manager" is an abstract task name which is mapped by
      the task file into the appropriate binary file for the
      machine where it is executed.
    "mc-manager",

    • There are NMgrArgs command-line arguments to "mc-manager",
      contained in the MgrArgs array. In this case we are simply
      passing our own command-line arguments to our submangers.
    MgrArgs,
    NMgrArgs,

    • The "mc-manager" process is to be executed with
      WorkDirPath as its current working directory.
    WorkDirPath,
```

- *The abstract-to-concrete mappings for task names to executables to use in resolving task names for deeper spawn requests.*

```
TaskFilePath,
```

- *The requested submanager is not essential, so don't delegate responsibility for this task when we terminate if it is not yet active.*

```
FALSE,
```

- *WorkEstimate and WorkExtensionEstimate specify the estimated processing times for "mc-manager" (in seconds for a machine with a task-specific work-rating of unity).*

```
WorkEstimate,
WorkExtensionEstimate,
```

- *WarnTime sets the absolute amount of warning time (in seconds) that should be provided to "mc-manager" if it becomes necessary to forcibly terminate it.*

```
WarnTime,
```

- MC_ALLOWANCE *specifies the relative funding allowance for the new "mc-manager" task. Since each submanager is spawned with the same allowance, funding is divided evenly among them.*

```
MC_ALLOWANCE,
```

- *Setting the time weight to 1.0 and the price weight to 0.0 specifies a preference for scheduling "mc-manager" on the machine with the soonest completion time, no matter what the price.*

```
1.0,
0.0,
```

- *Finally, we specify that tasks owned by the same user may compete, but that tasks in the same group should not compete.*

```
ManagerLib_SelfGroupId(),
FALSE,
ManagerLib_SelfUserId(),
TRUE
)
```

```
    PendingChildren := PendingChildren + 1
  end
```

• *Spawn Message Handlers*

HandleAttachMessage: **procedure**(taskId: integer, taskName: string)

modifies: Children (global)

effects: Processes an APPLICATION_ATTACH message. Constructs a Child structure for the newly attached task, and inserts it into Children. Adjusts relative funding allocation for children.

```
  c: Child
  begin
    c.taskId := taskId
    c.unitPrice := nil

    • Insert new child with key = taskId, data = c.
    HashTable_Insert(Children, taskId, c)
    AllocateFunding()
  end
```

HandleDetachMessage: **procedure**(taskId: integer, allowance: real)

modifies: Children (global)

effects: Processes an APPLICATION_DETACH or APPLICATION_ABORT message. Removes the Child structure for the newly detached task from Children. Adjusts relative funding allocation for remaining children.

```
  begin
    HashTable_Remove(Children, taskId)
    AllocateFunding()
  end
```

HandleProgressMessage: **procedure**(taskId: integer,
buffer: array of character, length: integer)

modifies: Progress (global)

effects: Processes an APPLICATION_MESSAGE message. Combines incremental progress encoded by buffer with aggregate Progress. Updates cost information for the child reporting progress. Adjusts relative funding allocation for remaining children.

```
  incrementalProgress: MonteCarloState
  begin
    DecodeProgress(incrementalProgress, buffer, length)
    CombineProgress(Progress, incrementalProgress)
```

```
    NReports := NReports + 1

    • Report progress to higher management after NSubmanagers reports.
    if (NReports mod NSubmanagers) = 0 then
      ReportProgress(Progress)
      ResetProgress(Progress)
    endif

    • Update unit price for child reporting progress.
    c := HashTable_Lookup(Children, taskId)
    if c ≠ nil then
      c.unitPrice := incrementalProgress.unitPrice
      AllocateFunding()
    endif
  end
```

HandleManagerBiddingMessage: **procedure**(taskId: integer)

modifies: Children (global)

effects: Processes an APPLICATION_MANAGER_BIDDING message. Constructs a Child structure for the new task, and inserts it into Children. Adjusts relative funding allocation for children.

```
  c: Child
  begin
    c.taskId := taskId
    c.unitPrice := nil

    • Insert new child with key = taskId, data = c.
    HashTable_Insert(Children, taskId, c)
    AllocateFunding()
  end
```

HandleExtendMessage: **procedure**(length, funds: real)

modifies: Progress (global)

effects: Processes an APPLICATION_EXTEND message. Sets our own unit price to funds / length.

```
  begin
    • Update global unit price information.
    if length ≠ 0.0 then
      Progress.unitPrice := funds / length
    endif
```

• *Release all extra money in reservoir to children.*
```
ManagerLib_ReleaseReservoir(1.0)
end
```

• *Main Program*

```
begin
```

• *Register Interrupt() procedure as handler for Unix SIGTERM signal.*

• *Initialize*
```
ManagerLib_Initialize()
Initialize(Progress)
```

• *Set NMgrArgs, MgrArgs using information in command-line arguments. May also set various other state variables using this information.*

• *Register message handlers for dispatch on incoming messages.*
```
ManagerLib_RegisterHandler(APPLICATION_MESSAGE,HandleProgressMessage)
ManagerLib_RegisterHandler(APPLICATION_ATTACH, HandleAttachMessage)
ManagerLib_RegisterHandler(APPLICATION_DETACH, HandleDetachMessage)
ManagerLib_RegisterHandler(APPLICATION_ABORT,  HandleDetachMessage)
ManagerLib_RegisterHandler(APPLICATION_EXTEND, HandleExtendMessage)
ManagerLib_RegisterHandler(APPLICATION_MANAGER_BIDDING,
                           HandleManagerBiddingMessage)
```

• *Initialize global unit price information.*
```
ManagerLib_SelfPurchaseInfo(time, funds)
if time ≠ 0.0 then
  Progress.unitPrice := funds / time
endif
```

• *Allocate some money for self preservation.*
Divert a portion of incoming funding to reservoir for extensions.
```
ManagerLib_SetSelfAllowance(MC_ALLOWANCE)
```

• *Get inherited path data from our own task description.*
```
ManagerLib_SelfPathInfo(WorkDirPath, TaskFilePath)
```

• *Spawn a local worker.*
```
SpawnWorker()
```

```
    • Request that MC_BRANCH remote managers be spawned.
    for i:integer := 1 to MC_BRANCH do
      SpawnManager()
    endfor

    • Process incoming messages.
    while true do

      • Block until message or interrupt is pending.
      if ManagerLib_PendingMessage(nil) then
        • Dispatch to appropriate registered handler.
        ManagerLib_HandleMessage()
      endif

      • If terminating, make final progress report and die.
      if TerminateFlag then
        ReportProgress(Progress)
        Terminate()
      endif

    endwhile

  end
```

B.5 Root Manager Module

• *Root Application Manager Module*
• *A Top-Level Manager for Distributed Monte-Carlo Computations.*

• *Note: A root application manager is nearly identical to an ordinary application manager. The important differences are: (1) the root manager presents progress to the top-level user, and (2) the root manager controls the overall amount and rate of funding that is "pumped" into the sponsorship hierarchy. Only the code for these special root operations is given below; the remainder of the program is identical to the ordinary application manager code.*

ReportProgress: **procedure**(incrementalProgress: MonteCarloState)
modifies: Progress (global)
effects: Combines incrementalProgress with global Progress. Processes top-level aggregate data (e.g. saving it to a file, displaying it to top-level user, etc.)

```
begin
  • Log aggregate progress to user-interface or file.
end
```

• *Main Program*

```
begin

  userName, groupName, localTaskFilePath: string

  • Register Interrupt() procedure as handler for Unix SIGTERM signal.

  • Set NMgrArgs, MgrArgs, localTaskFilePath, userName, and
    groupName using information in command-line arguments.
    May also set various other state variables using this information.

  • Initialize
  ManagerLib_RootInitialize(userName, groupName, localTaskFilePath)
  Initialize(Progress)

  • Register message handlers for dispatch on incoming messages.
  ManagerLib_RegisterHandler(APPLICATION_MESSAGE,HandleProgressMessage)
  ManagerLib_RegisterHandler(APPLICATION_ATTACH, HandleAttachMessage)
  ManagerLib_RegisterHandler(APPLICATION_DETACH, HandleDetachMessage)
  ManagerLib_RegisterHandler(APPLICATION_ABORT,  HandleDetachMessage)
  ManagerLib_RegisterHandler(APPLICATION_EXTEND, HandleExtendMessage)
```

```
ManagerLib_RegisterHandler(APPLICATION_MANAGER_BIDDING,
                           HandleManagerBiddingMessage);

• Initialize global unit price information.
  The root manager runs for free on the owner's workstation.
Progress.unitPrice := 0.0

• Allocate nothing for self preservation, giving all incoming
  funding to children.
ManagerLib_SetSelfAllowance(0.0)

• Get inherited path data from our own task description.
ManagerLib_SelfPathInfo(WorkDirPath, TaskFilePath)

• Do not spawn any local workers.

• Request that MC_BRANCH remote managers be spawned.
for i:integer := 1 to MC_BRANCH do
  SpawnManager()
endfor

• Control incremental funding via ManagerLib or top-level Spawn
  user-interface. This call specifies linear funding. Explicit,
  discrete funding drops may also be specified via calls to
  ManagerLib_RootDirectFunding(amount).
ManagerLib_SetRootFunding(budget, dropSize, period)

• Process incoming messages.
while true do

  • Block until message or interrupt is pending.
  if ManagerLib_PendingMessage(nil) then
      • Dispatch to appropriate registered handler.
    ManagerLib_HandleMessage()
  endif

  • If terminating, make final top-level progress report and die.
  if TerminateFlag then
    ReportProgress(Progress)
    Terminate()
  endif
```

```
    endwhile

end
```

B.6 Application Task File

In order to execute an application, the programmer must specify a *task file*, which maps abstract task names and machine types into concrete network path names for executable files. An abstract name and machine configuration is resolved using the first matching entry in the task file. The general format is as follows:

```
task  abstract-task-name
  cpu   hardware-family hardware-model hardware-special
  bin   nfs-style-pathname
  work  machine-specific-rating

  cpu  ...

task ...
```

A task file for the sample Monte Carlo application is given below. It indicates that executable versions of the **mc-manager** and **mc-worker** tasks exist for Sun3/140, Sun4/110, and Sun4/260 workstations. Furthermore, the relative power of each hardware configuration for each task is specified in the **work** field. Note, for example, that a Sun3/140 with floating point hardware (**fp**) runs this particular application twice as fast as a Sun3/140 without special hardware (* matches anything).

```
task mc-manager

        cpu sun3 140 fp
        bin pooh:/usr/waldspurger/bin/sun3/mc-manager
        work 0.4

        cpu sun3 140 *
        bin pooh:/usr/waldspurger/bin/sun3/mc-manager
        work 0.2

        cpu sun4 110 *
        bin bear:/u/huberman/bin/mc-manager
        work 1.0

        cpu sun4 260 *
        bin bear:/u/huberman/bin/mc-manager
```

```
        work 1.4

task mc-worker

        cpu sun3 140 fp
        bin pooh:/usr/waldspurger/bin/sun3/mc-worker
        work 0.4

        cpu sun3 140 *
        bin pooh:/usr/waldspurger/bin/sun3/mc-worker
        work 0.2

        cpu sun4 110 *
        bin bear:/u/huberman/bin/mc-worker
        work 1.0

        cpu sun4 260 *
        bin bear:/u/huberman/bin/mc-worker
        work 1.4
```

Bibliography

[Agh86] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, 1986.

[Bar85] A. Barak and A. Shiloh. A distributed load-balancing policy for a multicomputer. *Software Practice and Experience*, 901–913, September 1985.

[Cho82] T. C. K. Chow and J. A. Abraham. Load balancing in distributed systems. *IEEE Transactions on Software Engineering*, 401–412, July 1982.

[Cof73] Edward G. Coffman and Peter J. Denning. *Operating Systems Theory.* Prentice Hall, Englewood Cliffs, NJ, 1973.

[Dal89] William J. Dally. The J-machine: System support for actors. In Carl Hewitt and Gul Agha, editors, *Concurrent Object Programming for Knowledge Processing: An Actor Perspective*, MIT Press, Cambridge, MA, 1989.

[Dav83] Randall Davis and Reid G. Smith. Negotiation as a metaphor for distributed problem solving. *Artificial Intelligence*, 20:63–109, 1983.

[Dre88] 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.

[Ell85] J. R. Ellis. *Bulldog: A Compiler for VLIW Architectures.* PhD thesis, Yale University, February 1985. YALEU/DCS/RR-364.

[Fer88] 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.

[Fri84] Daniel Friedman. On the efficiency of experimental double auction markets. *American Economic Review*, 24(1):60–72, March 1984.

[Hai88] 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.

[Hew85] Carl Hewitt. The challenge of open systems. *Byte*, 10:223–242, April 1985.

[Hew86] Carl Hewitt. Offices are open systems. *ACM Transactions on Office Information Systems*, 4(3):271–287, July 1986.

[Hub88] 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.

[Kah88] 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.

[Kah89] Kenneth M. Kahn and Vijay A. Saraswat. *Money as a Concurrent Logic Program.* Technical Report, Xerox PARC, 1989.

[Kep88] Jeffrey O. Kephart, Tad Hogg, and Bernardo A. Huberman. *Dynamics of Computational Ecosystems.* Technical Report, Xerox PARC P88-00106, 1988.

[Kep89] Jeffrey O. Kephart, Tad Hogg, and Bernardo A. Huberman. Dynamics of computational ecosystems: Implications for DAI. In M. N. Huhns, editor,

Distributed Artificial Intelligence, Vol. 2, Morgan Kaufmann, Los Altos, CA, 1989.

[Len83] 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.

[Lis83] 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.

[Lis86] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development.* MIT Press, Cambridge, MA, 1986.

[Lis88] B. Liskov. Distributed programming in Argus. *Communications of the ACM*, 31(3):300–312, March 1988.

[Lo84] Virginia M. Lo. Heuristic algorithms for task assignment in distributed systems. In *International Conference on Distributed Computing Systems*, pages 30–39, IEEE, 1984.

[Lo87] 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.

[Mal83] T.W. Malone, R. E. Fikes, and M. T. Howard. *Enterprise: A Market-like Task Scheduler for Distributed Computing Environments.* Working Paper, Xerox PARC, Cognitive and Instructional Sciences Group, October 1983.

[Mal87] T. W. Malone, R. E. Fikes, K. R. Grant, and M. T. Howard. *Market-like Load Sharing in Distributed Computing Environments.* Working Paper 139, MIT Sloan School of Management, Center for Information Systems Research, April 1987.

[Mal88] 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.

[Man87] Carl R. Manning. *Acore: An Actor Core Language.* MPSG Apiary Design Note 7, MIT AI Laboratory, August 1987.

[Mil87] 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.

[Mil88] 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.

[Nel81] Bruce Nelson. *Remote Procedure Call.* Ph.D. Thesis, Technical Report CSL-79-3, Xerox Palo Alto Research Center, 1981.

[Par83] F. N. Parr and R. E. Strom. NIL: A high-level language for distributed systems programming. *IBM Systems Journal*, 22(1/2), 1983.

[Par87] Thomas S. Parker and Leon O. Chua. Chaos: A tutorial for engineers. *Proceedings of the IEEE*, 75(8):982–1008, August 1987.

[Pas88] Joseph Carlo Pasquale. *Intelligent Decentralized Control in Large Distributed Computer Systems.* PhD thesis, University of California, Berkeley, April 1988.

[Pet85] James L. Peterson and Abraham Silberschatz. *Operating Systems Concepts.* Addison-Wesley, Reading, MA, 2nd edition, 1985.

[Ras88] R. F. Rashid. From Rig to Accent to Mach. In Bernardo A. Huberman, editor, *The Ecology of Computation*, North Holland, Amsterdam, 1988.

[Sho82] 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.

[Smi80] 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.

[Smi81] Reid G. Smith and Randall Davis. Frameworks for cooperation in distributed problem solving. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-11(1), January 1981.

[Sob75] I. M. Sobol. *The Monte Carlo Method.* Mir Publishers, Moscow, 1975.

[Sut68] I. E. Sutherland. A futures market in computer time. *Communications of the ACM*, 11(6):449–451, June 1968.

[Tan85] Andrew S. Tanenbaum and Robert Van Renesse. Distributed operating systems. *ACM Computing Surveys*, 17(4):419–470, December 1985.

[Tha82] C. P. Thacker, E. M. McCreight, B. W. Lampson, R. F. Sproull, and D. R. Boggs. Alto: a personal computer. In Siewiorek, Bell, and Newell, editors, *Computer Structures: Principles and Examples*, pages 549–572, McGraw-Hill, New York, 1982.

[The83] Daniel G. Theriault. *Issues in the Design and Implementation of Act 2.* Technical Report, MIT AI Laboratory AI-TR-728, 1983.